Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Ingeniería Informática

Proyecto Final

Marching Cubes en Unity 3D

Autor del proyecto: Javier Garzo Pérez

Director del proyecto: Jorge Echeverria Ochoa

Zaragoza, 11 de septiembre de 2020

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma

Fecha

11 de septiembre de 2020

Dedicatoria y Agradecimiento

En primer lugar, me gustaría agradecer a mi familia, que me han apoyado todos estos años y me han brindado una educación y estudios de calidad a lo largo de mi vida. Gracias a vosotros he conseguido saber el valor del esfuerzo, que me ha permitido llegar hasta aquí.

A Jorge Echeverria por confiar en mí, aceptando la propuesta del proyecto. Por su interés y ayuda para que este proyecto alcanzara su calidad final.

A la Universidad San Jorge y a sus profesores, así como a todos los amigos que allí he hecho. Con vosotros he compartido unos de los mejores años de mi vida, que me han permitido desarrollarme como persona.

Gracias a todos por hacer esto posible.

Tabla de contenido

Tabla de figuras

Tabla de tablas

Resumen

El algoritmo de Marching Cubes fue presentado originalmente por William E. Lorensen y Harvet E. Cline en 1987 para la representación 3D de datos médicos. Este algoritmo era usado para la reconstrucción 3D de grupos de capas 2D generadas por escáneres médicos, aunque en la actualidad el algoritmo se ha abierto un pequeño nicho en el campo del desarrollo de videojuegos permitiendo generar terrenos editables en tiempo real de forma rápida y eficiente. Sin embargo, las opciones para implementarlo siguen siendo difíciles debido a que los motores que permiten implementarlo con facilidad son privados o de pago. Buscando democratizar los Marching Cubes para que estén al alcance todo tipo de desarrolladores nace la idea del proyecto.

Los objetivos del proyecto se han centrado en la creación de un motor de Marching Cubes con diferentes propiedades, como son la creación de un terreno aleatorio e infinito y un sistema de guardado del estado del terreno. Este conjunto de objetivos se ha dividido en *sprints* para establecer un orden y coste de desarrollo de cada elemento del motor.

El motor de Marching Cubes ha alcanzado todos los objetivos planteados inicialmente, permitiendo que sea estable y sencillo. El proyecto ha alcanzado el nivel básico esperado para el espacio temporal disponible y necesitara de un trabajo extra para alcanzar el nivel profesional deseado.

Abstract

The Marching Cubes algorithm was originally introduced by William E. Lorensen and Harvet E. Cline in 1987 for 3D rendering of medical data. This algorithm was used for the 3D reconstruction of 2D layer groups generated by medical scanners, although at present the algorithm has opened a small niche in the field of video game development, allowing the generation of editable terrain in real time quickly and cheaply. However, the options to implement it remain difficult because the engines that allow an easy implementation are private or paid to use. Seeking to democratize the Marching Cubes so that developers of any skill can start working on this, the idea of the project was born.

The objectives of the project have focused on the creation of a Marching Cubes engine with different properties, such as the creation of a random and infinite terrain and a system for saving the state of the terrain. This set of objectives has been divided into sprints to establish an order and development cost for each element of the engine.

The Marching Cubes engine has met all the objectives initially set, allowing it to be stable and simple. The project has reached the basic level expected for the available time and will need extra work to reach the desired professional level.

1. Introducción

Una de las peculiaridades de la informática es la capacidad de ir mejorando los algoritmos o encontrar nuevos usos a los que se habían destinado en un principio. Este último caso es lo que le sucede con el algoritmo Marching Cubes. El citado algoritmo fue inventado en 1987 como una unión multidisciplinar de la medicina y la informática grafica. Este permitía pasar de las capas 2D generadas por los escáneres médicos a representaciones 3D de la sección escaneada, siendo pionero en este tipo de presentación visual. El algoritmo se desarrolló de manera lineal durante décadas y orientado únicamente a su objetivo original, pero eso cambió con la liberación de la patente en 2005. La liberación de los Marching Cubes trajo consigo la posibilidad de explotar este viejo algoritmo en nuevos campos, siendo los videojuegos uno donde se incorporó unos años después de su liberación.

