Universidad Zaragoza

Trabajo Fin de Grado

Generación procedural de niveles para videojuegos action-RPG

Procedural level generation in action-RPG videogames

Autor:
Fabián Conde Lafuente (776127@unizar.es)

Director:
Carlos Bobed Lisbona (cbobed@unizar.es)

Grado de Ingeniería Informática

Escuela de Ingeniería y Arquitectura
2021/2022

23 de septiembre de 2022

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Fabián Conde Lafuente ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de Grado en Ingeniería Informática (Título del Trabajo)

Generación procedural de niveles para videojuegos action-RPG

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 23 de septiembre de 2022

Fdo:

Agradecimientos.

A mi familia y amigos por todo el apoyo recibido durante tantos años.

A Carlos Bobed por ayudarme con todas las duda y por orientarme a lo largo de todo el proyecto.

A Eduardo Mena por confiar en el proyecto.

Resumen

Este Trabajo de Fin de Grado nace de un proyecto personal. A lo largo de mi vida siempre me han interesado los videojuegos, y me encantaría dedicarme a este campo en el futuro. Junto a la asignatura de Videojuegos, este es mi primer contacto directo con esta carrera y con el motor de desarrollo de videojuegos de Unity. Inicialmente, se propuso al profesor de la asignatura de Videojuegos, que tras mostrar interés me puso en contacto con Carlos Bobed. Tras presentarle la propuesta, se acordó el alcance del trabajo para ajustarlo a las horas adecuadas para un Trabajo de Fin de Grado.

Se busca dar comienzo a las bases de un videojuego de acción con elementos de rol (estadísticas de personajes y subida de nivel) y elementos de *roguelike*[1]. Nos centraremos sobre todo en los elementos de roguelike. Dichos juegos suelen generar sus niveles de forma aleatoria en forma de laberintos divididos en pisos, los cuales aumentan en dificultad según se baja. En nuestro caso, buscamos trasladar estos conceptos sobre un mundo abierto[2] (en concreto una isla), el cual se encuentrará dividida en diferentes zonas con una dificultad diferente.

El trabajo consta de tres objetivos: Primero, la generación de un mundo abierto. Este mundo se encontrará dividido en biomas, cada uno de estos contendrá elementos, como puede ser el tipo de vegetación que aparece. Segundo, la creación de un algoritmo de generación de dificultad que sea progresiva, esto quiere decir que se busca que la dificultad crezca con las capacidades de lucha del usuario. Y tercero, se planteará un plantilla para la creación de los personajes, objetos y habilidades que se crearán en futuras versiones de este proyecto. Estos objetivos se tratarán a los largo de los cuatro capítulos en los que se divide este trabajo. En el primer capítulo se realiza una introducción, señalando el contexto, objetivos y tecnologías utilizadas. En el segundo se analizan los requisitos del proyecto. En el tercero, se estudió y se diseñó el prototipo. En el cuarto capítulo se describe la implementación del prototipo que se entrega junto a este trabajo, que debido al alcance del proyecto, se entrega una versión inicial, centrada en la generación del terreno, hay elementos del motor que han quedado en progreso y serán añadidos en el futuro.

En conclusión, en este TFG se han logrado todos los objetivos planteados. Considero que este trabajo ha sido un empuje muy grande a lo que me gustaría dedicarme en el futuro. Añadir, también que realizar un TFG de un proyecto personal es algo que puede ser muy duro para un estudiante ya que marcas tú el nivel de detalle de los objetivos y depende mucho de tí el resultado, sin embargo es algo muy reconfortante, ya que una idea que tuviste se ha hecho realidad.

[1]Subgénero de los videojuegos de rol que se caracterizan por una aventura a través de laberintos, a través de niveles generados por procedimientos al azar.

[2]Un videojuego de mundo abierto es aquel que ofrece al jugador la posibilidad de moverse libremente por un mundo virtual y alterar cualquier elemento a su voluntad.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Los videojuegos están cada vez más presentes en nuestra vida y cada día son más demandados. A veces ciertas empresas de la industria, ya sea por escasos recursos o por no tener suficiente tiempo o por simplemente ahorrar recursos, hacen usos de algoritmos procedurales, que nos permiten generar entornos, personajes, texturas, entre otras cosas, de forma aleatoria. Uno de los principales casos de uso que es el que se quiere explorar en este trabajo, es el uso de algoritmos procedurales para la generación de mundos. Algunos ejemplos de videojuegos exitosos que hacen uso de estos algoritmos son Minecraft o The Binding Of Isaac.

Este tipo de algoritmos nos permiten crear juegos casi infinitos con multitud de escenarios diferentes. Además, en comparación con el desarrollo tradicional, da la posibilidad de crear juegos de una forma más rápida y con un coste menor para las empresas, a la vez que a los jugadores les da la posibilidad de jugar múltiples veces siempre encontrándose escenarios totalmente diferentes.

En este trabajo se busca explorar cómo podemos generar mundos haciendo usos de algortimos procedurales para la creación de un motor que nos permitiese crear de un videojuego de acción con elementos de RPG (Role Playing Games) y roguelike. Además, se quiere desarrollar un sistema de combate y personajes similares a juegos de acción y aventuras como Dark Souls y The Legend of Zelda; se utilizarán sistemas de estadísticas con mejoras de estas como en los RPG; y se generará el terreno de forma aleatoria.

Vamos a investigar cómo crear un entorno abierto interactuable por el jugador, es decir, un mundo abierto, que se genere de manera aleatoria, incluyendo generación de enemigos, objetos que aparecen en el mapa y habilidades que el jugador puede tener disponible durante la partida. Esto se trata de un enfoque novedoso ya que los videojuegos roguelike normalmente tratan de la exploración de mazmorras.

Este proyecto se encuentra ampliamente basado en los videojuegos The Legend of Zelda, Dark Souls, The Binding of Isaac y Minecraft. La creación de biomas, el diseño de personajes, habilidades disponibles, etc se encontrarán basadas en estos juegos y adaptadas de alguna manera al motor que queremos desarrollar.

Dado el alcance del proyecto, no se podrá entregar una versión completamente terminada. Se acota la entrega a una versión inicial que será ampliada posteriormente por el alumno como proyecto personal. Esta versión del proyecto contiene una versión completamente funcional centrada en la generación del terreno, por lo que hay elementos del motor que han quedado en progreso y serán añadidos en el futuro.

1.1. Contexto

Este proyecto se trata de un proyecto personal. A lo largo de mi vida siempre he mostrado interés por los videojuegos y en el futuro me gustaría dedicarme al diseño y desarrollo de videojuegos. Junto al proyecto de la asignatura de videojuegos, este proyecto se trata de un primer contacto con esta carrera. Además, es mi primer contacto directo con el motor de desarrollo de videojuegos Unity, que es uno de los más utilizados a día de hoy junto a Unreal Engine. Teniendo esto en mente, desarrollé una propuesta de proyecto sobre un juego de acción con generación procedural de niveles. Esta propuesta se presentó inicialmente al profesor de la asignatura de videojuegos, Eduardo Mena, que tras mostrar interés me puso en contacto con el director de este TFG, Carlos Bobed Lisbona. Tras presentarle la propuesta, este aceptó la dirección de este TFG, y dada la ambición del motor completo, acordamos un alcance para ajustarlo más a las horas adecuadas para un Trabajo de Fin de Grado.

1.2. Objetivos del proyecto

En este proyecto se busca desarrollar un prototipo de videojuego de acción en un mundo abierto, que se trata de un entorno (o mundo) donde el jugador se puede mover libremente e interactuar con él a su voluntad. Este mundo abierto ha de poderse generarse de manera aleatoria, es decir, que a través de una semilla se nos permita desarrollar mundos con elementos aleatorios determinados por una serie de procedimientos que definirán cómo se construye. Adicionalmente, este proyecto necesita el desarrollo de un algoritmo de dificultad que determinará qué enemigos aparecen en el mapa.

Los objetivos del proyecto se pueden agrupar en tres puntos: generación del mundo, desarrollo de un algoritmo de generación de dificultad y plantear una plantilla para la construcción de personajes, objetos y habilidades.

1. **Generación del mundo**.

El objetivo es generar una isla que funcionará como mundo abierto, donde el jugador se moverá libremente y podrá interactuar por el entorno. En la isla existirán diferentes tipos de biomas, creando así variedad en el terreno. Cada bioma tendrá diferentes características que afectarán al terreno: tipos de árboles, presencia de flores, arbustos y hierba, color de la tierra, cantidad de vegetación y color de la vegetación y agentes externos que modifiquen el terreno (por ejemplo, lluvia).

Se investigarán diferentes tipos de algoritmos procedurales para la generación de terrenos, para luego estudiar cuáles de éstos son los más adecuados para el motor.

Además se buscará plantear para desarrollo futuro una manera de construir poblados y estructuras distribuidas por el escenario. Esto dará al usuario un mayor número de interacciones con el mapa, como podría ser lugares de combate, comercio e interacciones con personajes. Con estos edificios se podrán generar también caminos que unan los diferentes puntos del mapa.

2. **Desarrollo de un algoritmo de generación de dificultad.**
   En los videojuegos la dificultad de un enemigo depende de múltiples factores, los cuales son: posición del enemigo (por ejemplo, si un enemigo que aparece a la espalda del jugador es más complicado defenderse), estadísticas y habilidades del enemigo y la inteligencia del enemigo (cómo usa sus herramientas). Ésta suele ser una medida subjetiva y depende del tipo de jugador al que quiera el diseñador del juego enfocarse. Para un mayor ajuste de la dificultad, el nivel de dificultad de un enemigo deberá asignarse manualmente.

   En este trabajo se buscará investigar cómo adaptar la dificultad a través de la definición de un algoritmo que determine la cantidad y tipo de enemigos que se pueden generar, y otro que asigne la dificultad a cada zona de forma justa para el jugador. Algo a tener en cuenta es la necesidad de una dificultad gradual y progresiva, esto quiere decir, que la dificultad vaya creciendo con acorde al jugador y que nunca encuentre un reto desproporcionado para éste.

3. **Plantilla para la construcción de personajes, objetos y habilidades.**
   En nuestro motor se va a presentar una plantilla para el desarrollo de personajes del juego. Esta plantilla debe tener en cuenta que los personajes pueden realizar acciones y habilidades y tienen estadísticas de combate. Las acciones se refieren a por ejemplo: correr, rodar, atacar, defender, etc. Las habilidades corresponden con el lanzamiento de magia curativas o ofensivas (como podrá ser el lanzamiento de una bola de fuego). Las estadísticas de combate se refieren a cuánto daño inflige y reciben los personajes, algunos ejemplos son: puntos de vida, puntos de estamina, ataque, ataque mágico y defensa física y defensa mágica.

Únicamente se diseñará la plantilla sobre la que se diseñarán los personajes. Esto se debe a la complejidad que supone el desarrollo de personajes, inteligencia artificial y el sistema de combate. Esto se considera no abordable en el presente proyecto.

1.3. Tecnologías utilizadas

El sistema operativo que se ha utilizado para todo el proyecto ha sido Windows 10. Se ha utilizado el motor de videojuegos Unity[1] para el desarrollo del prototipo y como entorno de desarrollo se ha utilizado Visual Studio 2019. Para el control de versiones utilizamos Github.

El lenguaje de programación se ha utilizado C# y se utiliza JSON para la carga y descarga de elementos del proyecto. Para el desarrollo de los modelos tridimensionales se ha utilzado Blender[2]. Para la creación de texturas se ha utilizado Aesprite[3] y para el diseños se ha usado Inkscape[4].

1.4. Cronograma

En este punto se enumeran las fases por las que ha pasado el proyecto, las cuales son:

1. Estudio sobre la generación de terreno y biomas en videojuegos y sobre la generación procedural de dificultad. Se estudió diferentes necesidades para la obtención de un mapa generado de manera procedural y como debería de funcionar la generación procedural. Para llevar este estudio a cabo se consultaron diferentes documentos y otras fuentes de información (como puede ser blogs y videos).

2. Análisis del motor. Habiendo realizado el estudió se realizó un análisis del motor para determinar cuáles eran los requisitos de este. A su vez se planteó una estructura. Posteriormente se diseñó el funcionamiento de los diferentes requisitos y se mostraron alternativas para finalmente dar con una óptima.

3. Implementación y pruebas del motor. Se implementaron los diferentes requisitos del motor y se realizaron pruebas para garantizar su funcionamiento.

4. Desarrollo de la demostración del motor. Se unieron cada una de las partes implementadas, para obtener una demostración de las capacidades del motor.

En la Figura 1.1 se estima el coste de cada una de las fases de cronograma. Este diagrama

[1]https://unity.com/ , accedido por última vez 18/09/22
[2]https://www.blender.org/ , accedido por última vez 18/09/22
[3]https://www.aseprite.org/ , accedido por última vez 18/09/22
[4]https://inkscape.org/es/ , accedido por última vez 18/09/22

se realizó al comienzo del desarrollo del proyecto, por lo que difiere de los tiempos reales que se han llegado a hacer para cada fase.

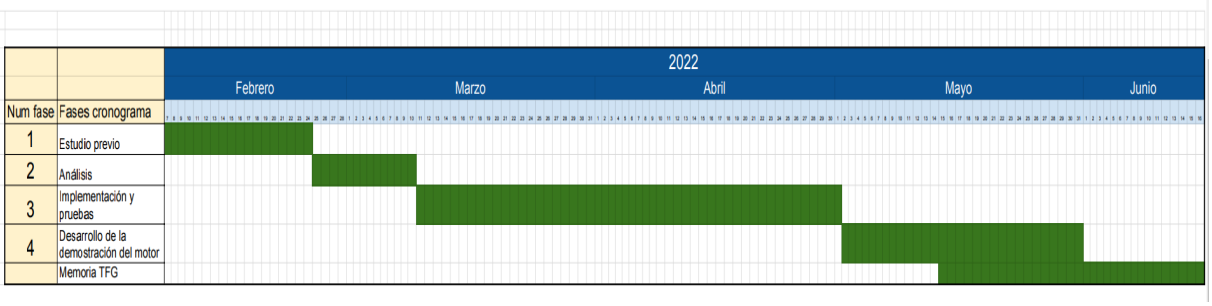


Figura 1.1: Diagrama de Gantt

1.5. Organización de la memoria

En este primer capítulo se ha llevado a cabo una pequeña introducción a modo de presentación del trabajo de fin de grado, explicando contexto, objetivos, fases del proyecto y tecnologías utilizadas. En el segundo capítulo del trabajo, se explica el trabajo de análisis del proyecto, analizando sus funcionalidades del proyecto y estructura de éste. En el tercer capítulo se entra en detalle del diseño del proyecto, se analizan las diferentes soluciones que podemos aplicar para realizar las funciones establecidas en el Capítulo 2. En el cuarto capítulo se detalla el prototipo y cómo se ha implementado cada una de sus componentes. En este caso nuestro prototipo permite generar entornos tridimensionales (con sus biomas y objetos) donde un modelo de jugador provisional se puede mover. El prototipo incluye la implementación del algoritmo de dificultad. En el capítulo cinco, se valora los resultados obtenidos, el trabajo a futuro y la valoración personal del proyecto. Finalmente en los anexos encontramos información adicional relacionada con el proyecto, tales como el manual del usuario, modelos y control de actividades entre otros.

Capítulo 2

Análisis del proyecto

En este capítulo se va a realizar un resumen de la fase de análisis realizada para este proyecto. En esta fase se estudió y analizó los objetivos del proyecto y se plantearon una serie de requisitos que este debería de cumplir, dada la longitud de estos se decidió realizar un resumen de estos[1]. Adicionalmente, se analiza la estructura del proyecto.

2.1. Requisitos del proyecto

Los requisitos los podemos agrupar en cuatro grupos diferentes. Uno por cada objetivo (generación de mapa, algoritmo de generación de dificultad y la plantilla de construcción de personajes, objetos y habilidades), y otro adicional para los requisitos de la partida.

1. **Partida** Los requisitos de partida son los requisitos que definen como el jugador interactúa con el entorno que le rodea. Este punto corresponde a la Tabla A.1 del Apéndice A de requisitos funcionales de partida.

   La partida comenzará apareciendo el jugador en un punto inicial y terminará al derrotar al jefe final en un punto final (RF 1-3). En cada partida, la posición de los puntos inicial y final son diferentes (RF 4). El jugador se moverá libremente por un mapa desconocido que descubrirá explorando (RF 5-6). Las partidas son repetibles, es decir, se puede volver a generar el mismo mapa siempre que el jugador quiera (RF 7).

2. **Generación de mundo** Los requisitos de generación de mundo definen cómo se genera el mundo y estructuras que rodea al jugador (como montañas, mares, árboles). Este punto corresponde a la Tabla A.2 del Apéndice A de requisitos funcionales de generación de mapa.

[1]Para ver la completitud de los requisitos y estructuras ver Apéndice A.

En cada partida se debe de poder generar un mapa aleatorio o generar uno que ya fue generado en el pasado (RF 13-14). Estos mapas tienen diversas alturas y se encuentran rodeados por agua, presentando biomas que dependen de la altura, humedad y temperatura (RF 15-20). En cada bioma se crearán elementos característicos de este (por ejemplo, árboles) (RF 21-22). Las estructuras se generan en el mapa y pueden ser interactuables (el nivel de interacción depende de la propia estructura, si se trata de un edificio, se debería de poder entrar a este) (RF 23-24).

3. **Algoritmo de generación de dificultad** Los requisitos del algoritmo de generación de dificultad definen los requisitos necesarios para el funcionamiento que debe de seguir el algoritmo. Este punto corresponde a la Tabla A.3 de requisitos funcionales del algoritmo de generación de dificultad del Apéndice A.

   La generación de dificultad se encuentra basada en la generación de laberintos de un roguelike. Aquí cambiamos el concepto de pisos que aumentan de dificultad, por celdas del mapa que crearán núcleos de dificultad (RF 27-28). La dificultad define la colocación de los enemigos y el número de éstos (RF 35 y 43). En el mapa existen dos celdas especiales, que son la celda de en la que aparece el jugador al inicio de la partida y la celda del final donde aparece el jefe final (RF 30-31). Las celdas se dividen en tres tipos de niveles de dificultad: Fácil, Medio y Difícil y existen reglas para cada una de los niveles de dificultad. En particular, cada celda podrá tener adyacente el mismo nivel de dificultad o uno por encima (RF 38-40). La celda inicial estará rodeada por celdas de dificultad Fácil y la final por celdas de dificultad Difícil (RF 36-37). Las celdas inicio y final deben de encontrarse a la distancia suficiente como para que el jugador pase por todas las dificultades en su camino (RF 41-42).

4. **Plantilla de construcción de personajes, objetos y habilidades** Los requisitos de la plantilla de construcción de personajes, objetos y habilidades definen cuales son las reglas que se deben seguir para la creación de personajes, objetos y habilidades. Este punto corresponde a las Tablas A.4, A.5 y A.6 de requisitos funcionales de partida del Apéndice A.