Si bien la liberación del algoritmo fue fundamental para que este se usara en otros campos, el incremento de la potencia de los ordenadores también influyó en gran medida en poder incorporar los Marching Cubes a los videojuegos. El algoritmo se aplicó concretamente a la edición del terreno, permitiendo una edición total y orgánica del terreno, siendo el Mine wars 2081 [1], en 2012, el primer videojuego en implementar el algoritmo. Sin embargo pese a tener ventaja de edición total frente los terrenos habituales (planos 2D con diferentes alturas) o ser más orgánico que los terrenos formados únicamente por cubos su uso sigue siendo reducido. El uso reducido del algoritmo se debe a que la mayoría de las compañías de videojuegos que realizan una implementación del algoritmo lo mantienen de forma privada, la otra opción es pagar una suma por un motor de Marching Cubes [2]. El secretismo de compañías, que no indiquen el uso del algoritmo, y que no exista una versión gratuita para implementar el algoritmo hace que este sea bastante más difícil de conocer o implementar. Pese a que las compañías no indiquen el uso de este algoritmo, la comunidad de jugadores y desarrolladores muchas veces afirman que se trata de Marching Cubes como en los siguientes casos [3], [4] y [5] (juegos que se comentaran en el documento).

El objetivo del proyecto nace de la dificultad que existe a la hora de desarrollar un juego que utilice este algoritmo, ya que requiere o un desarrollo propio o comprar un motor de Marching Cubes que termine de adaptarse bien a tu proyecto. Bajo la necesidad de democratizar el uso de Marching Cubes nace la idea del proyecto, enfocándose en generar un motor de Marching Cubes público en Unity 3D, cuyos usuarios serán desarrolladores que a pesar de no tener conocimientos muy profundos de programación puedan implementar el algoritmo en sus juegos. Este objetivo vendrá precedido de otros de menor rango, como es el de investigación de videojuegos que usen Marching Cubes, así como comprensión del funcionamiento del algoritmo.

El resultado del proyecto es un motor de Marching Cubes que alcanza una calidad óptima en el tiempo que se ha invertido en su realización y permite que sea la base de juegos sencillos o implementaciones propias del algoritmo por parte de otros desarrolladores. El resultado del proyecto posibilita conseguir una calidad profesional en el motor con un desarrollo extra, lo que haría que fuera un posible sustituto a versiones de pago de Marching Cubes apto para todo tipo de desarrolladores.

La estructura del proyecto comienza con un estado del arte que explicara los Marching Cubes, los sistemas de *chunks* y el estado actual de los terrenos en los videojuegos (campo que mejora los Marching Cubes). Luego se describen los objetivos principales del proyecto, que tratan de forma técnica que características busca tener en el motor de Marching Cubes generado. Seguido de la metodología, que explica cómo se organizó el proyecto. Después viene la implementación, que abarca cómo ha evolucionado el proyecto a lo largo del desarrollo del mismo. Se continua con el estudio económico, para visualizar los costes del proyecto. A continuación, vienen resultados que ha llegado alcanzar el proyecto una vez terminado, así como un análisis de la organización que ha llevado el proyecto comparado a la idea inicial. El último punto es la conclusión, que abarca si se ha obtenido un motor de Marching Cubes optimo y las futuras implementaciones que se podrían añadir a este.

2. Estado del arte

2.1. Marching Cubes

2.1.1. Historia

El algoritmo fue desarrollado por William E. Lorensen y Harvet E. Cline mientras trabajaban para General Electric, quienes lo presentaron en la conferencia de SIGGRAPH de 1987 [6], grupo organizado alrededor del interés de la computación grafica.