   Se va a dividir a los personajes en dos tipos: activos y pasivos. Los personajes activos tendrán un comportamiento definido, mientras que los pasivos no lo tendrán (RF 44-48). Cada personaje activo tendrá características de combate y podrán tener habilidades y objetos equipados limitados por un peso máximo (RF 49-50). Cada personaje puede realizar diversas acciones, correr, rodar, atacar, defenderse de un ataque, lanzar proyectiles, ejecutar habilidades e interactuar con el entorno (RF 52). Los objetos se dividen en tres grupos: activos, pasivos y mixtos. Los activos son objetos que otorgan a los personajes habilidades, los pasivos otorgan estadísticas y efectos y los mixtos otorgan ambos (RF 55-57). Existen tres tipos de habilidades:

activas, pasivas y mixtas. Las activas son conjuntos de acciones y efectos y subidas de estadística que pueden consumir maná y estamina al activarse. Estas tienen una duración y al terminarse necesitan un tiempo de refresco para poder volverse a usar. Las habilidades pasivas proporcionan efectos y subidas de estadística de manera pasiva (no es necesario activarlas y duran infinitamente) (RF 67-73).

2.2. Estructura del proyecto

El proyecto se divide en dos bloques. El primer bloque es la generación de mundo, compuesto por la generación de los mapas (mapas de elevación, humedad y temperatura), la generación del mapa de dificultad (que depende de los mapa generado ya que determina cómo se colocan los enemigos) y la creación de las estructuras. El segundo bloque corresponde a las estructuras de los personajes, objetos y habilidades.

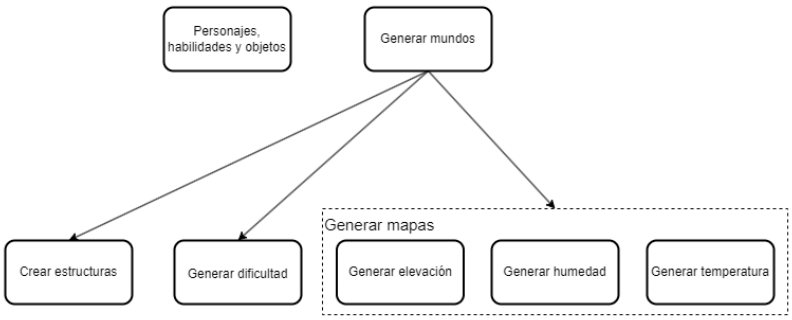


Figura 2.1: Estructura del proyecto

Capítulo 3

Estudio y diseño del proyecto

En este capítulo se explica el proceso de estudio y diseño de los diferentes objetivos del proyecto (ver Sección 1.2) y comprende dos fases: una fase de estudio de soluciones para los diferentes objetivos para obtener una idea general de cómo se pueden llevar a cabo los distintos objetivos; y otra fase en la que con ideas en mente, se diseñaron el comportamiento y funcionalidades de los objetivos presentando diversas alternativas y eligiendo las más óptimas.

La primera de las fases trató de la investigación previa al diseño de los diferentes objetivos y buscamos diferentes formas en las que se pueden obtener. Una vez se obtuvo un conjunto relevante de información se procedió a realizar el diseño. La información que se consultó proviene de multitud de fuentes que comprenden entre vídeos, blogs, tesis y documentos de investigación. Inicialmente se buscó información sobre cómo otras personas generaban los mundos abiertos [7][6][12][11][13]. Tras esto, se realizó una investigación sobre métodos para la generación de dificultad en diversos entornos utilizando diferentes algoritmos [2][22][1][14]. Finalmente, se buscó información sobre la generación de estructuras de manera procedural [8][16][5].

Este capítulo divide el contenido en tres secciones diferentes, que corresponden a los mismos objetivos del proyecto: generación del mundo, desarrollo de un algoritmo de generación de dificultad y plantear una plantilla para la construcción de personajes, objetos y habilidades.

3.1. Generación del mundo

Esta sección corresponde al objetivo de Generación del mundo de la Sección 1.2. Para la generación de mundo, queremos generar aleatoriamente islas, donde los jugadores se moverán libremente. Dentro de esta isla queremos que existan diferentes tipos de biomas,

los cuales deberán de tener diferentes tipos de elementos que los caractericen (por ejemplo: tipos de vegetación o color de la tierra).

La generación del mundo cuenta con tres pasos que se desarrollan en las siguientes subsecciones: Generación de terreno, Generación de biomas, Generación de estructuras.

3.1.1. Generación de terreno

Necesitamos generar un terreno aleatorio. Una de las formas para la generación de terrenos es a través de mapas de ruido. Un mapa de ruido se trata de un mapa en escala de grises (de valores 0 a 1) generado de manera pseudo-aleatoria, que simula un ruido estático generado a través de un valor de entrada (dada una entrada se generará un mapa). Utilizaremos estos mapas de ruido como base para generar mapas que necesitemos, como mapas de altura, humedad o temperatura. Posteriormente, y dado que queremos generar islas, investigaremos formas sobre las que a partir de un terreno podemos obtener una isla.

Mapas de ruido

La generación del mapa de ruido se encuentra basada en el post "Making maps with noise functions" de Red Blob Games [4]. Este es uno de los blogs más populares y utilizados en los referente a la generación aleatoria de terreno. En él se explica paso a paso cómo se puede obtener un terreno tridimensional generado proceduralmente. En nuestro caso seguiremos este blog para la obtención de mapas de ruido.

En este blog utilizan el ruido de Perlin (ver Sección del E.1 Apéndice E). Sin embargo, existen multitud de alternativas a ruido de Perlin, entre las que destacan:

- Ruido Simplex [24]. Algoritmo de obtención de ruido de Perlin, pero este se encuentra patentado.
- Ruido cúbico [19]. De acuerdo con Job Talle [19], se trata de una versión escalada del ruido blanco utilizando interpolación cúbica.
- Ruido de Diamond-Square [9] [23]. Tipo de ruido que genera mapas que contienen 2^n+1 puntos de ancho y 2^n+1 de alto. Crea un mapa de ruido utilizando el algoritmo Diamond-Square, que genera una nube fractal.
- Ruido de Worley [25]. Se trata de un algoritmo para la creación procedural de texturas. Es especialmente útil para la generación de texturas de piedra, agua o células biológicas.

Cualquiera de las alternativas es válida. No obstante, en nuestro caso nos quedaremos con

el ruido de Perlin por dos simples razones: primero es la solución más utilizada, por lo que podemos encontrar multitud de referencias sobre la generación de terrenos utilizando este algoritmo y, segundo, dado que usaremos Unity, ésta nos ofrece una función que lo implementa en la biblioteca Mathf.

Creación de la isla

Queremos obtener un mundo que sea una isla. Para conseguir ésta, utilizaremos el ruido de Perlin para generar un mapa de elevación Con esto obtendremos un mapa con valores entre 0 a 1 (siendo 0 el valor más bajo de elevación y 1 el más alto). El problema que nos encontramos es que es muy improbable encontrar un mapa de ruido que interpretado como mapa de elevación nos dé una mapa con una isla, por lo que deberemos de aplicar alguna función que nos permita obtenerla. Una posible solución pasa por encontrar una función que reduzca la altura de los puntos más próximos a los extremos del mapa. Estas funciones pueden estar basadas en la distancia de un punto con respecto al centro (o respecto a otro punto establecido) para así obtener un terreno con menor altura. Para ello se puede utilizar la distancia euclídea o de Manhattan. En nuestro caso, se utilizará como inspiración las funciones propuestas en el post de reddit del usuario KdotJPG sobre la generación de islas [7]. Todas las alternativas que se proponen son válidas, y en nuestro caso explicaremos cada una de éstas en el prototipo.

En nuestro caso exploraremos las funciones basadas en:

- Se utilizará la distancia euclídea (y de Manhattan) de un punto desde el centro para reducir la altura de manera proporcional (contra mayor sea la distancia más se restará a la altura) normalizada por la distancia máxima.
- Utilizando valores normalizados de las coordenadas de un punto con respecto a las anchura y la altura del mapa en concreto, para obtener un suavizado más abrupto que el que se consigue con la distancia.

3.1.2. Generación de biomas

Se quiere asignar a cada punto de la isla un bioma, esto nos permitirá aplicar al terreno modificaciones características de los biomas e instanciar elementos propios de este en el territorio. Existen diversas alternativas:

- Basados en aleatoriedad:
  - Diagrama de Voronoi: Se divide el espacio en regiones para luego ser asignadas aleatoriamente a un bioma.
  - Blobs: Se elige un punto aleatorio del mapa y se realiza una expansión de

este bioma hasta que algún identificador (por ejemplo, puntos que ya han sido expandidos en la iteración o con altura igual a cero consideradas como agua) determine que ya no puede expandirse más. Se repite un número de veces con diferentes biomas hasta que el usuario quiera.

- Basados en características del entorno:

  - Basado en altura: Se divide el mapa con respecto a la altura. Todos los puntos correspondientes a un intervalo de altura pasarán a pertenecer a un bioma concreto.

  - Realista: Un punto pertenece a un bioma dependiendo de su altura, humedad y temperatura.

Está información fue obtenida del post "How to randomly generate biome with perlin noise?" de Game Development [6] que pertenece a la página Stack Exchange. Se eligió la implementación realista. Se implementa una versión en el post "Making maps with noise functions" de Red Blob Games [4], pero esta no se ha usado como referencia para la selección de un bioma porque deja de lado la temperatura en la selección de un bioma.

Para obtener nuestra solución, se va a simular que la posición de la isla se encuentra en un punto aleatorio de un planeta. Esto se usará como punto de referencia para calcular el mapa de temperatura, la cual se calculará teniendo en cuenta la posición con respecto al ecuador y los polos de un punto del mapa. Además se deberá reducir la temperatura según la altura del punto. Los mapas de elevación y humedad se generarán utilizando el ruido de Perlin. En el caso de la humedad, también se tendrá en cuenta que la humedad desciende con respecto a la altura.

Para determinar que un punto pertenece a un bioma, se comprueba intervalos de altura, humedad y temperatura y se valorará a cual pertenece de acuerdo a una lista de biomas definidos. Basamos en el esquema de clasificación de zonas de vida de Holdridge[1], mostrado en la Figura 3.1.

[1]Imagen obtenida de Wikipedia, https://es.wikipedia.org/wiki/Bioma

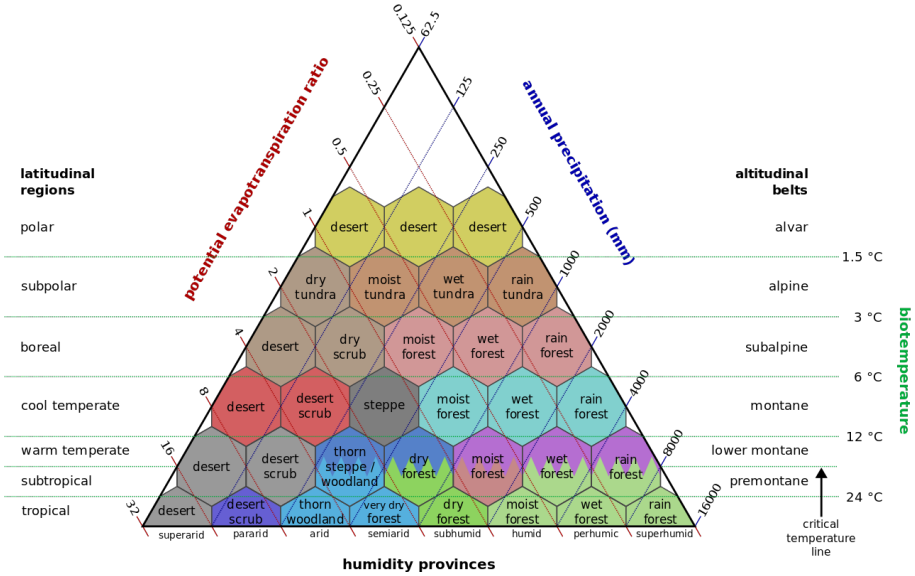


Figura 3.1: Esquema de clasificación de zonas de vida de Holdridge

3.1.3. Generación de estructuras

Queremos generar a lo largo del mapa estructuras como poblados, edificios individuales (como templos o granjas), puestos de vendedor o enemigos y caminos.

Para obtener la localización de las estructuras, se seleccionará celdas del mapa aleatoriamente. El número de estructuras que se va a seleccionar dependerá del tamaño del mapa. Para la generación de los poblados se utilizará una variación del diagrama de Voronoi. En el post de Reddit "Fantasy medieval cities for the monthly challenge"[21] comentan como debería de funcionar esta variación. Del mapa elegiremos un lugar aleatorio sobre el cual queremos crear un poblado y elegiremos puntos aleatorios en este que se tratarán de edificios. Utilizaremos el diagrama de Voronoi para crear polígonos sobre el poblado y cada una de estas aristas corresponderá con un camino. Los edificios se crearán alrededor de estos caminos. Los poblados estarán limitados por la celda o celdas dependiendo del tamaño y los bordes se generarán de manera aleatoria. El número de edificios determinará lo grande que es un poblado. Tanto los edificios individuales como los edificios de los poblados serán generados de la misma manera. Los elementos necesarios para generar un edificio serán generados aleatoriamente. Utilizando la triangulación de Delaunay, sobre todos los poblados del mapa crearemos los triángulos y con sus aristas construiremos los caminos, y se introducirá un ruido sobre cada una camino para crear una camino más realista.

3.2. Generación de dificultad

En esta sección discutiremos cómo se va a gestionar la dificultad en un entorno generado en su totalidad aleatoriamente. Para ello debemos establecer algoritmos que nos ayuden a valorar cómo colocar las celdas neutrales y enemigas.

El objetivo de la generación de dificultad es encontrar una algoritmo que nos permita crear una dificultad justa y progresiva, es decir, se ajusta la dificultad que enfrenta un jugador según progresa por el mapa.

Se va a dividir el mapa en celdas y cada una de las celdas tendrá asociado un nivel de dificultad. Estas celdas serán independientes de los biomas. Para asignar sobre cada celda una dificultad tenemos que tener en cuenta que van a existir tres niveles de dificultad, Fácil, Media y Difícil, y que existen los puntos de Inicio y Final que deberán de estar rodeados por dificultades Fácil y Difícil respectivamente. Para asegurar una curva de dificultad justa se debe asegurar de que la progresión de la dificultad sea escalada y no de saltos entre dificultades. Imaginemos que un jugador inicia la partida en un punto, en su camino al jefe final debe de pasar por zonas que vayan incrementando de dificultad en un nivel, mantenerse o disminuir de dificultad. En resumen, existen reglas de para los niveles de dificultad que pueden tener celdas contiguas, por ejemplo, no podremos acceder de una celda Fácil a una celda de dificultad Difícil sin haber pasado una media.Otra regla que se considera esencial es que en su camino el jugador pase por todas las dificultades.

Dicho esto tenemos diferentes opciones de algoritmos para generar esta dificultad. A continuación se explicará cómo funciona cada uno:

1. Restricciones o reglas: En este algoritmo se va a suponer que se tiene un conjunto de reglas que definen cuales son las dificultades admisibles de las celdas colindantes, por ejemplo si tenemos la regla Fácil las celdas colindantes a una solamente pueden ser Fácil o Medio (en ningún caso Difícil). Elegido una dificultad de una celda, podemos seleccionar una de las dificultades posibles aleatoriamente o por alguna condición como el número total de celdas marcadas con esa dificultad.

2. Clústeres: En este algoritmo se eligen celdas que actuarán como centroide de un clúster, luego se le asigna a cada una de las restantes celdas cuál es el clúster más cercano. A cada una de las clústeres se le asigna una dificultad. Sobre este algoritmo se puede realizar algunas modificaciones, por ejemplo:

    - Al seleccionar un centroide podemos asignarle la dificultad y, en vez de ajustarlos en un bucle, realizar una progresión de dificultades entre dos centroides adyacentes.

    - Podemos asignar sobre cada centroide una valor que decida su área de influencia

sobre el territorio.

Este algoritmo nos permitiría crear áreas de dificultad que podrían ser interesantes para crear una mapa más consistente. Tiene principalmente un problema, pueden existir celdas que tengan la misma distancia de varios centroides, a estas las llamaremos celdas disputadas. Podemos solucionar esto utilizando el valor de influencia de cada clúster (caso de la modificación), o dandole prioridad a una dificultad (por ejemplo, en caso de disputa seleccionar la menor dificultad).

3. Diagrama de Voronoi: Se eligen diferentes celdas como centros a lo largo del mapa, luego se aplica el algortimo que genera diagramas de Voronoi a partir de puntos en el mapa. Esto divide el mapa en áreas poligonales según la proximidad de cada punto, a cada área se le asociará una dificultad. Todas las celdas que se encuentren en un triángulo tendrán la misma dificultad.

En nuestro caso utilizaremos el segundo algoritmo, añadiendo a los clústeres un valor de influencia que disminuirá con respecto a la distancia del clúster. Un punto pertenece a un clúster dependiendo del valor de influencia con respecto a este (eligiendo pertenecer siempre al mayor).

3.3. Diseño de personajes, objetos y habilidades

En esta sección explicamos cómo es la plantilla de objetos, habilidades y personajes que deberán de seguir cada uno de estos.

3.3.1. Objetos

Los objetos se dividen en dos tipos: consumibles y equipables. Los objetos consumibles son aquellos que son de un solo uso. Principalmente existen diferentes tipos, por ejemplo: pociones de vida, de maná, cura estados o llaves. Los objetos equipables son objetos que te dan mejoras a las estadísticas de combate y/o habilidades. Cada objeto tendrá un peso asignado.

3.3.2. Habilidades

Las habilidades son otorgadas por un objeto equipable. Existen tres tipos diferentes de habilidades: habilidades pasivas, activas y mixtas. Las habilidades pasivas se encuentran activas todo el rato y producen efectos de forma continua sobre el jugador. Los efectos que puede producir son mejoras a las estadísticas o la aplicación de algún estado alterado al golpear a un enemigo. Las habilidades activas necesitan ser activadas por el jugador, esto significa que solo se ejecutarán cuando el jugador lo indique. La activación de estas puede consumir maná, en el caso de hacerlo si no se dispone del suficiente maná no pueden

ser activadas. Al activarlas entra en enfriamiento (tiempo de espera hasta el siguiente uso). Las habilidades activas dan mejoras de estadísticas, aplican efectos de combate y/o ejecutan acciones. Existe una tercera que es una combinación de ambas. Las habilidades mixtas tienen funcionalidades de las activas y de las pasivas.

Las habilidades pueden aplicar estados alterados a quien los recibe. Existen diferentes estados alterables que pueden ser aplicados sobre un personaje. En la Tabla 3.1 se explican los diferentes estados alterados que existen.