El algoritmo fue pensado como una forma de representar en 3D los resultados obtenidos mediante escáneres médicos, que presentaban los resultados como un grupo de capas 2D consecutivas, por ejemplo, la Figura 1A. Estos eran obtenidos de tecnologías como la tomografía axial computarizada (TC), la resonancia magnética (RM) o la tomografía computarizada de emisión monofotónica (SPECT). El algoritmo permitía transformar los resultados de los escáneres, que venían en capas 2D, en una reconstrucción 3D de la parte deseada mediante el uso de Marching Cubes, obteniendo un resultado como por ejemplo el de la Figura 1B, donde se reconstruye la estructura ósea de un cráneo, resultados ya obtenidos por William E. Lorensen y Harvet E. Cline cuando presentaron el algoritmo.

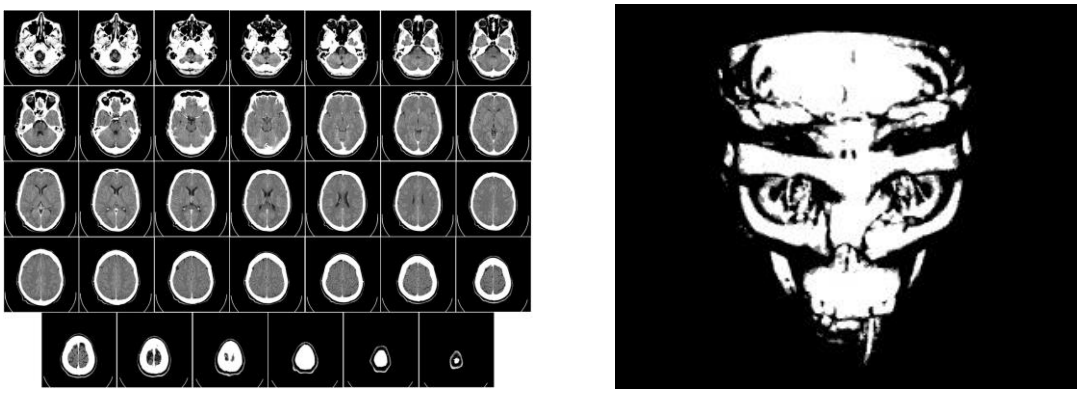

Figura 1 – A) Resultado TC. Fuente: https://en.wikipedia.org/wiki/CT_scan. B) Resultado Marching Cubes 1987. Fuente: Marching Cubes: A high resolution 3D surface construction algorithm

Para comprender un poco la evolución que sufre el algoritmo a lo largo de su historia hay que entender su funcionamiento, que será explicado en profundidad en el siguiente punto. En primer lugar, se divide en celdas la capas 2D obtenidas por el escáner, para después juntar las celdas contiguas de diferentes capas formando cubos. Dentro de cada cubo se pude generar un grupo diferente de mallas (*mesh* en inglés, es una superficie 3D generada a través de un conjunto de vértices con valores propios en un sistema de coordenadas). El conjunto posible de mallas diferentes que se pude generar dentro de un cubo es llamado los estados del cubo.

El primer algoritmo de Marching Cubes tenía una tabla de 15 estados del cubo (15 posibles mallas dentro del cubo). Este algoritmo inicial permitía construir modelos 3D, pero en algunos casos llegaba a producirse agujeros en la malla, problemas topológicos o discontinuidades. El primero en detectar estos problemas fue Martin J. Durst en 1988 [7].

Las primeras soluciones para evitar estos fallos en la malla pasaron por realizar cálculos extra, como podría ser el caso de Nielson and Hamann [8], que en 1991 crearon un test llamado "Asymptotic Decider", que conseguía recalcular el lado correcto que deberían mirar la cara de la malla. Otro ejemplo seria en 1994 cuando Natarajan [9] también realizaba cálculos extra, fijándose ser en los puntos críticos para evitar los fallos en la malla, así como añadir unos subcasos en la tabla de 15 posibles estados.

La mejora más popular fue la de Chernyaev [10], the Marching Cubes 33, que extiende a 33 el número de estados del cubo en Marching Cubes evitando así los fallos en la malla.