Estado alterado	**Descripción**
Quemadura	Inflige daño por segundo y reducir la defensa física del personaje.
Envenenamiento	Inflige daño por segundo.
Confusión	Para los enemigos provoca que se mueva aleatoriamente por el terreno y para el jugador invierte los controles.
Ralentización	Reduce la velocidad de movimiento y velocidad de ataque.
Congelado	Congela al personaje y le impide realizar acciones. Los personajes también pueden salir del estado de congelación realizando input de movimiento.
Aturdimiento	Provoca que el personaje no se pueda mover durante un tiempo. Todo ataque recibido durante este estado rompe el aturdimiento y será crítico.

Cuadro 3.1: Estado alterados que afectan a los personajes

3.3.3. Tipos de personajes

Para los personajes existen dos clasificaciones:

- Personajes activos o pasivos. Los personajes activos son todos aquellos personajes que tienen una inteligencia artificial y pueden realizar acciones y tienen características de combate. Pueden llegar a interactuar con el jugador de otras formas que no sean la lucha. Los personajes pasivos no tienen características de combate, y por lo tanto no se puede luchar con ellos.

- Personajes controlables o no controlables. Viene determinado por si el jugador puede controlarlos o no controlarlos de alguna manera.

Todos los personajes activos poseen las estadísticas de combate definidas en el Tabla 3.2. Los personajes podrán realizar una o varias de estas acciones como se marca en la Tabla 3.3.

Estadística	Descripción
Vida	Vitalidad del personaje. Determina cuántos golpes puede recibir un personaje.
Ataque físico.	Representa el daño de contacto que puede infligir un personaje con un arma, proyectil o habilidad.
Ataque mágico	Representa el daño que puede infligir un ataque de naturaleza mágica con un arma, proyectil o habilidad.
Defensa física	Reduce el daño físico recibido por un ataque físico.
Defensa especial	Reduce el daño mágico recibido por una ataque mágico.
Velocidad de movimiento	Velocidad con la que se mueve un personaje sobre el terreno.
Velocidad de ataque	Velocidad de descanso entre ataques con armas.
Maná.	Determina el número de habilidades que un personaje puede lanzar. Cada habilidad consume una cantidad de maná diferentes. Si no se dispone de maná, no será posible lanzar habilidades. Esta estadística está únicamente relacionada con el jugador.
Regeneración de maná	Regeneración de la estadística maná. Determina la cantidad de maná que se regenera por segundo. Esta estadística está únicamente relacionada con el jugador.
Estamina	Determina el número de acciones que un personaje puede realizar. Si el personaje se queda sin estamina no podrá realizar acciones. Si tiene poca estamina puede realizar acciones independientemente del coste de esta.
Regeneración de estamina	Regeneración de la estadística estamina. Determina la cantidad de estamina que se regenera por segundo.

Cuadro 3.2: Estadísticas de combate de personajes activos

Estadística	Descripción
Correr	El jugador se mueve más rápido hacia la dirección en la que se mueve. Consume Estamina.
Rodar	El personaje rueda hacia la dirección en la que esté mirando. Con esta acción se puede esquivar un ataque (físico o mágico) y no recibir daño. Consume Estamina.
Defender	El personaje se defiende para parar un golpe físico. Consume Estamina. Si al defender el golpe se queda sin estamina el personaje se rompe su defensa y el personaje queda aturdido.
Atacar con arma	Ataca con un arma cuerpo a cuerpo. Existen varios tipos de ataques con armas cuerpo a cuerpo. Consume Estamina.
Lanzar proyectiles	Lanza un proyectil. Consume Estamina.
Lanzar ataques especiales o habilidades	Ejecuta una habilidad. Consume Estamina y/o maná.

Cuadro 3.3: Acciones que los personajes activos pueden realizar

Existen dos clases de personajes activos, son héroe y enemigos. Cada personaje poseerá un inventario de objetos asociados y, con estos, una serie de habilidades. El héroe es el personaje activo controlable por el jugador. Este tiene un inventario limitado que determina el número de objetos que puede llevar por el peso. Será decisión del jugador elegir cuáles objetos quiere tener en el inventario. Los objetos que den habilidades solo pueden ser cargados si se dispone de espacio de habilidades suficiente. El jugador podrá seleccionar qué habilidades quiere ser capaz de realizar. Los enemigos, al contrario que el héroe, no dispone de un máximo número de objetos y habilidades. El número de habilidades y objetos que lleva dependerá de la dificultad del enemigo. Al morir pueden dejar caer objetos. Los objetos que lleva determinan qué objetos podrá soltar el usuario al derrotarlo.

Debido al alcance del proyecto las plantillas de los objetos, habilidades y personajes no se llegaron a implementar. Aunque se decidió añadir como objetivo hasta la fase de diseño por completitud. En un futuro como proyecto personal estás serán implementadas.

Capítulo 4

Implementación del prototipo

El prototipo desarrollado para este TFG cubre la implementación de la generación del mapa y el algoritmo de dificultad. La generación del mapa construye un mapa tridimensional con biomas rodeado por agua, le aplica distintos agentes que modifican su elevación e instancia diferentes objetos sobre él. El algoritmo de dificultad divide el mapa en celdas y le asigna una dificultad con respecto a dos puntos aleatorios. En este punto vamos a explicar como funciona la función principal del prototipo que se encarga de generar el mundo y ejecuta el algoritmo de dificultad.

4.1. Descripción en alto nivel

La funcionalidades del prototipo se pueden resumir en una sola función que es la encargada de generar el mundo. En el siguiente pseudocódigo veremos de manera básica como funciona este prototipo y en la siguiente sección analizaremos el funcionamiento de cada una de la lineas.

```
void GenerarMundo (int seed, int RealMapSize) {
    Random.inicializar_semilla(seed);  // Sección 4.2.1
    MapData mapData = GenerarMapa(RealMapSize); // Sección 4.2.2
    GenerarBiomas(); // Sección 4.2.3
    AplicarAgentesModificadores(); // Sección 4.2.4
    CrearModeloTerreno(); // Sección 4.2.5
    PoblarBiomas (); // Sección 4.2.6
    GenerateMapaDificultad(mapData); // Sección 4.2.7
    AplicarTexturas(); // Sección 4.2.8
}
```

4.2. Descripción detallada del prototipo

4.2.1. Inicialización de la semilla

Unity contiene la clase *Random*, esta clase se puede inicializar con una semilla y a partir de una semilla se puede generar la misma secuencia de números aleatorios.

Nota: Durante todo esta sección se va a suponer que todos los valores aleatorios se generan en una secuencia especifica, ya que el mapa no podría ser reproducible si se generan los datos de forma diferente.

4.2.2. Generación de los mapas

La función de generación de mapa sigue el diagrama de flujo de la figura 4.1.

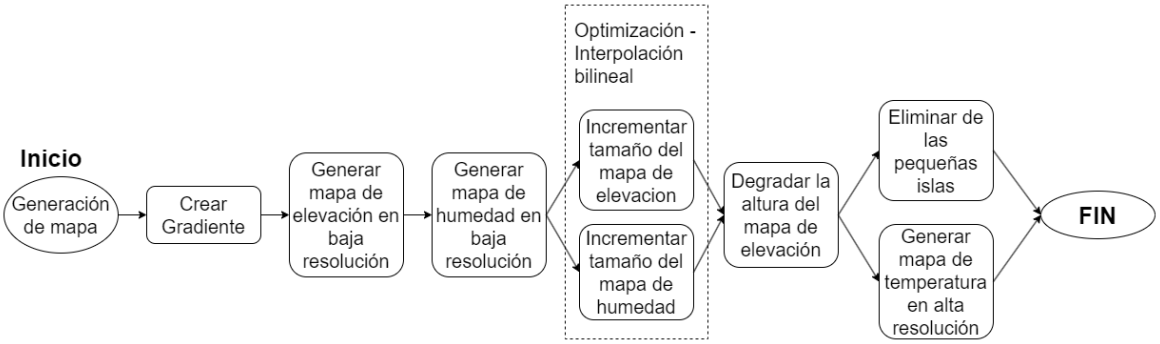


Figura 4.1: Diagrama de flujo de la generación de mapa

En los siguientes apartados vamos a describir cada uno de los pasos que sigue el prototipo para la generación de los mapas.

Mapa de ruido

Vamos a generar los mapas de ruidos basandonos en ruido de Perlin. Unity ofrece una función que genera este ruido: **public static float PerlinNoise(float x, float y);**

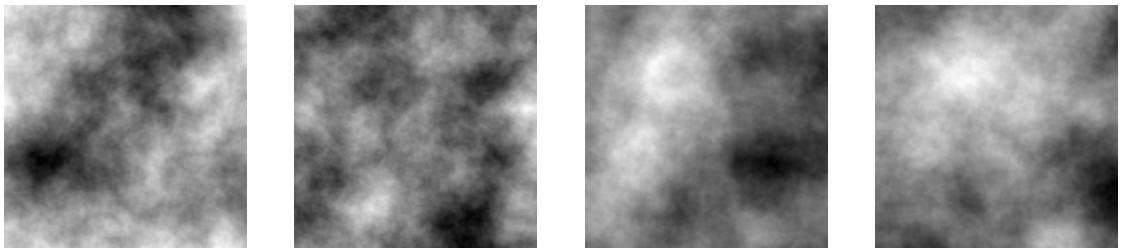
La función necesita de un valor x e y que se representarán como las posiciones x e y del punto del mapa sobre el cual queremos obtener un ruido. Para obtener consistencia este se normaliza los valores x e y para obtener siempre mismo mapa independientemente del tamaño, y para añadir aleatoriadad se suma un valor aleatorio x e y (introduciendo una semilla estos valores serán siempre iguales).

Para añadirle más variabilidad a la generación podemos añadirle octavas al terreno, es decir, mapas de ruido iguales al original con una frecuencia y amplitud, que determinarán

la frecuencia con la que aparecen picos en el mapa y la altura de cada punto. La frecuencia estará determinada por un parámetro introducido llamado *lacunarity* y la amplitud por el parámetro *persistence*, sus valores base son 2 y 0.5. Además, con el fin de introducir más aleatoriedad se puede añadir a cada octava un offset en x e y diferente.

Finalmente debemos de asegurarnos que estos valores se encuentren del 0 a 1, esto lo realizaremos en un bucle que itera a través de los valores del mapa aplicando la función *Mathf.InverseLerp(minNoiseValue, maxNoiseValue, noiseMap[x, y])*, que determina donde el valor cae en dentro de estos dos puntos devolviendo un valor del 0 al 1 según la proximidad con el máximo y el mánimo.

En la Figura 4.2 podemos observar los diferentes mapas de ruido que podemos obtener con estas valores y diferentes valores de la semilla.



(a) Mapa generado con semilla 2

(b) Mapa generado con semilla 532

(c) Mapa generado con semilla 863256

(d) Mapa generado con semilla 1520526

Figura 4.2: Ejemplos mapas de ruido de Perlin

Crear Gradiente

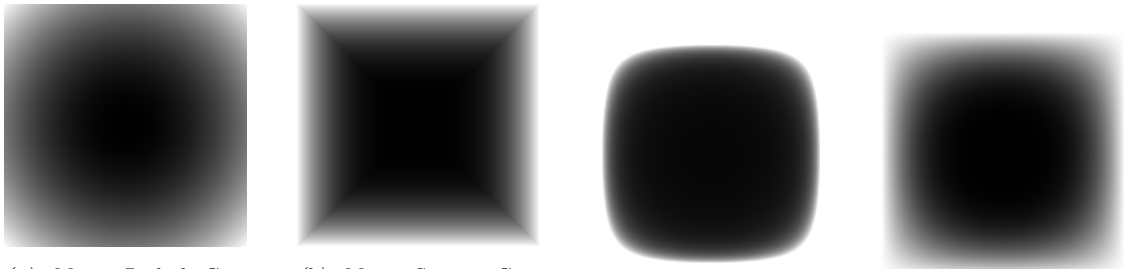
El gradiente se trata de crear un mapa en escala de grises (valores de 0 a 1) utilizados para la generación de islas a partir de una mapa de elevación. En el prototipo se implementaron varias versiones y en la Figura 4.3 se muestran los resultados de cada una éstas.

1. Radial Gradient: En un mapa de nxm se define un punto central, se recorre todos los puntos del mapa y se asigna el valor 1 - la distancia euclidea de ese punto.

2. Square Gradient: Se le asigna a cada punto un valor que depende de la distancia del punto con el centro y los bordes.

3. *Fall off map*: Asigna a cada punto un valor según su distancia con el centro y los bordes. Divide este valor por un estimación del tamaño de la isla y eleva a un exponente e el resultado. Finalmente ajusta los resultados entre 0 y 1.

4. Squared Bump: Aplica a cada punto de un mapa la formula: $valor(x, y) = 1 - (1 - nx^2) * (1 - nx^2)$. Las variables nx y ny se trata de la coordenadas x e y normalizadas en un rango de [-1,1]. Obtiene resultados al Square Gradient pero con las esquinas suavizadas.



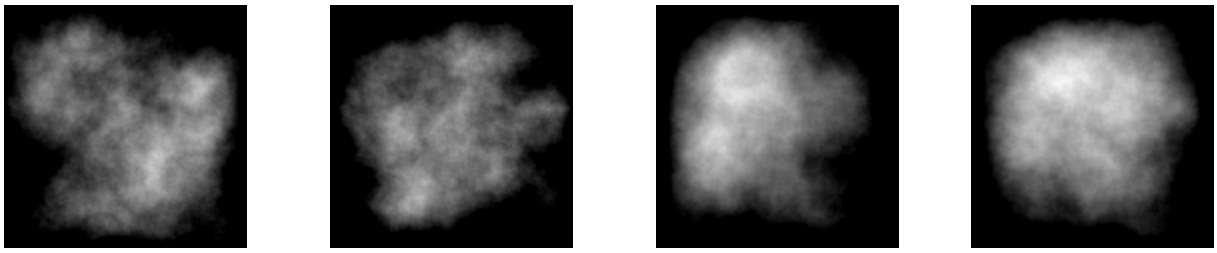
(a) Mapa Radial Gradient *(b) Mapa Square Gradiente* *(c) Mapa Fall off map* *(d) Mapa Squared Bump*

Figura 4.3: Ejemplos de mapa de gradientes

Todas estas son capaces de generar una isla. Los resultados de los gradientes 1,2, y 3 obtienen resultados equivalentes, se seleccionó *Square Bump* por los resultados visuales que proporcionaba.

Mapa de elevación en baja resolución

Para la generación del mapa de elevación generaremos una mapa de ruido basado en unos compensación (u offset en inglés) elegida aleatoriamente. Una vez con el mapa de ruido, le restamos a un mapa de gradiente. Los valores del mapa de elvación estarán contenidos de 0 a 1. Algunos ejemplos de mapas de elevación que podemos obtener son los que observamos en la Figura 4.4.



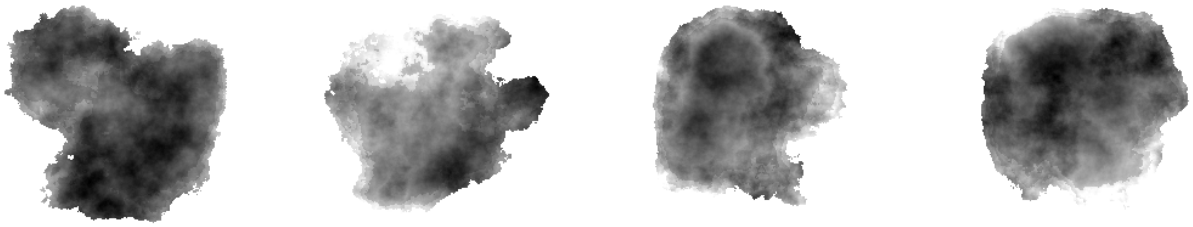
(a) Mapa generado con semilla 2 *(b) Mapa generado con semilla 532* *(c) Mapa generado con semilla 863256* *(d) Mapa generado con semilla 1520526*

Figura 4.4: Ejemplos mapas de elevación

Mapa de humedad en baja resolución

Para el mapa de humedad generaremos a su vez un mapa de ruido basado en offsets aleatorios (debemos asegurar que los offsets generados para este mapa de ruido sean diferentes a los del mapa de ruido del mapa de elevación, ya que queremos un mapa diferente). Luego, utilizaremos el mapa de elevación para limitar la humedad máxima de un punto con una altura h, ya que en un entorno real la humedad disminuye con la altura [20]. Limitaremos la humedad máxima como 1 - la elevación del punto, de tal manera que un punto con elevación 0.3, su minima humedad será de 0 y su máxima de 0.7.

En nuestro prototipo hemos considerado que todo punto con altura igual a 0, se corresponde con un punto de agua, por lo tanto su humedad debe ser máxima. Además, hemos añadido un multiplicador según cuan proximo este un punto del mar. Algunos ejemplos que podemos obtener son los que observamos en la Figura 4.5.



(a) Mapa generado con semilla 2

(b) Mapa generado con semilla 532

(c) Mapa generado con semilla 863256

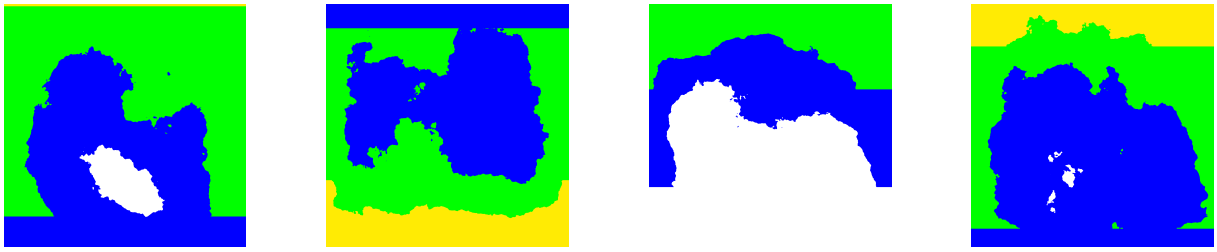
(d) Mapa generado con semilla 1520526

Figura 4.5: Ejemplos mapas de humedad

Mapa de temperatura

Para obtener los valores de temperatura se realizará una interpolación de la posición de la isla respecto a los dos polos, es decir, se establecerá un tamaño de la isla y se elegirá una posición aleatoria del centro de la isla en la latitud del planeta, y un vez hecho esto se interpolará sobre la posición del punto del mapa respecto a los dos polos y el ecuador. Para poder realizar esta interpolación es necesario calcular la temperatura máxima (temperatura del ecuador) y la temperatura mínima (temperatura para los polos). Para tener en cuenta los efectos de la altura sobre la temperatura se disminuirá 0.6^{o}C cada 100 metros de la isla [3]. Se necesita para esto declarar un multiplicador que transforme la altura de 0 a 1 a una medida en metros. Para obtener la temperatura un punto del mapa, en este prototipo supondremos que la temperatura depende de la posición con respecto a los polos y el ecuador y la elevación respecto al mar. En un modelo más realista podrían existir otros factores. Podemos ver ejemplos de los resultados obtenidos en la Figura 4.6.