A partir del 2005 se acabó la patente permitiendo su uso de forma pública. Esto hizo posible que la gente pudiera empezar a usarlo de la forma que quisiese, creando avances tanto de forma pública como de forma privada, lo que ayudo a que se descentralizara la línea oficial de desarrollo, haciendo que existan proyectos relacionados con las Marching Cubes con diferentes niveles de profesionalidad.

2.1.2. Funcionamiento

Hemos visto que existen diferentes evoluciones de los Marching Cubes, del mismo modo que diferentes técnicas, el funcionamiento tratará de la primera versión presentada, que contaba con 15 posibles estados del cubo.

Como hemos explicado en la parte de historia, el punto de partida del algoritmo Marching Cubes son un grupo de imágenes 2D, por lo que se entiende que partimos de ese tipo de materia prima con la cual generaremos el modelo 3D.

El primer paso del algoritmo es dividir cada imagen 2D en celdas, para así comprobar si los vértices de las celdas quedan dentro o fuera del objeto, como se observa en la Figura 2A, donde los puntos verdes son vértices contenidos por la figura. Repitiendo este proceso con todas las imágenes, podemos agrupar las celdas contiguas de diferentes capas, para formar unos cubos, los cuales se obtienen al unir cuatro vértices de cada capa contigua, como se ve en la Figura 2B, a los cuales llamaremos por su término en ingles en adelante, *voxel*.

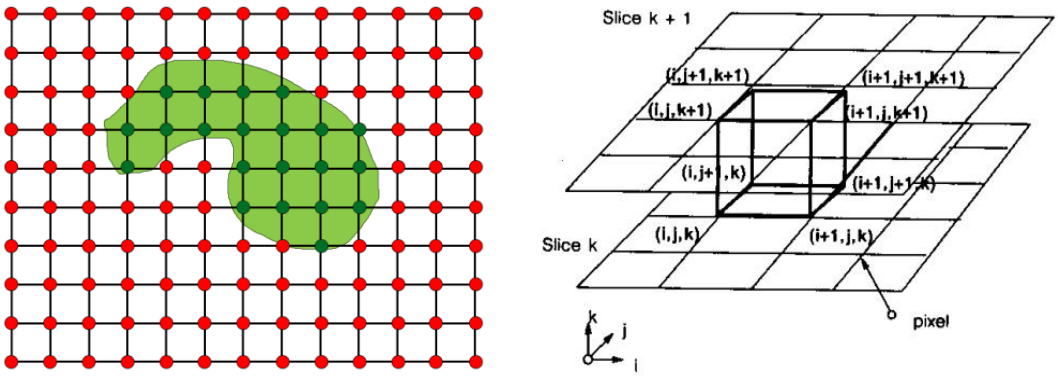

Figura 2 – A) Comprobación celdas. B) Unión de dos planos en un voxel Fuente: Marching Cubes: A High Resolution 3D Surface Construction Algorithm

Una vez calculada la pertenencia o no de los vértices del *voxel* pasamos a la segunda parte del algoritmo, la triangulación, donde se crea la malla para cada *voxel* en función de que vértices están contenidos dentro de la figura.

Siendo cada *voxel* un cubo, este contiene ocho vértices, donde cada vértice puede tener dos estados diferentes (contenido o no en el objeto), por lo tanto, tenemos $2^8 = 256$ posibles casos que pueden ser insertados en el *voxel*. Pese a tener un total 256 casos del *voxel*, es posible simplificarlos debido a la simetría que presenta la forma del cubo en solo 15 configuraciones diferentes, es decir podríamos representar todos los casos con solo 15 mallas diferentes.

Tratando en más profundidad el tema anterior, los 256 casos se pueden reducir aplicando diferentes propiedades. La primera propiedad que aplicamos es que si el estado de los vértices se invierte, la malla sigue siendo la misma, solo que los triángulos invierten su normal, como se aprecia en la Figura 3, donde el color rojo corresponde a la cara visible de los triángulos y el color azul corresponde a la cara invisible (la que no se renderizaría). Como se puede observar, cambiar el estado de todos los vértices solo intercambia el color de las caras, es decir la normal de los triángulos. La otra propiedad está relacionada con la simetría rotacional, la Figura 4 es un ejemplo de esta, donde todas las configuraciones de la izquierda coinciden con la de la derecha, con la única diferencia de que las configuraciones de la izquierda han sido resultado de aplicar una rotación diferente al *voxel* de la derecha. La suma de estas dos propiedades no permite dejar una tabla con 15 casos, como se pude observar en la Figura 5.

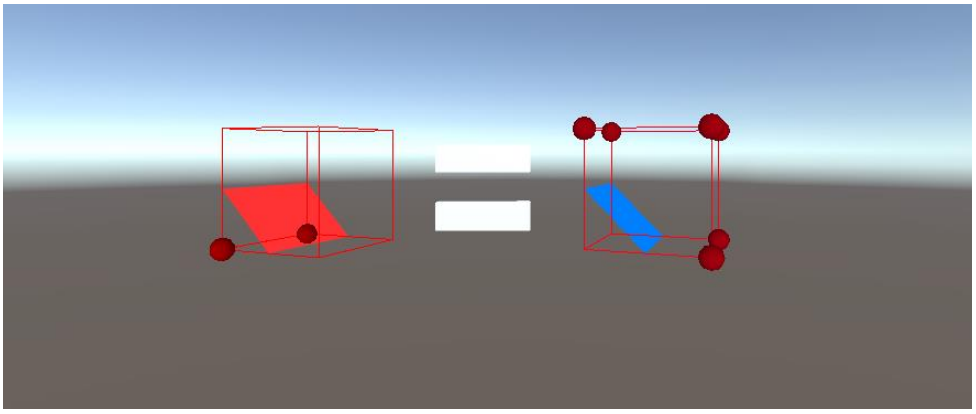

Figura 3 – Igualdad del voxel al invertir los estados de los vértices.

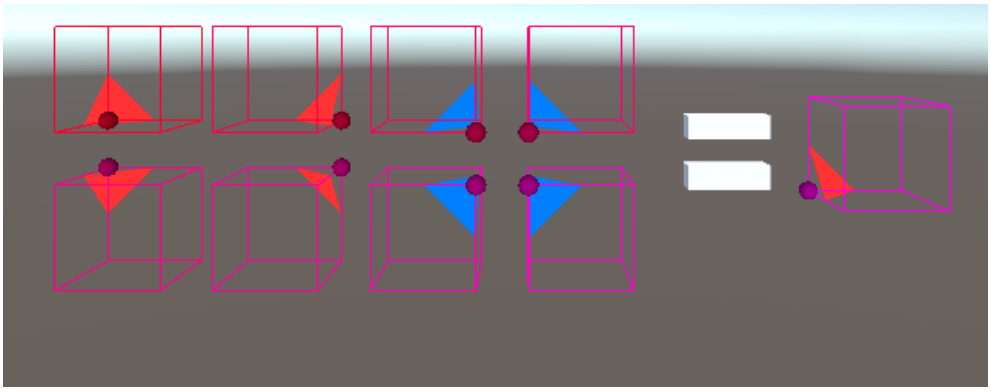

Figura 4 - Igualdad del voxel por simetría rotacional.