A diferencia de los mapas de elevación y humedad, el mapa de temperatura ya lo generamos en alta resolución. Hacemos esto por las diferencias en el cálculo, en este caso solo iteramos una vez por todos lo puntos, mientras que en los otros mapa iteramos varias veces (en concreto depende del número de octavas).



(a) Mapa generado con semilla 2

(b) Mapa generado con semilla 532

(c) Mapa generado con semilla 863256

(d) Mapa generado con semilla 1520526

Figura 4.6: Ejemplos mapas de temperatura

Ampliar la resolución de los mapas

El cálculo de los mapas de elevación y humedad puede ser algo costoso en tiempo debido a la creación un mapa de gradientes más grande (los cálculos de los valores pueden llevar mucho tiempo) y a número de octavas del mapa, recordemos que para obtener un mapa se utiliza un triple bucle de anchura, altura y octavas.

Por esta razón se decidió trabajar con mapas de menor resolución para luego ampliarlos. Con esto obtenemos mapas iguales y en menos tiempo. Para ampliar la resolución hemos utilizado el algoritmo de interpolación bilineal. El cual nos permite interpolar un valor respecto al valor de dos variables de un mapa. Por ejemplo si tenemos un mapa bidimensional podemos generar valores interpolando desde puntos existentes en el mapa (con sus valores se calcula el nuevo valor).

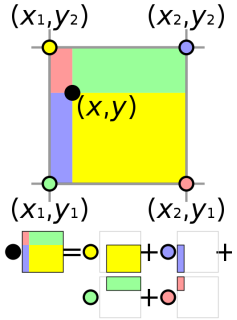


Figura 4.7: Ejemplo interpolación lineal

La Figura 4.7 explica de manera visual cómo funciona la interpolación bilineal: el valor que tendrá una punto situado en (x,y) dependerá de los valores de los cuatro puntos que se encuentran a su alrededor, un punto influirá más al valor de este según su proximidad.

Utilizando la interpolación bilineal creamos mapas de ruido de baja resolución para luego incrementarlos a la resolución que queremos. En la Tabla 4.1 podemos observar la comparación en tiempo del coste de las formas de crear mapas.

Número de prueba	Semilla	Tiempo (s) Mapa grande	Tiempo (s) Bilinear interpolation
1	2	2.783	0.3569
2	532	2.7919	0.3569
3	863256	2.8173	0.3569
4	1520526	2.8959	0.3569

Cuadro 4.1: Comparación en segundos de los costes en tiempo de los diferentes algoritmos

Degradación de la elevación

Al restar el mapa de ruido por el mapa de gradientes es posible que se generen saltos de altura muy grandes, para solucionar esto se aplica un suavizado en las alturas de los mapas degradando las alturas comprendidas entre el 0 a 0.15 utilizando una función exponencial.

Eliminación de islas pequeñas

Es posible que en el mapa de elevación se generen islas pequeñas alrededor de nuestra isla principal. En nuestro caso comprobamos si existen utilizando una algoritmo de etiquetación, etiquetaremos como 0 a todo punto con altura menor a 0 (considerada agua) y al resto de punto se etiquetará según el número de isla que sea. Al final de este algoritmo nos quedaremos con la isla con mayor número de puntos en ella y eliminaremos el resto de isla (se pone como su altura 0). Para etiquetar los puntos guardaremos en un diccionario los puntos que no han sido etiquetados. Cogeremos un punto aleatorio y le asignaremos una etiqueta, asignaremos a cada punto vecino esta etiqueta siempre y cuando no hayan sido etiquetados ya. Repetiremos hasta que no quede ninguno disponible.

Para no perder información, es necesario aplicar esto después del incremento del tamaño del mapa. En la Figura 4.8 el mapa con cada una de las islas marcadas de un color según su etiqueta.

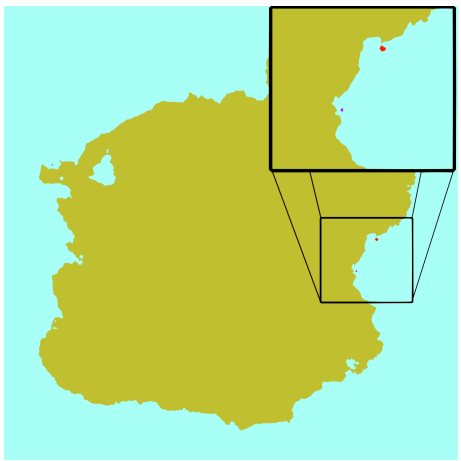


Figura 4.8: Mapa de etiquetas

4.2.3. Generar los biomas

Para cada uno de los chunk o trozos del mapa, queremos asignar a cada punto un bioma. Para esto se comparará cada uno de los valores de elevación, humedad y temperatura con una lista de biomas que pueden existir que contienen los valores mínimo y máximo que permiten.

4.2.4. Aplicar agentes modificadores

Tras ya tener un mapa completo de la isla sin discontinuidades aplicamos los agentes ambientales que sirven para poder añadir más diversidad y formas más complejas a los biomas (por ejemplo dunas y rastros de erosión generados por la lluvia). Para esto se creó una estructura que permitiese al usuario poder aplicar diversos agentes para añadir diversidad. Cada bioma contiene variables que determinan cómo afectan estos agentes sobre el terreno, produciendo diferentes resultados dependiendo del tipo del terreno.

Agente suavizador Sigue un algoritmo de *Paseo Aleatorio*, reduce la elevación de los puntos por los que pasa teniendo en cuenta los puntos que les rodea.

Agente erosión de lluvia Simula la erosión que produce la lluvia. Una gota de lluvia erosiona el terreno (reduce la altura de los puntos por los que cae) y se lleva sedimentos que va dejando por su camino. Las gotas se generan teniendo en cuenta la humedad del bioma donde cae y los efectos que puede provocar están determinados por las propiedades del punto donde se encuentra. Cada gota se mueve en dirección al punto de menor altura más cercano y terminará al llegar a él o al evaporarse. Para el desarrollo de este algoritmo se estudió la implementación de dos fuentes [15] [17], para luego implementarlas en el prototipo.

Agente erosión de viento Cada bioma tiene un porcentaje de terreno que está formado principalmente por sedimentos, y estos sedimentos pueden ser transportados por una partícula de viento. Cada una de las partículas de viento se generan desde los esquinas del mapa y recorren en línea recta hasta llegar al otro extremo. En su camino recogen sedimentos que utilizarán para erosionar el terreno y a su vez dejarán resto de sedimentos. La implementación de este agente se encuentra basada en "Simple Particle-Based Wind Erosion"del blog Nick's Blog [18].

4.2.5. Crear modelo del terreno

El modelo del terreno es el terreno por el que el jugador andará y sobre el cúal se generarán los elementos del mapa. La creación de éste se refiere a la creación de una malla de triángulos definida por los valores de la posición del mapa y su elevación (obtenida del

mapa de elevación). Para obtener este modelo se implementó un algoritmo de generación de mallas de triángulos basado en el video de Sebastian Lague de "Procedural Landmass Generation (E05: Mesh)"de su serie "Procedural Landmass Generation"[10]. En este algoritmo se recorren todos los puntos de izquieda a derecha y crean triangulos con sus puntos adyacentes siguiendo el setido de las agujas del reloj. En la Figura 4.9 se puede observar como funciona para un mapa de 3x3.

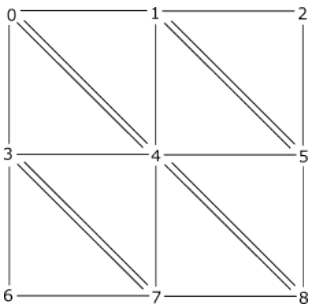


Figura 4.9: Imagen funcionamiento del algoritmo de generación de malla de triángulos en un mapa de 3x3

4.2.6. Poblar los biomas

Poblar un bioma significa añadir sobre el modelo del terreno generado en el punto anterior los diferentes elementos característicos de estos biomas (como árboles, hierbas o roca). El proceso se divide en dos partes. En la primera parte establecemos la posición, escala y rotación de los diferentes objetos de cada uno de los biomas. Los objetos que se encuentran en el mapa se eligen basándose en la probabilidad del objeto de aparecer (esta probabilidad es normalizada con respecto al número de objetos que pueden aparecer en un bioma) y la probabilidad del bioma de tener objetos. Estos objetos se guardan en una lista con sus características esperando ser instanciados en el mapa.

4.2.7. Generar mapa de dificultad

Se quiere generar un algoritmo de dificultad que divida el mapa en celdas y cree clústeres de celdas sobre el mapa. Para realizar esto debe de elegir el número de clústeres que van a existir en el mapa. Cada centro de clúster (que será una celda del mapa) tendrá asociado un valor que llamaremos intensidad. Utilizaremos este valor junto a la distancia entre las celdas para determinar a qué clúster pertenece una celda. El algoritmo de dificultad que hemos generado supone que existen tres tipos de dificultad: Fácil, Medio y Difícil, y dos clústeres especiales, inicio y final, que corresponden a los puntos de inicio de partida y final de partida. El pseudocódigo del algoritmo es el siguiente:

```
1 float[,] GenerarMapaDificultad (int n, int m, int numeroDificultades,
    float[,] elevación) {
2     float[,] mapaDificultad = new float[nCeldas, mCeldas];
3     byte[,] mascara = GenerarMascara(elevacion, nCeldas, mCeldas);
4     int numClusteres = 2 + GenerarNumeroAleatorio(numeroDificultades,10)
    ;
5     GenerarClusteresEspeciales();
6     AsignarPesoDificultades();
7     AsignarNumeroClusteresPorCadaDificultad();
8     GenerarAtributosClusteres();
9     ExpandirCluteres();
10    SeleccionarCluster();
11    FiltroConvolucion (mapaDificultad);
12    AplicarMascara(mapaDificultad, mascara)
13    return mapaDificultad;
14 }
```

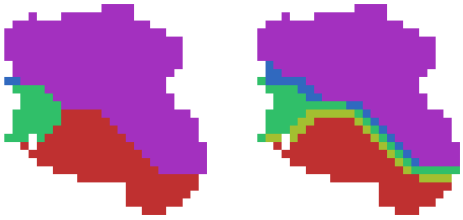
Línea 2: El mapa de dificultad se trata de una matriz de nxm que indica el nivel de dificultad de esta. Línea 3: Generamos una máscara que determina qué celdas pueden tener una dificultad asignada, no queremos que se utilicen las celdas que se encuentran en agua o mayormente en agua para generar enemigos. Línea 4: Definimos el número mínimo de clúster de dificultad que va a ver (variable numeroDificultades) y elegimos un número aleatorio entre este y una constante (en nuestra implementación hemos puesto un valor máximo de 10). Línea 5: Creamos los clústeres especiales inicio y final. Línea 6: Asignamos a cada tipo de dificultad un peso aleatorio (valor entre 0 y 1). Este peso determina el número de clústeres que va a existir para cada tipo de dificultad. Los clústeres de inicio y final solo existirán una única vez. Línea 7: En base al peso crearemos un número de clústeres para cada dificultad. Asignamos intensidad (potencia del clúster en su punto central), ratio de caída (como disminuye la intensidad según se aleja del punto inicial) y posición. Línea 8: Generamos los atributos de cada clústeres, intensidad del clúster sobre el punto según su distancia con el punto central, ratio de caída de la intensidad según la distancia y posición. Linea 9 y 10: Expandimos los clústeres y calcularemos el valor de intensidad de un clúster en una celda. Una vez calculado, una celda pertenecerá al clúster que tenga más intensidad sobre ella. Linea 11: Aplicaremos un filtro de convolución para suavizar la dificultad entre las celdas contiguas. Linea 12: Aplica la máscara para eliminar aquellas celdas a las cuales se le ha asignado dificultad y que no son válidas.

4.2.8. Aplicar texturas

Finalmente, para mejorar el aspecto visual del escenario generado, tenemos que aplicar las texturas al modelo tridimensional del terreno. En nuestro prototipo hemos asociado un coloar y una textura a cada bioma, puediendo tener dos formas de aplicar las texturas basándonos en ambos.

Aplicar texturas a partir de los colores de los biomas

A partir de los diferentes biomas del mapa se crea una textura, la cual representa el mapa de colores del terreno. Esta textura será la base de un material que se aplicará al modelo



(a) Sin aplicar Filtro de convolución *(b) Aplicando Filtro de convolución*

Figura 4.10: Ejemplo de resultado del algoritmo de dificultad

Nota: Los colores de la imagen anterior y que tipo de dificultad o clúster representan. Rojo = Inicio, Amarillo = Fácil, Verde = Medio, Azul = Difícil y Morado = Final

del terreno.

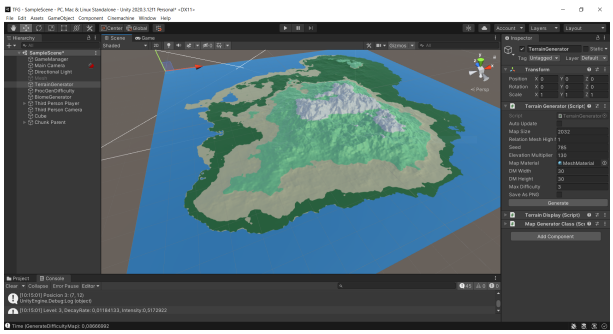
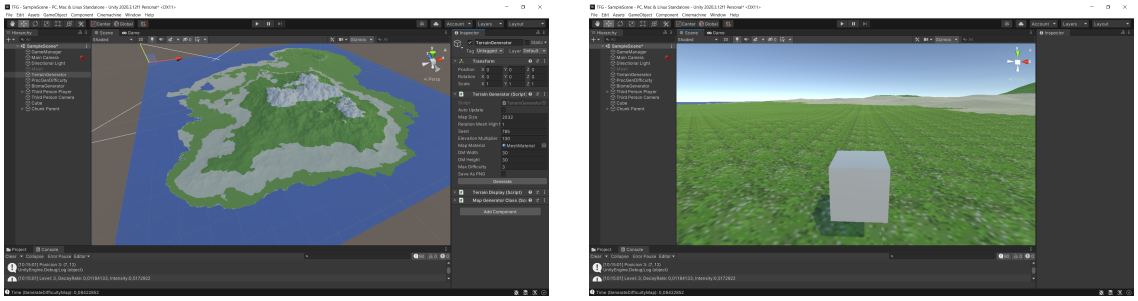


Figura 4.11: Mapa con una textura basada en los colores de los biomas

Aplicar texturas a partir de texturas de los biomas

En este caso crearemos los materiales para cada uno de los biomas. Para la aplicación de los materiales se debe de dividir el terreno en diferentes trozos en base al bioma y aplicar el material del bioma.



(a) Mapa visto desde una perspectiva general *(b) Mapa desde una perspectiva cercana*

Figura 4.12: Mapa con diferentes texturas según el bioma

Capítulo 5

Conclusiones

En este capítulo se van a mostrar las conclusiones obtenidas tras el desarrollo del Trabajo de Fin de Grado. Adicionalmente se comentará el trabajo a futuro del proyecto. Por último, se realiza una valoración personal sobre el trabajo realizado.

5.1. Resultados obtenidos

Todos los objetivos que fueron planteados al inicio del proyecto se cumplieron. En un primer momento a partir de las ideas que se entregaron, se acotó a los objetivos del proyecto que podemos observar en la Sección 1.2. Esto se llevó a cabo tras muchas conversaciones ya que se quería reducir el tamaño del proyecto por el alcance de este. Luego de esto, se escribieron los requisitos funcionales y no funcionales y se crearon unos diagrama de clase y paquete tempranos. Tras esto, se diseñaron las partes del proyecto (generación de mapas, algoritmo de generación de malla de triángulos, algoritmo de dificultad, creación de biomas, etc). Esto vino junto a una larga fase de estudio, ya que se que quería dar varias alternativas y encontrar en el papel cuál sería la más óptima. Con las partes diseñadas se pasó a la implementación y testeo de cada de éstas, que eventualmente se unió todo para formar el prototipo del proyecto.

El desarrollo del proyecto no ha tenido problemas mayores. Únicamente se pueden destacar dos problemas que se encontraron al querer añadir más contenido de cual fue inicialmente planeado. Los problemas vinieron con la implementación de los agentes y la aplicación de texturas. El primero de los problemas se debe a que el mapa original no estaba planeado para ser modificado, se tuvo que cambiar para que los agentes pudiesen actualizar el mapa. El segundo problema se debe a que se quería aplicar múltiples texturas sobre el modelo del terreno, dada mi inexperiencia con *shaders*, no fui capaz de lograrlo. Por lo que se optó por una solución alternativa en la que no haría falta hacer uso de esto, sino que dividimos

el modelo en trozos y a cada uno se le aplicamos una textura.

En conclusión, el proyecto se ha llevado a cabo sin ningún problema mayor y se ha conseguido todos los objetivos que se pusieron. Se implementó la generación del mapa y se diseñó como debería de funcionar el algoritmo de generación de estructuras, se plantearon diferentes soluciones para el algoritmo de generación de dificultad y se desarrolló la que se consideró mejor y se desarrolló en la fase de diseño una plantilla para la construcción de personajes, objetos y habilidades.

5.2. Trabajo a futuro

En esta sección vamos a comentar algunas de las posibles adiciones que se podrían añadir en el proyecto. Debido al tamaño del proyecto, no nos vamos a detener en nombrar todo el trabajo, si no que vamos a nombrar las adiciones más importantes al proyecto.

- Las dos adiciones al proyecto que podrían ser más relevantes serían añadir las partes de este proyecto que fueron planteadas para ser dejadas en la fase de diseño. Estas son la implementación de sistema de personajes, objetos y habilidades y el algoritmo de creación de estructuras.

- Ruido de Perlin 3D. Hasta ahora en el proyecto hemos estado hablando de mapas bidimensionales de ruido generados con el ruido de Perlin, con el ruido de Perlin 3D podemos obtener mapas tridimensionales. Esto nos permite obtener terrenos más complejos que permitirían la aparición de estructuras geológicas más complejas como cuevas, acantilados. Esto significa la utilización de un nuevo algoritmo para crear el modelo del terreno, un ejemplo es el algoritmo *Marching Cubes* [1]. La razón por lo que no se ha implementado es que podría vulnerar el requisito de que todo el mapa debe de ser accesible por el jugador. Este algoritmo puede generar estructuras que son inaccesibles para el jugador como son las islas en el aire. Como trabajo futuro se investigará cómo implementar esto para generar estructuras geológicas complejas sin vulnerar ese requisito.

- Durante la implementación de los agentes se encontró un problema con la implementación de la erosión por viento, y es que las partículas de viento necesitan de sedimentos para erosionar el terreno. En la solución propuesta se añadió a posteriori una capa de sedimentos sobre el terreno que corresponde a un porcentaje del terreno según el bioma. En futuras versiones se puede añadir el mapa de alturas con diversas capas con propiedades distintas, simulando así las diferentes capas de tierra de diferentes materiales que pueden existir en el terreno, así como una capa de sedimentos.

[1] https://es.wikipedia.org/wiki/Cubos_de_marcha

- Desarrollo de un algoritmo de instanciación de objetos. En el estado actual, todos los objetos se mantienen en memoria y se muestran según la posición del jugador. En futuras versiones, sería interesante desarrollar un algoritmo que no mantenga en memoria todos los objetos.

- A partir del algoritmo de dificultad desarrollado y una vez implementado diferentes enemigos, se debería implementar una algoritmo que coloque los enemigos por el mapa. Un ejemplo que fue pensado es que se puede asignar a cada uno de los enemigos puntos de amenaza y que cada celda, dependiendo de su dificultad, tenga una serie de puntos de amenaza para gastar, y que en base a estos existan unos enemigos u otros.

5.3. Tiempo estimado empleado

En esta sección se presenta una estimación de los tiempos en horas empleado para la realización de este TFG. En la Figura 5.1, se puede observar el tiempo estimado para la realización de las diferentes fases de desarrollo. Las fases son Estudio previo, Análisis, Estudio y diseño, Implementación y Documentacion en este orden. La fase de Estudio previo comprende el desarrollo de la propuesta y las discursiones iniciales para acotar el proyecto, se tratan de la Sección 1.2 del Capítulo 1. La fase de Análisis se trata del desarrollo de los requisitos del sistema y el diseño de la estructura del proyecto, corresponde al Capítulo 2. La fase Estudio y diseño corresponde a la realización del Capítulo 3, se plantearon diversas alternativas para cumplir con los objetivos y se eligieron las mejores. La fase de Implementación corresponde al Capítulo 4, en el cual se lleva a cabo la implementación de las diferentes partes del prototipo y, finalmente, se une. La fase Documentación corresponde con la realización de la memoria. En la Tabla 5.1 se muestran las horas empleadas para cada fase.

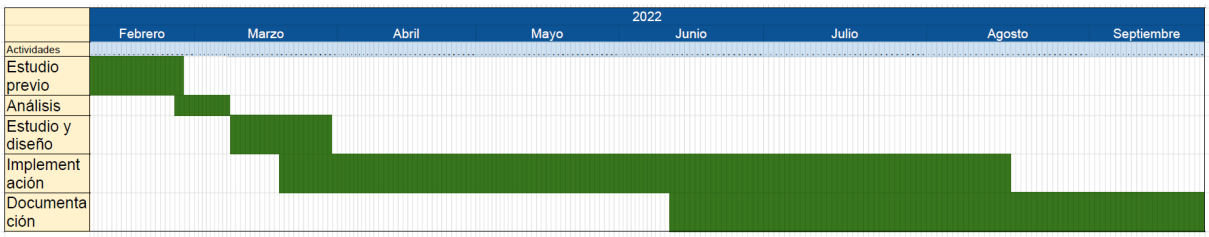


Figura 5.1: Diagrama de Gantt con el tiempo estimado para cada fase

Tarea	Horas
Estudio previo y análisis	55
Estudio y diseño	45
Implementación y evaluación	190
Documentación	100

Cuadro 5.1: Tabla de horas dedicadas a cada fase

En total las horas dedicadas al TFG han sido de **395**.

5.4. Valoración personal

La elaboración de este proyecto ha sido, junto a la asignatura de Videojuegos, mi primera experiencia real con el desarrollo de videojuegos. En lo personal esto me ha permitido aprender mucho acerca de Unity y del desarrollo de videojuegos en un entorno más próximo a la realidad. Además esto me ha permitido aplicar muchos de los conocimientos obtenidos a lo largo de la carrera, más en concreto, los conocimientos de Ingeniería del Software (por análisis, estudio y diseño del proyecto) y de Proyecto Software (por la planificación de un proyecto). El hecho de que este Trabajo de Fin de Grado se trata de un proyecto personal me ha hecho ver lo duro que puede ser desarrollar videojuegos. Además, me ha hecho ver la importancia de este tipo de proyectos para obtener una perspectiva sobre lo que es el desarrollo de un sistema que ha sido pensado e implementado por ti. El realizar un TFG como proyecto personal en mi opinión es algo muy duro, pero muy reconfortante, una idea que fue diseñada en tu cabeza ha visto la luz y se ha implementado. Yo creo que es algo que debería de hacer más gente sobre todo si se quieren dedicar a la industria de los videojuegos porque te da una primera visión de lo que es el desarrollo de un Indie. Agradecer aquí también a Carlos Bobed y Eduardo Mena por creer en el proyecto, no todo el mundo aceptaría el desarrollo de algo que puede ser muy complicado.

Bibliografía

[1] *Algorithm for game difficulty/pacing curve generation in single player RPG*. 2014. URL: https://gamedev.stackexchange.com/questions/67875/algorithm-for-game-difficulty-pacing-curve-generation-in-single-player-rpg.

[2] Johan Classon y Viktor Andersson. «Procedural Generation of Levels with Controllable Difficulty for a Platform Game Using a Genetic Algorithm». En: 2016.

[3] *Does elevation affect temperature?* 2021. URL: https://www.onthesnow.com/news/does-elevation-affect-temperature/.

[4] Red Blob Games. *Making maps with noise functions*. Red Blob Games, 2015. URL: https://www.redblobgames.com/maps/terrain-from-noise/.

[5] *How to generate trees or other structures over chunks in a 2D Minecraft-like game*. 2021. URL: https://stackoverflow.com/questions/67557147/how-to-generate-trees-or-other-structures-over-chunks-in-a-2d-minecraft-like-gam.

[6] *How to randomly generate biome with perlin noise?* 2020. URL: https://gamedev.stackexchange.com/questions/186194/how-to-randomly-generate-biome-with-perlin-noise.

[7] KdotJPG. *Procedural island generation noise generation*. 2021. URL: https://old.reddit.com/r/proceduralgeneration/comments/kaen7h/new_video_on_procedural_island_noise_generation/gfjmgen/.

[8] George Kelly y Hugh McCabe. «A Survey of Procedural Techniques for City Generation». En: 14 (ene. de 2006), págs. 1-29. DOI: 10.21427/D76M9P.

[9] Klayton Kowalski. *Game Development Tutorial Diamond Square and Procedural Map Generation*. URL: https://www.youtube.com/watch?v=4GuAV1PnurU.

[10] Sebastian Lague. *Procedural Landmass Generation (E05: Mesh)*. URL: https://www.youtube.com/watch?v=4RpVBYW1r5M.

[11] *Minecraft style world generation*. 2021. URL: https://gamedev.stackexchange.com/questions/148826/minecraft-style-world-generation.

[12] *On Procedural Generation*. 2015. URL: https://hero.handmade.network/forums/game-discussion/t/182-on_procedural_generation.

[13] *Poisson Disk Sampling*. 2009. URL: http://devmag.org.za/2009/05/03/poisson-disk-sampling/.

[14] *progression and difficulty curve in an open world roguelike*. 2019. URL: https://www.reddit.com/r/roguelikedev/comments/b2zveh/progression_and_difficulty_curve_in_an_open_world/.

[15] ranmantaru. «Water erosion on heightmap terrain». En: *ranmantaru* (2011). URL: http://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/.

[16] Samuel Rufat y Hovig Minassian. «Video games and urban simulation: New tools or new tricks?» En: *Cybergeo* 2012 (jul. de 2012). DOI: 10.4000/cybergeo.25561.

[17] SebLague. *Hydraulic-Erosion*. 2019. URL: https://github.com/SebLague/Hydraulic-Erosion/blob/Coding-Adventure-E01/Assets/Scripts/Erosion.cs.

[18] *Simple Particle-Based Wind Erosion*. URL: https://nickmcd.me/2020/11/23/particle-based-wind-erosion/.

[19] Job Talle. *Cubic Noise*. Job Talle, 2017. URL: https://jobtalle.com/cubic_noise.html.

[20] Enis Turgut y Öznur Usanmaz. «An Analysis of Altitude Wind and Humidity based on Long-term Radiosonde Data». En: *ANADOLU UNIVERSITY JOURNAL OF SCIENCE AND TECHNOLOGY A - Applied Sciences and Engineering* 17 (dic. de 2016), págs. 830-830. DOI: 10.18038/aubtda.279852.

[21] u/watawatabou. *Fantasy medieval cities for the monthly challenge*. URL: https://www.reddit.com/r/proceduralgeneration/comments/668sqb/fantasy_medieval_cities_for_the_monthly_challenge/.

[22] *What is the ideal difficulty curve for procedural levels?* 2020. URL: https://www.reddit.com/r/gamedesign/comments/jt35vc/what_is_the_ideal_difficulty_curve_for_procedural/.

[23] Wikipedia. *Diamond-square algorithm*. URL: https://en.wikipedia.org/wiki/Diamond-square_algorithm.

[24] Wikipedia. *Simple Noise*. URL: https://en.wikipedia.org/wiki/Simplex_noise.

[25] Wikipedia. *Worley Noise*. URL: https://en.wikipedia.org/wiki/Worley_noise.

Apéndice A

Análisis del proyecto

A.1. Requisitos funcionales y no funcionales

A.1.1. Requisitos funcionales

Los requisitos funcionales se dividen en 4 partes:

1. **Partida**

Número	Descripción
RF-1	La partida comienza con la aparición del personaje al inicio.
RF-2	La partida termina al derrotar al jefe final.
RF-3	El jefe final se encontrará en un punto del mapa que llamaremos punto final.
RF-4	Los puntos final e inicial no serán siempre iguales y dependerán de la partida.
RF-5	El jugador debe moverse por todo el mapa sin obstáculos.
RF-6	Al inicio de la partida el jugador desconoce el mapa, solamente lo descubrirá si explora.
RF-7	El jugador debe poder repetir la misma partida siempre que quiera.
RF-8	El terreno debe de estar iluminado correctamente.
RF-9	Se le permite al jugador poner el juego en pausa.
RF-10	El jugador puede modificar las configuraciones de juego.
RF-11	El jugador debe de poder abandonar la partida en todo momento.
RF-12	No se podrá guardar el proceso de una partida.

Cuadro A.1: Requisitos funcionales de una partida

2. **Generación de mapa**

Número	Descripción
RF-13	El mapa se generará aleatoriamente.
RF-14	Pueden repetirse mapas.
RF-15	El mapa estará limitado por agua.
RF-16	El mapa tendrá diferentes alturas.
RF-17	El mapa deberá de contener terreno por debajo del mar y montañas lo suficientemente altas para ser montañas nevadas pero siempre siendo accesibles para el jugador.
RF-18	El mapa contendrá diferentes biomas.
RF-19	Los biomas depende del nivel de altura, temperatura y precipitaciones.
RF-20	Los biomas se aplicarán sobre el mapa formando zonas.
RF-21	Sobre cada una de las zonas se generarán elementos caracterásticos de los biomas.
RF-22	Los elementos que dependen de un bioma son la vegetación y otros elementos del terreno como por ejemplo rocas.
RF-23	Todo el mapa debe ser accesible en todo el momento por el jugador, no pueden generarse regiones no accesibles para el jugador con las habilidades ya definidas.
RF-24	Se generan estructuras sobre el terreno, estas estructuras se pueden generar váa programación.
RF-25	Los edificios no son accesibles, funcionarán como decoración.
RF-26	El terreno puede disponer de contenedores que proveerán de objetos.

Cuadro A.2: Requisitos funcionales de generación de mapa

3. **Generación de dificultad**

Número	Descripción
RF-27	El mapa se encuentra dividido en celdas y cada una tiene asociada una dificultad y estas deben de seguir un orden de dificultad. Por ejemplo se generarán dificultades Fácil, Medio y Difácil y este ese orden. Un conjunto de celdas creará nucleos de dificultad similares a los pisos en un roguelike.
RF-28	Estas dificultades pueden dividirse en sub dificultades que determinen el número de enemigos que pueden generarse.
RF-29	Cada dificultad podrá tener adyacente una dificultad concreta.
RF-30	Existen dos puntos especiales, los puntos de inicio y final. Estos puntos no pueden generar ningún enemigo.
RF-31	Los puntos inicial y final corresponden a una celda del mapa.
RF-32	En el punto inicial no puede existir ningún personaje más allá del jugador.
RF-33	El punto final solamente puede contener el jefe final.
RF-34	El jefe final será el mismo en cada partida, no se generará aleatoriamente.
RF-35	La asignación de dificultad sobre cada celda debe de cumplir ciertas condiciones para que sea justo para el jugador.
RF-36	El punto inicial solo puede estar rodeado por celdas de dificultad fácil.
RF-37	El punto final solo puede estar rodeado por celdas de dificultad difácil.
RF-38	Las celdas con dificultad fácil solo pueden tener celdas adyacentes con dificultades fácil o medio o ser la celda de inicio.
RF-39	Las celdas con dificultad medio solo pueden tener celdas adyacentes con dificultades fácil, medio o difácil.
RF-40	Las celdas con dificultad difácil solo pueden tener celdas adyacentes con dificultades medio, difácil o ser la celda de final.
RF-41	La celda inicial es una celda aleatoria que cumpla una distancia mánima con respecto a la celda final.
RF-42	La distancia mánima debe garantizar que el jugador pase por todas las dificultades.
RF-43	La dificultad determina la cantidad y tipo de enemigo.

Cuadro A.3: Requisitos funcionales de generación de dificultad

4. Construcción de personajes, habilidades y objetos

Número	Descripción
RF-44	Existen dos tipos de personajes según su actividad, personajes activos y no activos (o pasivos).
RF-45	Los personajes activos son los personajes que tienen un comportamiento.
RF-46	Los personajes pasivos no tienen un comportamiento, pero pueden interactuar con el jugador.
RF-47	Existirán dos tipos de personajes según puedan o no ser controlados por el jugador. El primero será el héroe y el segundo su acompañante, y los otros serán los NPCs.
RF-48	El héroe será controlado por el jugador.
RF-49	Cada personaje activo tendrá características de combate: Vida, ataque físico y mágico, defensa física y mágica, Velocidad de movimiento, Velocidad de ataque, estamina y maná.
RF-50	Cada personaje tendrá habilidades y objetos equipados limitados por un peso máximo.
RF-51	Los jefes tienen un comportamiento más complejo.
RF-52	Los personajes activos podrán realizar acciones de combate, como por ejemplo: correr, rodar, atacar, defenderse de un ataque, lanzar proyectiles, ejecutar habilidades, etc.
RF-53	Los personajes activos pueden sufrir efectos alterados.
RF-54	Todos los estados tienen una duración. Pueden acabarse utilizando un objeto o habilidad.

Cuadro A.4: Requisitos funcionales de construcción de personajes

Número	Descripción
RF-55	Los objetos se dividen en tres grupos: activos, pasivos y mixtos.
RF-56	Los objetos activos otorgan habilidades al jugador.
RF-57	Los objetos pasivos mejoran o empeoran las estadísticas. También pueden producir efectos.
RF-58	Los objetos pasivos son de dos tipos: consumibles o equipables.
RF-59	Los consumibles tendrán efecto inmediato.
RF-60	Los equipables podrán dar habilidades y/o subir las estadísticas mientras el personaje los lleve consigo.
RF-61	Los objetos pueden aparecer al derrotar a un enemigo, en un barril o cofre.
RF-62	Las armas son objetos equipables.
RF-63	Las armas aplican daño físico y/o mágico.
RF-64	Cada personaje puede llevar una única arma.
RF-65	Las armas pueden realizar acciones de ataque.
RF-66	Las armas pueden tener habilidades pasivas.

Cuadro A.5: Requisitos funcionales de construcción de objetos

Número	Descripción
RF-67	Las habilidades pueden producir estados alterados sobre quien la recibe.
RF-68	Existen dos tipos de habilidades: activas, pasivas y mixtas.
RF-69	Las habilidades activas son conjuntos de acciones y efectos y subidas de estadística.
RF-70	Las habilidades activas pueden consumir maná y estamina.
RF-71	Las habilidades activas tienen una duración y al terminarse necesitan un tiempo de refresco para poder volverse a usar.
RF-72	Las habilidades pasivas proporcionan efectos y subidas de estadística de manera pasiva (no es necesario activarlas y duran infinitamente).
RF-73	Las magias son habilidades activas.
RF-74	Las magias son aplicadas por proyectiles, ataques con armas o habilidades.
RF-75	Las magias consumen maná al activarse.
RF-76	Los ataques realizados por las magias o armas tienen un rango de acción.
RF-77	Las magias aplican daño mágico o aplicar subidas de estadísticas o estados alterados.

Cuadro A.6: Requisitos funcionales de construcción de habilidades

A.1.2. Requisitos no funcionales

Número	Descripción
RNF-1	El jugador se moverá utilizando el teclado.
RNF-2	El jugador podrá también configurar el mando para moverse con él.
RNF-3	La plataforma destino es Windows 10.
RNF-4	Con un ordenador de media-baja gama el juego debe de funcionar a al menos 30 fotogramas por segundo.
RF-5	El juego no debe de hacer uso de red.
RNF-6	Las teclas deben de ser reconfigurables.
RNF-7	El nivel del sonido y de la música se puede ajustar.
RNF-8	El brillo debe de poder ser cambiado.
RNF-9	El juego detectará si existe una configuración hecha por el jugador para el juego, si es así, la cargará.
RNF-10	El juego solo guardará información de configuración en el ordenador del jugador.
RNF-11	La resolución puede ser cambiada.
RNF-12	Al iniciar la partida el juego debe de detectar la resolución de la pantalla que esté usando el jugador, si nunca se ha iniciado el juego antes.
RNF-13	Se utilizará una estética 3D Low poly mezclado con elementos en 2D para la interfaz gráfica.
RNF-14	El juego se desarrollará con el motor Unity y con cdigo C#.
RNF-15	El jugador puede repetir siempre la misma partida introduciendo una semilla.
RNF-16	La cámara se colocará en posición isométrica.
RNF-17	La cámara seguirá al jugador en todo el momento.

Cuadro A.7: Requisitos no funcionales

Apéndice B

Diseño del sistema

B.1. Diagrama de diseño

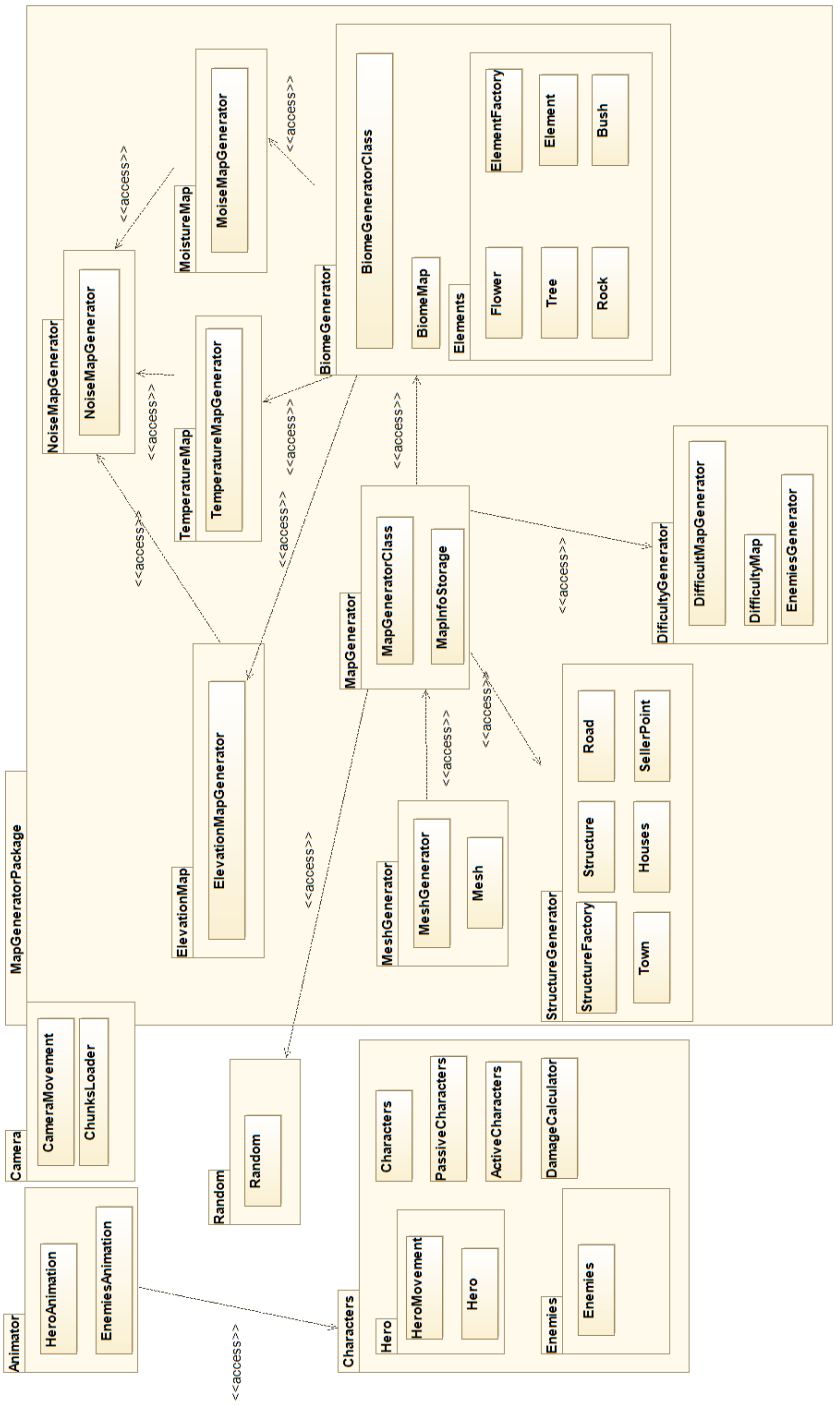


Figura B.1: Diagrama de paquetes de diseño

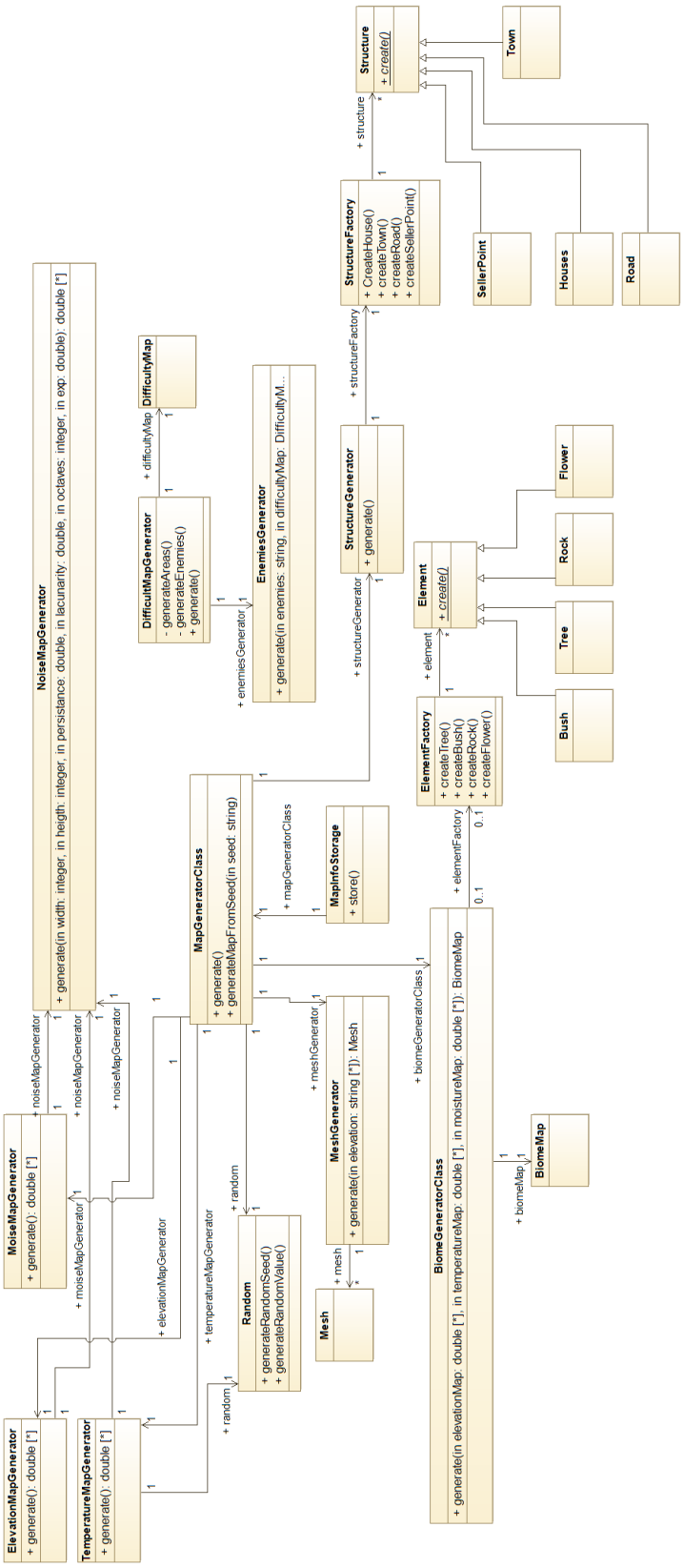


Figura B.2: Diagrama de clases de diseño

Apéndice C

Manuales de usuario

En este capítulo se busca proporcionar al usuario información más extensa sobre el contenido. Exploramos las clases que existen en el prototipo y entraremos más en detalle en el funcionamiento de las diferentes partes del prototipo. Vamos a dividir este capítulo según su función: Clases principales y de apoyo, Generación de mapas, Generación de biomas, Agentes, Generador de dificultad, SaveAndLoad y Otros.

Las clases de Otros se tratan de las clases que implementan algoritmos ya existentes y que se encuentran basados en implementaciones ya existentes, y por esta razón y por espacio se excluyen de este apartado. Nos centraremos en los algoritmos desarrollados por mi.

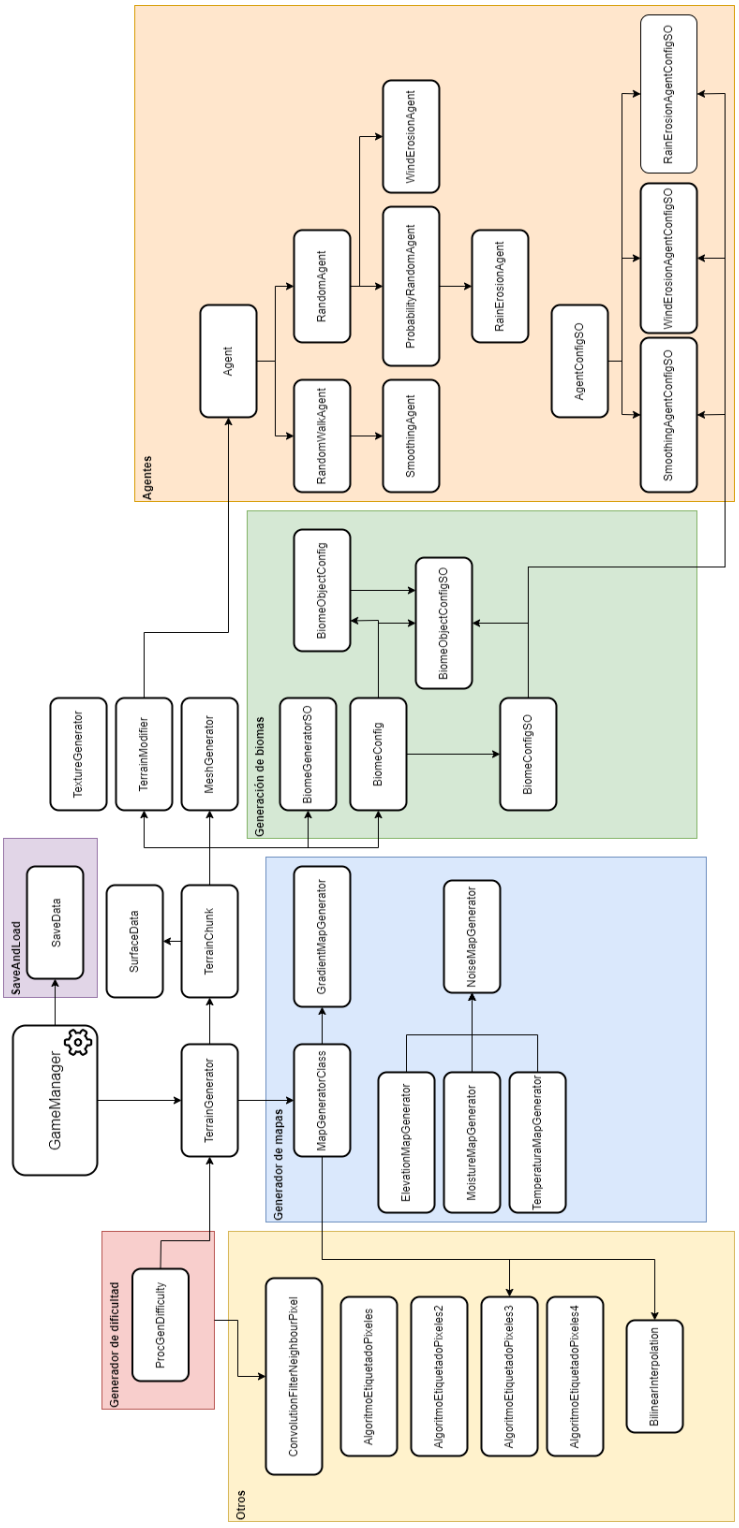


Figura C.1: Estructura de clases de prototipo

C.1. Clases principales

Se trata del conjunto principal de las clases que recogen todas las funcionalidades implementadas en el prototipo. Estas clases son las clases GameManager, TerrainGenerator y TerrainChunk.

GameManager Clase principal del prototipo. Al comienzo del juego, se encarga de invocar a TerrainGenerator, que se encarga de generar el terreno. Luego, crea la lista de trozos de terreno (tipo de dato TerrainChunk). En cada uno de los frames se comprueba la posición de la entidad Jugador estos se utilizará para cargar o no algunos chunks. Esta clase se trata de una clase especial de Unity que se ejecutará al comienzo de la ejecución, además esta clase se hereda de la clase especial de Unity MonoBehaviour, que implementa las funciones Start() y Update() que serán llamadas al principio de la instanciación de la clase y en cada frame respectivamente.

TerrainGenerator Función encargada de la generación del Terreno. Contiene toda la funcionalidad relacionada con esta, además de (de manera temporal) la generación de dificultad. Esta clase también es la encargada de la inicialización de la semilla de la clase Random. A continuación explicaremos cada una de las funciones de esta clase.

- InitTerrain(): Inicializa el objeto base del Terreno.
- GenerateTerrain(): Inicializa la semilla aleatoria, genera se encarga de generar los mapas, dividir el mapa en trozos de tierra, llama a las funciones para crear los biomas y poblarlos de objetos, aplica los agentes, genera el mapa de dificultad y aplica las texturas sobre los trozos de terreno.

TerrainChunk Esta clase representa el trozo del terreno. Un problema al que se enfrenta el prototipo es que es imposible mantener en memoria un terreno de grandes dimensiones entero, por lo que es necesario partirlo en trozos (o chunks) que constituirán el terreno entero. Esto permitirá no tener en memoria en todo momento el mapa entero, además de que facilita el paralelismo para la creación de los objetos del mapa. Otra razón para usarlo es que Unity no permite más una malla de triángulos de un tamaño superior a 65025 vértices.

En la propia clase generadora de terreno, TerrainGenerator, tras la creación de los mapas, se dividirá el mapa en trozos de un tamaño máximo de 255x255 (o en otro valor, en el prototipo se ha usado este).

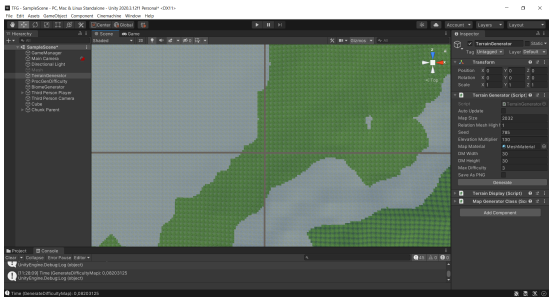
Los trozos se instanciarán en la clase TerrainChunk, que será la encargada de crear la malla de triángulos, crear los biomas y objetos y aplicar modificadores sobre el trozo.

El número de chunks existentes en un mapa no es fijo por lo que se deberá de calcular utilizando la siguiente operación: int numberOfChunks = Mathf.RoundToInt(mapSize / (float)ChunkSize); Nos devuelve el número de chunks en anchura por ejemplo. Por simplificación del prototipo se ha supuesto que el mapa siempre es cuadrado. Como se supone que el número de Chunks va a ser numberOfChunks x numberOfChunks, se iterará por todos el número de chunks y se crearán las clases TerrainChunk, para esto habrá que hacer una copia de los datos de MapData que deben de tener cada chunk. Estas se guardarán temporalmente en una lista en TerrainGenerator, ya que este será el encargado de llamar a las diferentes funciones de TerrainChunk, antes de guardar todos sus datos y borrarse.

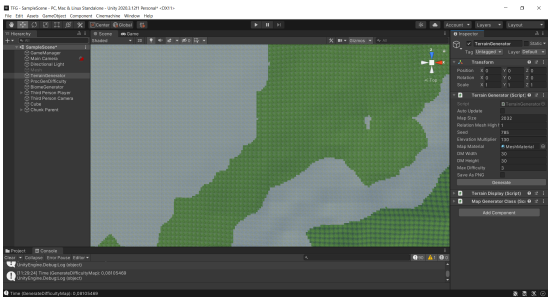
Una alternativa para no tener que copiar sería utilizar los mapas de ruido y sacar cada uno de los trozos ya independientes. Esto se puede conseguir utilizando los offset aleatorios calculados para cada uno de los sumandole los offset de los trozos del mapa (cada chunk se encontraría a 255 de distancia en anchura y 255 en altura), sería necesario también mantener las variables anchura y altura de la función de generación del mapa de ruido (habría que hacer algún cambio a esta), además de que sería necesario ajustar también el gradiente.

Un problema a lo que se puede encontrar es que por cómo se encuentra implementado el mallado se forman separaciones entre las mallas. Por esta razón, se deberá de variar el tamaño de los trozos de acuerdo a su posición. Las reglas que sigue para variar el tamaño de acuerdo a su posición es la siguiente:

- Si el Chunk se encuentra en la posición anchura - 1 y en altura -1 entonces se encuentra en la esquina inferior derecha y se trata del último trozo de terreno. Por lo tanto, no tiene que rellenar ninguna separación.
- Si el Chunk se encuentra en anchura -1 y su altura es menor a altura -1, entonces se formarán huecos entre los chunks de arriba y abajo. Por lo que el chunk de menor altura deberá de incluir los puntos de la primera fila del otro.
- Si el Chunk se encuentra en altura - 1 y su anchura es menor a altura - 1, entonces se formarán huecos en los laterales. Se deberán de tapar de manera similar al anterior.
- Si es menor a anchura -1 y altura -1, deberá tapar huecos en todas las direcciones.



(a) Imagen de mallado sin arreglar las separaciones entre las mallas



(b) Imagen de mallado habiendo arreglado las separaciones de las mallas

Figura C.2: Comparación de mallado arreglando las separaciones de mallas

Esta clase también es la encargada de aplicar las texturas de cada uno de los chunks. Para ello, al invocar el constructor de esta clase existe el booleano useTextures que se utiliza para saber como se tiene que aplicar las texturas sobre el modelo del terreno. Las funciones que se encargan de llevar esto a cargo son: CreateMaterialFromColorMap() y CreateMaterialFromTextures(). La función CreateMaterialFromColorMap() crea una textura a partir de los colores de los biomas, se la aplica un material y aplica este material al modelo del terreno. La función CreateMaterialFromTextures() aplica unos materiales creados previamente con la texturas de los biomas sobre el modelo del terreno.

C.2. Clases de apoyo

SurfaceData Clase relacionada con el TerrainChunk, representa los datos de las características del terreno. Estos datos son la elevación, características del terreno relacionadas con los agentes, listas de biomas que se encuentran en el chunk y que bioma se encuentra en cada punto.

MeshData Clase encargada de la generación de la malla de triangulos para el modelo del terreno. Para la creación de la malla de triángulos se utilizará la clase de Unity Mesh, que dado un conjunto correcto de vértices, triángulos y uv, es capaz de crear una malla de triángulos de forma correcta. Para crear estos conjuntos se ha utilizado un algoritmo, a continuación se explicará suponiendo si se tiene un mapa de 3x3:

- Se necesitan tres conjuntos de elementos para crear la malla de triángulos. Dos esenciales, el conjunto de vértices y de triángulos, y los dos opcionales, uvs, coordenadas de los vértices en la textura (de 0 a 1), o colores, que indica el color de un vértice.

    - El conjunto de vértices se trata de un vector de 3 componentes que representa las coordenadas x, y y z. El tamaño de ese conjunto es igual a la anchura por la altura del mapa.

- El conjunto de triángulos es un vector que contiene los índices de los vértices. Van en tríos, es decir, cada tres índices forman un triángulo, esto quiere decir que el tamaño del vector debe ser múltiplo de tres. El tamaño de ese conjunto es igual a la anchura - 1 por la altura - 1 del mapa por 6 (el número de aristas por cada par de triángulos).

- Tanto el conjunto de uvs y colores tienen el tamaño de la anchura por la altura del mapa.

- Se crea una variable llamada índice del vértice (o vertexIndex) que se utilizará con índice del vértice a la hora de crear los triángulos.

- Se iterará a lo largo de todos los vértices del mapa para añadir los elementos.

  - Se crea un nuevo vértice y se añadirá al índice vertexIndex actual. Cada uno de los vértices tendrá una altura (valor y) que corresponderá con la altura del punto que deberá de ser multiplicada por una constante que representa la elevación máxima de un punto.

  - Se creará un nuevo vector de uv (por ejemplo), y se añadirá al conjunto de uvs con el índice vertexIndex. Por ejemplo, el valor se puede normalizar es decir, dividir el valor de la coordenada x por la anchura, e igual con la coordenada y, aunque esto depende de la textura que se quiere aplicar.

  - Para crear los triángulos hay que asegurarse de que el punto actual sea menor a la anchura -1 y la altura -1.

    - Ejemplo de cómo se añade:

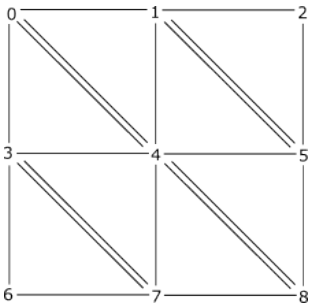


Figura C.3: Imagen funcionamiento del algoritmo de generación de malla de triángulos

    - Para crear el primer triángulo se necesitan los puntos 0,3,4. Es decir, se necesitan los (vertexIndex, vertexIndex + width + 1, vertexIndex + width).

    - Para el segundo triángulo se necesitan los puntos 0,1,4. Es, decir, se nece-

sitan los vertexIndex + width + 1, vertexIndex, vertexIndex + 1).

- Si se sigue ese método se pueden generar todos los triángulos.

- Al final de cada iteración se incrementará el valor de vertexIndex.

- Repetir hasta que se haya pasado por todos los puntos.

- Una vez creados los conjuntos se almacenan en la clase MeshData para que luego con esta se pueda crear la malla.

El algoritmo devolverá una instancia de la clase MeshData para que luego en código se ejecute la función CreateMesh() que crea la malla de triángulos.

Pseudocódigo del algoritmo:

```
MeshData GenerateMesh(float elevation[,], float elevationMultiplier, int
    width, height) {
    Vector3[] vertices= Vector3[width * height];
    int[] triangles = int[(width - 1) * (height - 1) * 6];
    Vector2[] uvs = Vector2[width * height];
    int vertex_index = 0;
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            float real_elevation = elevation[x,y] * elevationMultiplier;
            vertices[vertexIndex] = Vector3 (x, real_elevation, y);
            uvs[vertexIndex] = Vector2(x / width, y / height);
            if (x < width - 1 && y < height - 1) {
                AddTriangle(vertexIndex, vertexIndex + width + 1,
    vertexIndex + width);
                AddTriangle(vertexIndex + width + 1, vertexIndex,
    vertexIndex + 1);
            }
            vertexIndex++;
        }
    }
    return MeshData {vertices, triangle,uvs};
}
```

Existe una variación necesaria para poder aplicar las texturas sobre el modelo del terreno. Esta varianción consiste en dividir el modelo del terreno en modelos de menor tamaño. En este caso, tendrá n instancias de meshData, una para cada bioma, y añadirá a cada una de éstas los vértices que pertenecen a el bioma y los vertices adyacentes (esto es necesario para evitar que se generen huecos entre las mallas). A partir de los vértices de cada una de los meshData se crearán los triángulos. Esta variación se trata de la función GenerateTerrainMultipleMesh().

C.3. Generación de mapas

Conjunto de clases que se encargan de la generación de mapas. Antes de empezar debemos de diferenciar concepto mapa de mundo, cuando hablamos de mapa nos referimos a la generación de matrices bidimensionales de números, mientras que cuando hablamos de mundo nos referimos al entorno tridimensional con el cual el usuario puede interactuar.

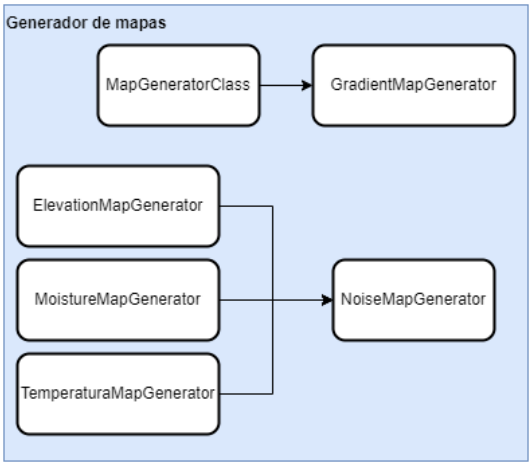


Figura C.4: Estructura generación de mapas

NoiseMapGenerator Clase encargada de la generación de mapas de ruido. La función contiene principalmente dos funciones:

```
1 public static float[,] generate(int width, int height, int octaves,
    float persistance, float lacunarity, Vector2[] offset);
2
3 public static float GenerateNoiseValue(int x, int y, int width, int
    height, int octaves, float persistance, float lacunarity, Vector2[]
    offset);
```

- generate() Esta función se divide en dos partes, en la primera se generan los valores de ruido recorriendo en un bucle todos los puntos del mapa marcados por los parámetros width y height (achura y altura respectivamente), en la segunda parte se normalizan los valores de ruido respecto al máximo y mínimo calculado utilizando InverseLerp[1]. Para generar los valores de ruido creará valores de ruido para cada una de las octavas (definido por el parámetro octaves, determina el número de veces que se va a calcular el valor de ruido para cada punto), estos valores se generan con la función Mathf.PerlinNoise(nx, ny) que necesita dos parámetros nx e ny será el valor del punto normalizado por la anchura y altura multiplicado por la frecuencia

[1]Determina donde un valor se encuentra entre dos puntos, devolviendo valores entre 0 y 1.

y sumado por un offset para añadir más aleatoriedad. La frecuencia la calcularemos utilizando el parámetro lacunarity y en cada iteración deberá de crecer. La amplitud decrece en cada iteración y determina la elevación máxima en un iteración, se calcula con el parámetro persistance.

- GenerateNoiseValue() Genera ruido de manera individual.

GradientMapGenerator Clase que contiene todas las funciones que se encargan de generar mapas que se utilizarán para la creación de la isla. Estas funciones son las mismas explicadas en la sección 4.2.2.

```
/*
 * Función que genera el mapa de un gradiente radial.
 * Parámetros:
 * width, anchura del mapa.
 * height, altura del mapa.
 * center, punto que representa el centro sobre el que se creará la gradiente
 * maxDistance, distancia del centro con el punto más lejano respecto a este
 * inverse, invierte los valores del mapa.
 * exp, eleva cada uno de los valores del mapa por exp.
 */
public static float[,] RadialGradient(int width, int height, Vector2 center, float maxDistance, bool inverse, float exp);

/*
 * Función que genera el mapa de un gradiente cuadrada.
 * Parámetros:
 * width, anchura del mapa.
 * height, altura del mapa.
 * center, punto que representa el centro sobre el que se creará la gradiente
 * inverse, invierte los valores del mapa.
 * exp, eleva cada uno de los valores del mapa por exp.
 */
public static float[,] SquareGradient(int width, int height, Vector2 center, bool inverse)

/*
 * Función que genera el mapa de un gradiente especial que depende del tama o de la isla que se quiere crear. Similar a los anteriores
 * Parámetros:
 * width, anchura del mapa.
 * height, altura del mapa.
 * islandSize, tama o de la isla que se quiere crear.
 * inverse, invierte los valores del mapa.
```

```
31  * exp, eleva cada uno de los valores del mapa por exp.
32  */
33 public static float[,] FalloffMap(int width, int height, float
       islandSize, bool inverse, float exp)
34
35 /*
36  * Crea un mapa de gradiente similar al anterior pero con una progresión
        menos pronunciada de valiores más bajos a más altos.
37  * Parámetros:
38  * width, anchura del mapa.
39  * height, altura del mapa.
40  * exp, eleva cada uno de los valores del mapa por exp.
41  */
42 public static float[,] SquaredBump (int width, int height, float exp)
```

ElevationMapGenerator Clase que encapsula todas las funciones necesarias para la creación de una mapa de elevación. Su función principal es Generate().

```
1 public static float[,] Generate(int width, int height, int octaves,
      float persistance, float lacunarity, float offsetX, float offsetY,
      float[,] gradient)
```

La función Generate() crea un mapa de ruido de Perlin invocando a la función Generate de NoiseMapGenerator con los parámetros width (anchura del mapa), height (altura del mapa), octaves , persistance, lacunarity y offsets generados aleatoriamente sumados por los offsetX e offsetY. A partir del mapa de ruido restará el mapa de gradiente contenido en la variable gradient sobre cada uno de los puntos para así generar la isla. El mapa de elevación tendrá elevaciones comprendidas entre 0 y 1. La clase también contiene otras variaciones de esta función algo más optimizadas como:

```
1 public static float[,] GenerateOptimized(int width, int height, int
      octaves, float persistance, float lacunarity, float offsetX, float
      offsetY, float[,] gradient)
2
3 public static float[,] GenerateParallel(int width, int height, int
      octaves, float persistance, float lacunarity, float offsetX, float
      offsetY, float[,] gradient)
```

- GenerateOptimized() genera el mapa de elevación con cada valor de un punto por separado, por lo que se pierde la normalización de los valores según el valor máximo y mínimo del mapa. Esto significa que genera mapas diferentes a Generate
- GenerateParallel() genera el mismo mapa que Generate pero la resta con el mapa de gradiente se realiza de manera paralela ya que los puntos no dependen unos de otros.

MoistureMapGenerator Clase que encapsula todas las funciones necesarias para la creación de una mapa de humedad. Su función principal es Generate().

```
public static float[,] Generate(int width, int height, int octaves,
    float persistance, float lacunarity, float offsetX, float offsetY ,
    float[,] elevation)
```

La función Generate() genera un mapa de humedad de tamaño width y height (anchura y altura respectivamente). Con los parámetros width (anchura del mapa), height (altura del mapa), octaves , persistance, lacunarity y offsets generados aleatoriamente sumados por los offsetX e offsetY, crea un mapa de ruido de Perlin invocando a la función Generate de NoiseMapGenerator. Luego, itera por cada punto utilizando el mapa de elevación para limitar la humedad máxima de un punto con una altura h, limita la humedad máxima como 1 - la elevación del punto, de tal manera que un punto con elevación 0.3, su mínima humedad será de 0 y su máxima de 0.7. Además multiplica los valores cercanos al agua (se supondrá altura bajas) por 1.4 si el punto tiene una altura menor a 0.1 y a 1.2 si es menor a 0.2. También todo punto con elevación 0 (se supone agua) su humedad será siempre de 1, es decir, máxima. El mapa de humedad tendrá humedades comprendidas entre 0 y 1.

TemperatureMapGenerator Clase que encapsula todas las funciones necesarias para la creación de una mapa de temperature. Su función principal es Generate().

```
public static float[,] Generate(int width, int height ,float[,]
    elevation, float relationPixelPerGradeOfLatitude)
```

La función Generate se divide en dos partes, en la primera establecemos al temperatura de cada punto en base a la posición de la isla, el tamaño de esta y la temperatura de los polos y el ecuador. Para cada punto se realiza una interpolación según su ubicación en grados con los polos y el ecuador, asignando según la cercanía a estos puntos una temperatura. En la segunda parte tenemos en cuenta la elevación del terreno para disminuir la temperatura de este. Se utiliza el mapa de elevación ya que la temperatura disminuye con respecto a la elevación del terreno. Para calcularlo utilizamos la siguiente fórmula:

```
temp = ((minTemperature - maxTemperature) * temp + maxTemperature) -
    elevation * 0.6f * Globals.MAX_HEIGHT / 100;
```

Se supone que cada 100 metros se disminuye . Para hacer este cálculo se necesita traducir los valores del mapa de elevación a metros, en nuestro caso hemos supuesto que la altura máxima es de 4000 metros. En este caso, el mapa de temperatura tendrá temperaturas comprendidas entre -20 y 40 grados centígrados.

MapData Estructura que encapsula los mapas de elevación, humedad y temperatura.

MapGeneratorClass Clase encargada de la generación de los mapas de elevación, humedad y temperatura. Esta clase tiene atributos por defecto para la creación de los mapas de ruido. Estos son: lacunarity (determina la frecuencia con la que el mapa llega a sus valores máximos), persistance (determina la altura máxima de cada punto para las octavas), octaves (número de mapas que se van a generar) y offsetX y offsetY (offset base para cada uno de los mapa). También contiene el atributo público relationPixelPerGradeOfLatitude que determina la relación de un pixel (o punto) con la latitud del planeta en el que suponemos que existe la isla .Son atributos públicos que se pueden modificar por el usuario. Tiene la función Generate(). Pseudo-código de la misma:

```
1  MapData Generate (int RealMapSize) {
2      int mapSize = 255;
3      MapData mapData;
4      float[,] mapaGradiente = GenerarGradiente(mapSize);
5      float[,] LowResMap_elevacion = GenerarMapaElevacion(mapSize,
       mapaGradiente);
6      float[,] LowResMap_humedad = GenerarMapaHumedad(mapSize, elevacion,
       mapaGradiente);
7      float[,] HIResMap_elevation = BilinearInterpolation(
       LowResMap_elevacion, mapSize, RealMapSize);
8      float[,] HIResMap_humedad = BilinearInterpolation(LowResMap_humedad,
       mapSize, RealMapSize);
9      float[,] MapElevacion_degardado = DegradarMapaAltura (
       HIResMap_elevacion, 0f, 0.15f);
10     float[,] Mapa_Temperatura = GenerarMapaTemperatura(mapSize,
       elevacion);
11     mapData.elevacion    = EliminarPeque asIslas(MapElevacion_degardado)
       ;
12     mapData.humedad      = HIResMap_humedad;
13     mapData.temperatura = Mapa_Temperatura;
14     return mapData;
15 }
```

Se van a generar dos tipos de mapas, unos mapas pequeños de tamaño 255x255 (marcados con el prefijo LowResMap_) y un mapa grande de tamaño establecido por el parámetro RealMapSize x RealMapSize (marcados con el prefijo HighResMap_). En la Línea 4 se genera el mapa de gradiente para ello utilizaremos la clase GradientMapGenerator, en nuestro caso utilizaremos la función SquareBump. En la Línea 5-6 se genera el mapa de elevación y de humedad con las clases ElevationMapGenerator y MoistureMapGenerator de tamaño 255x255. En las Líneas 7-8 incrementaremos el tamaño del mapa utilizando la función Apply de la clase BilinearInterpolation (Ver en sección **??**). En la línea 9, al restar el mapa de ruido por el mapa de gradientes es posible que se generen saltos

de altura muy grandes, para solucionar esto se aplica un suavizado en las alturas de los mapas degradando las alturas comprendidas entre el 0 a 0.15 utilizando una función exponencial. En la línea 10 genera el mapa de temperatura de un tamaño RealMapSize x RealMapSize. En la línea 11 se aplica el algoritmo de eliminación de pequeñas islas con la clase AlgoritmoEtiquetacionPixeles3.

C.4. Generador de biomas

La clase principal para la generación de los biomas se trata de BiomeGeneratorSO. El resto de clases relacionadas a los biomas sirven de apoyo guardando información acerca de esto, o en el caso de BiomeConfig representando el propio bioma.

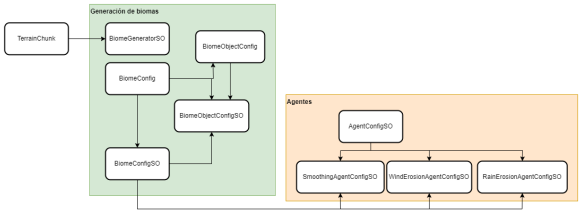


Figura C.5: Estructura de generación de los biomas

BiomeConfigSO Se trata de un scriptable object de Unity (Ver sección E.2 del Apéndice E). Es instanciable desde el editor de Unity y representa las características de un bioma. Las características se ajustan desde el mismo editor y son: Intervalos de elevación, humedad y temperatura donde aparecen (obligatorio valores entre 0 a 1), color y textura asociada al bioma, nombre, objetos que aparecen, la probabilidad de aparición de objetos y las características sobre interactúa cada bioma con los agentes (lista de la clase abstracta AgentConfigSO).

BiomeConfig Clase creada por el BiomeGeneratorSO. Almacena las listas con los puntos que contiene un bioma en concreto, sus características, puntos en los que tiene objetos (a través de una lista de BiomeObjectConfig) y puntos los cuales se encuentran vacíos. Esta clase contiene la función createBiome() que es la encargada de a partir de un bioma vacio de objetos, asignar a puntos libres un objeto según la probabilidad de aparición del objeto y la probabilidad general del bioma de que aparezcan objetos. Finalmente, crea objetos en sus diferentes posiciones.

BiomeGeneratorSO Contiene una lista de tipo BiomeConfigSO que indica los biomas que se pueden crear y cuáles no. En el prototipo es un objeto creado en el editor, y desde este en la variable pública de la lista se añaden las instancias de los Scriptable Objects de BiomeConfigSO. La función Generate() de esta clase es la encargada de asignar cuál es el

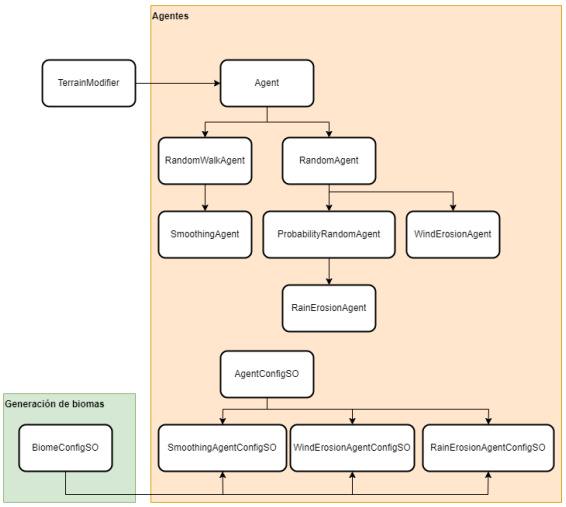


Figura C.6: Estructura agentes

bioma (BiomeConfig) al que pertenece cada uno de los puntos de acuerdo a los intervalos de elevación, humedad y temperatura.

BiomeObjectConfigSO Almacena el objeto a crear, así como sus probabilidades de aparecer. Estas probabilidades se normalizan de acuerdo a las probabilidades de aparición de objetos de un bioma en concreto. Cada bioma (en concreto la clase BiomeConfigSO) contendrá una lista de este tipo de datos para indicar los objetos que genera y la probabilidad de que se genere este objeto en este bioma. Se trata de un Scriptable Object, por lo tanto, se añaden a la lista pública del Scriptable Object de BiomeConfigSO en el editor.

BiomeObjectConfig Clase que almacena la posición, escala, rotación y tipo de un objeto, que está o no instanciado en el mundo. Esta clase se utiliza en BiomeConfig para almacenar los objetos existentes en el bioma.

C.5. Agentes

Dividimos los agentes según su funcionalidad. Las clases con el sufijo ConfigSO contienen los atributos que definen cómo afectan cada uno de estos biomas y se trata de ScriptableObjectsE.2. El resto de clases se tratan de las clases principales que definen el comportamiento de los agentes dada las características de un terreno definidas por el tipos de clases ConfigSO.

TerrainModifier Clase encargada de aplicar los diferentes agentes existentes. Esta clase contiene la función Apply que necesita que se le pase el parámetro por referencia surface de tipo SurfaceData sobre la cual se aplicará los agentes SmoothingAgent, RainErosionAgent,

WindErosionAgent.

Luego de la ejecución de los agentes actualiza los valores de las surfaceData y el TerrainChunk.

Las clases de Agentes se dividen en dos tipos. Las clases Agent, RandomWalkAgent, SmoothingAgent, RandomAgent, ProbabilityRandomAgent, WindErosionAgent y RainErosionAgent son las clases que implementan los agentes. Las clases AgentConfigSO, SmoothingAgentConfigSO, WindErosionAgentConfigSO, RainErosionAgentConfigSO son las clases de tipo Scriptable Objects que recogen los datos sobre cómo afectan los agentes.

Agent Clase abstracta y base de todos los agentes. Contiene las funciones abstractas Start(), Evaluate() y Next(). Cada una de las clases hijas reimplementan alguna de estas funciones o todas.

- Start() Función encargada de inicializar los atributos del agente.
- Evaluate() Función encargada de hacer una ejecución del agente.
- Next() Función que pasa al siguiente punto para ejecutar el agente.

Estas funciones se utilizan en la función pública Execute() que es la encargada de ejecutar todo el agente. Esta función se llama desde el TerrainModifier para cada una de los agentes.

Código de la clase Agent:

```
public abstract class Agent {
    protected virtual void Start(SurfaceData surf) {
        this.surf = surf;
    }
    protected abstract void Evaluate(ref SurfaceData surface);
    protected abstract Vector2 Next();
    public virtual void Execute(ref SurfaceData surf, int iterations) {
        Start(surf);

        int i = 0;
        while (i < iterations)
        {
            Evaluate(ref surf);
            Next();
            i++;
        }
    }
}
```

RandomWalkAgent Clase abstracta que implementa en la función Next() el algoritmo de RandomWalk. Hereda de la clase Agent.

SmoothingAgent Clase que implementa el agente suavizador. Esta clase hereda del algoritmo de RandomWalkAgent.

RandomAgent Clase abstracta que implementa la función Next(). Esta elige una posición aleatoria del terreno cada vez que se implementa.

ProbabilityRandomAgent Agente que hereda de RandomAgent. Dada una serie de biomas, elige una nuevo punto aleatorio de bioma en base a la probabilidad de elegir un bioma.

RainErosionAgent Clase hereda de ProbabilityRandomAgent. Necesita elegir un punto en base a las probabilidades de los biomas, ya que el punto elegido es el punto donde simularemos la gota de lluvía y queremos que ésta tienda aparezca en biomas más húmedos. Se trata de la clase que implementa el agente de la lluvia. Cada gota se mueve en dirección al punto de menor altura más cercano y termina al llegar a él o al evaporarse. Una gota de lluvia erosiona el terreno (reduce la altura de los puntos por los que cae) y se lleva sedimentos que va dejando por su camino. Las gotas se generan teniendo en cuenta la humedad del bioma donde cae y los efectos que puede provocar están determinados por las propiedades del punto donde se encuentra.

WindErosionAgent Clase que hereda de RandomAgente. Utiliza esta clase para generar aleatoria particulas de viento que recorrerán todo el terreno.

AgentConfigSO Clase abstracta que representa el contenedor de características de un agente.

SmoothingAgentConfigSO Clase que contiene el atributo que indica el número de puntos alrededor de un punto que afectarán en el suavizado de altura.

WindErosionAgentConfigSO Clase que contiene los parámetros de suspensión de sedimentos en el aire, abrasión del viento sobre el terreno, la dureza del sedimento y el ratio de asentamiento.

RainErosionAgentConfigSO Clase que contiene la velocidad de erosión de la lluvia, de deposito de sedimentos y velocidad de evaporación.

C.6. SaveAndLoad

Esta sección está pensada para la carga y descarga de objetos y el terreno en ejecución. Este en el prototipo se ha implementado, no obstante, por falta de pruebas, no se ha añadido finalmente. Sin embargo en las pruebas desarrolladas se mostró capaz de guardar los elementos en un archivo JSON y la carga a partir de estos. La clase encargada de hacer esto es SaveData y tiene la capacidad de guardar los valores identificador, coordenadas, posición, multiplicador de la elevación y biomas que aparecen. Los elementos dentro de un objeto se guardan debido a que se tratan de objetos serializables.

C.7. Generador de dificultad

La clase generadora de dificultad se trata de la clase ProcGenDifficulty. El funcionamiento de la clase generadora de dificultad ya ha sido explicada en detalle con lo que en esta sección nos limitaremos a comentar las funciones que existen y cómo funcionan sin entrar en detalle. Esta clase se compone de dos funciones:

- Generate(int width, int height, List¡Vector2Int¿validPos, int MaxDifficultyLevel): Es la función encargada de generar el mapa de dificultad. Esta función crea los clústeres de dificultad y le asigna a cada clúster un nivel de dificultad. Los niveles de dificultad que aparecen dependen de la variable MaxDifficultyLevel. Utilizamos esta y una constante para determinar aleatoriamente el número de de clústeres que existirán. A cada uno de los niveles de dificultad se le asignará una variable aleatoria, ésta se utilizará para determinar cuántos clústeres existen para un nivel de dificultad. Cuando los clústeres ya están creados se les asigna una intensidad. Ejecuta la función SpawnIndividualArea(), que calcula el valor de la intensidad de un clúster sobre cada punto teniendo en cuenta que este valor se reduce con respecto a la distancia. Finalmente, elige cuales son los clústeres a los que pertenece una celda de acuerdo al valor de intensidad, y aplica el filtro de convolución para reducir el valor de dificultad.

- SpawnIndividualArea(): Asigna a cada punto del mapa el valor de la intensidad de un cluster teniendo en cuenta que se reduce con la distancia.

Apéndice D

Glosario de terminos

- Roguelike. Subgénero de los videojuegos de rol que se caracterizan por una aventura a través de laberintos, a través de niveles generados por procedimientos al azar.
- Rol Playing Game o RPG. Es un género de videojuegos donde el jugador controla las acciones de un personaje (o de diversos miembros de un grupo) inmerso en algún detallado mundo. Definición de Wikipedia: `https://es.wikipedia.org/wiki/Videojuego_de_rol`
- Mundo abierto. Un videojuego de mundo abierto es aquel que ofrece al jugador la posibilidad de moverse libremente por un mundo virtual y alterar cualquier elemento a su voluntad.
- Bioma. También llamado paisaje bioclimático o área biótica es una determinada parte del planeta que comparte el clima, flora y fauna. Definición de Wikipedia: `https://es.wikipedia.org/wiki/Bioma`

Apéndice E

Información adicional

E.1. ¿Qué es el ruido de perlin?

El ruido de perlin es una función matemática inventada por Ken Perlin para la generación de texturas de la película Tron (1982). Para obtener los valores utiliza interpolación entre un gran número de gradientes precalculados de vectores que construyen un valor que varía pseudo-aleatoriamente en el espacio o tiempo. No es capaz de generar valores aleatorios de manera infinita, tiene su limitaciones.

Existe una variante para entornos tridimensionales 3D Perlin noise que nos permite obtener texturas tridimensionales, para conseguir esto se puede obtener combinando varias veces la función de perlin noise en un punto en un espacio 3d e interpolando entre los valores perlin_noise(x,y), perlin_noise(x,z) y perlin_noise(y,z) y sus reversas.

Código de ejemplo utilizando Unity:

```
public static float PerlinNoise3D(float nx, float ny, float nz) {
    float xy = Mathf.PerlinNoise(nx, ny);
    float xz = Mathf.PerlinNoise(nx, nz);
    float yz = Mathf.PerlinNoise(ny, nz);

    // reverse.
    float yx = Mathf.PerlinNoise(ny, nx);
    float zx = Mathf.PerlinNoise(nz, nx);
    float zy = Mathf.PerlinNoise(nz, ny);

    return (xy + xz + yz + yx + zx + zy) / 6f;
}
```

Es una de las funciones más utilizadas para generar ruido y sobre todo para la generación

de entornos aleatorios. Es utilizada por ejemplo en Minecraft.

Fuente: Wikipedia. https://es.wikipedia.org/wiki/Ruido_Perlin

E.2. Scriptable Object

Se trata de una Clase definida en el UnityEngine. Un Scriptable Object es un contenedor de datos que se puede usar para guardar gran cantidad de datos, independiente de las instancias de clases. Uno de los principales casos para Scriptable Object es para reducir el uso de memoria del un proyecto eviatndo copias de valores.

Los principales casos para usar un Scriptable Object son:

- Guardar y almacenar datos durante la sesión de Editor.
- Guardar datos como Asset en un proyecto para usar en tiempo de ejecución.

Fuente: Documentación Unity3D `https://docs.unity3d.com/2019.4/Documentation/Manual/class-ScriptableObject.html`

E.3. Inicialización de la clase Random con una semilla

Unity contiene la clase *Random*, que se encarga de generar los numeros pseudo-aleatorios. A esta clase se puede inicializar con una semilla. A partir de esta semilla generará en la misma ejecución los mismos números aleatorios. Podemos llevar esta inicialización a cabo con la siguiente instruccion: *public static void* ***InitState****(int seed);*

En la implementación actual solo se admiten enteros como semilla ya que la clase *Random* de Unity solo deja establecer como semillas números enteros. Una manera de solucionar esto es utilizar la función ***GetHashCode()*** de la clase string que nos devolverá diferentes valores para diferentes entradas. Ejemplo:

```
1     string seed =     Assd28662    ;
2     int iseed = seed.GetHashCode();
3     Random.InitState(seed);
```

Apéndice F

Modelos 3D

F.1. Modelo temporal de jugador

Figura F.1: Modelo temporal del jugador

F.2. Modelo árbol

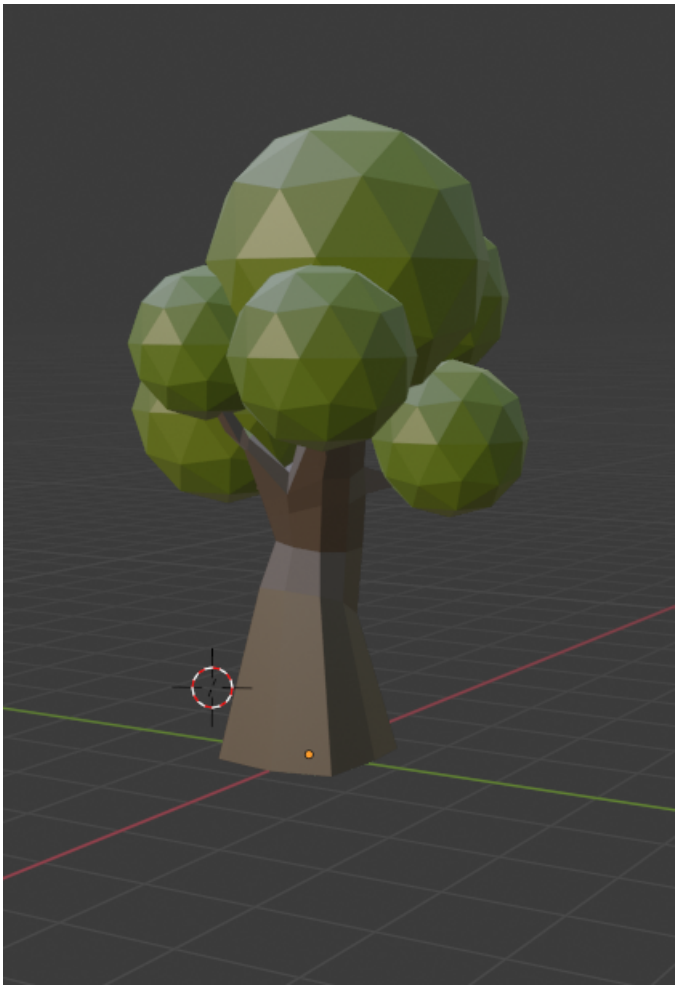


Figura F.2: Modelo 3D de un árbol

F.3. Modelo hierba

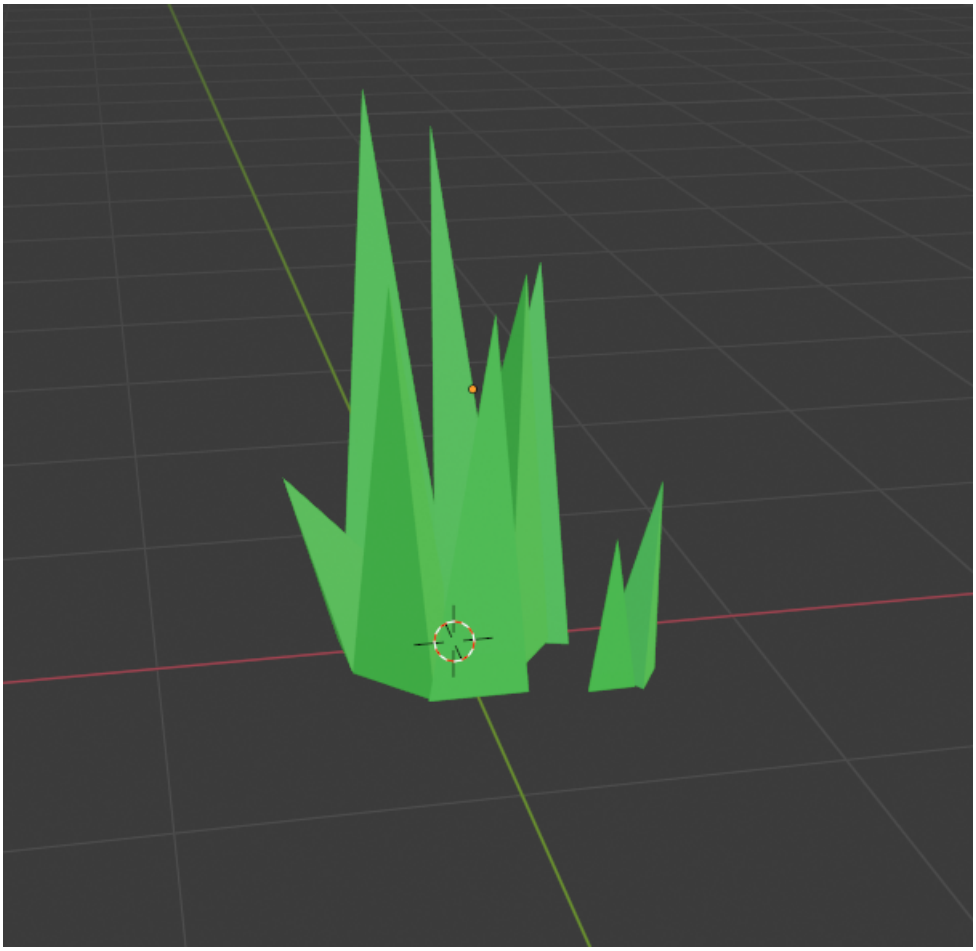


Figura F.3: Modelo 3D de hierba

F.4. Modelo roca

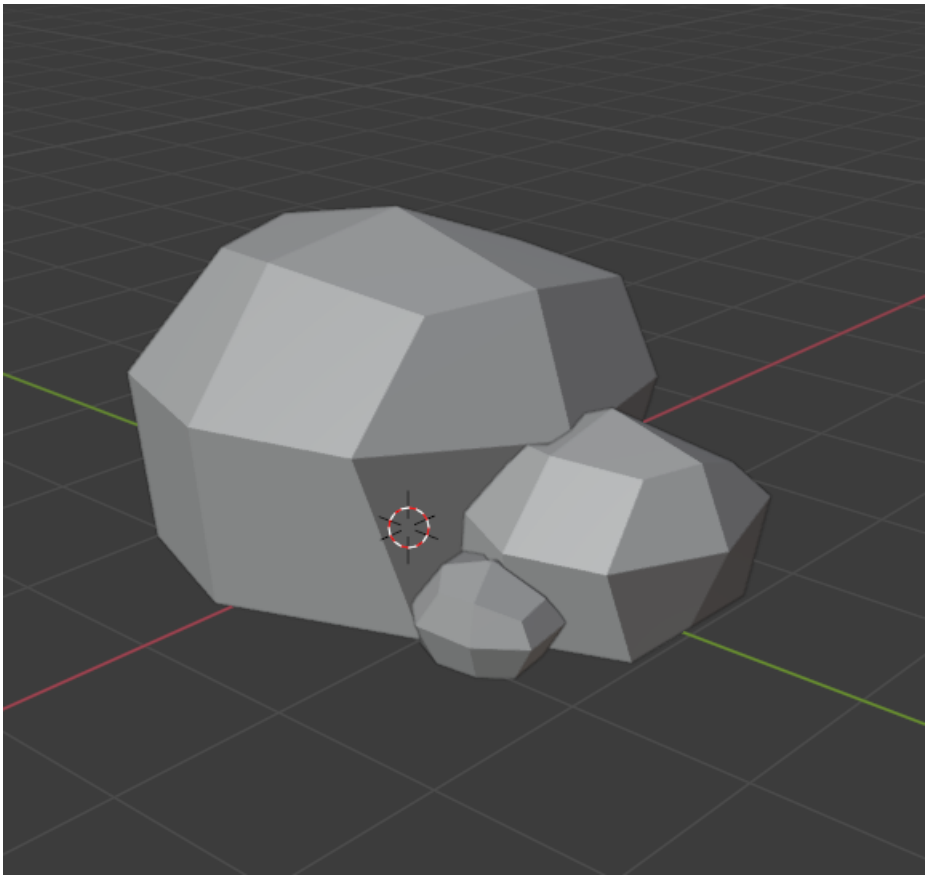


Figura F.4: Modelo 3D de una roca