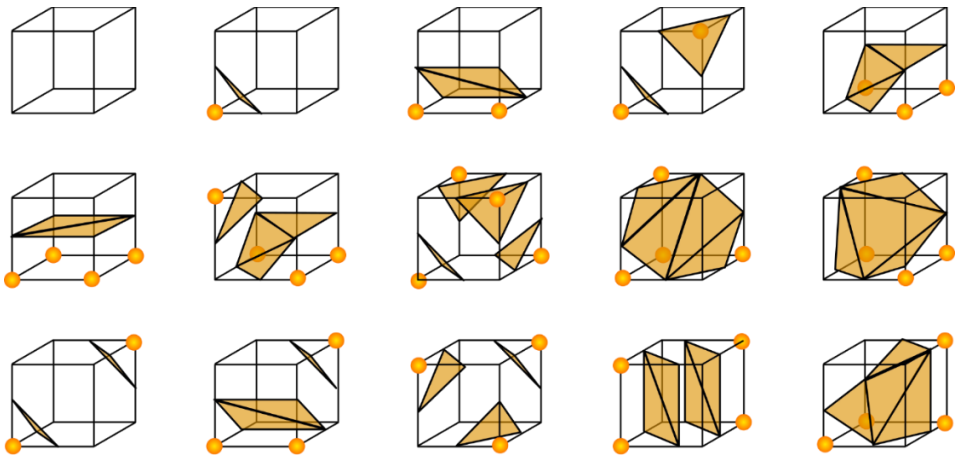

Figura 5 – Configuraciones del voxel. Fuente: https://es.wikipedia.org/wiki/Cubos_de_marcha

Siguiendo con la triangulación, después de explicar los posibles casos del *voxel*, partimos al punto de cómo saber qué caso necesitamos construir. Para conseguir eso tenemos los 256 casos en una tabla de triangulación, cada índice de la tabla nos indica que triangulación deberemos generar y para saber que índice debemos escoger nos ayudamos de Figura 6A, donde los indicadores "v" indican los vértices del cubo que se usaran para saber el índice de la tabla de triangulación, la cual nos devolverá un conjunto de puntos "e" que indican los vértices de la triangulación.

Por ejemplo, digamos que tenemos dos vértices, v1 y v4, dentro de la figura y queremos generar el *voxel* correspondiente, entonces sustituimos en el *array* index los vértices contenidos por 1 el resto por 0, obteniendo el resultado index = 00001001. Este resultado viene en binario por lo que su correspondiente decimal seria 9. Miramos el índice 9 en la tabla de triangulación y contiene "[e1, e9, e3, e3, e9, e11]", indicando que debemos formar dos triángulos utilizando esos vértices en orden, quedándonos el resultado de la Figura 6B.

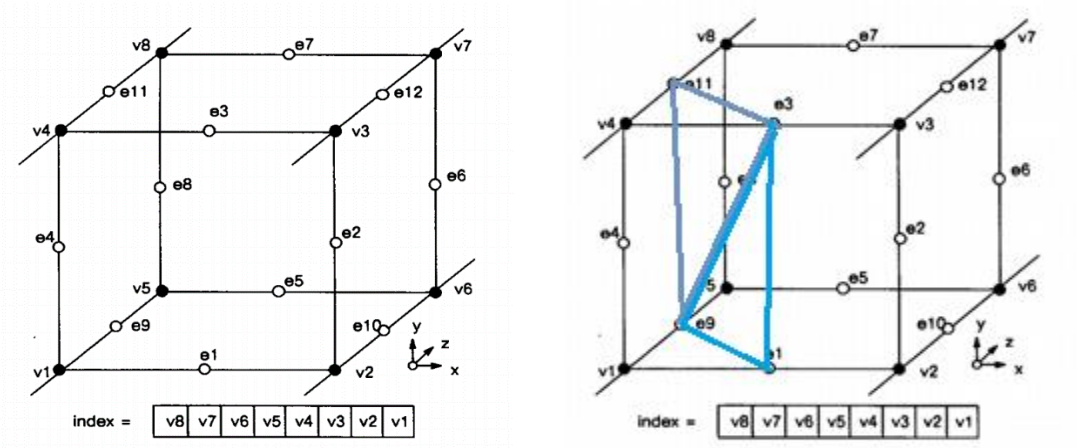

Figura 6 – A) Mapa vértices cubo. Fuente: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. B) Resultado tabla triangulacion, v1 y v4 contenidos.

Al final este proceso se va repitiendo con cada *voxel*, es decir, va recorriendo todos los *voxels* ("marching" en inglés) y generando el objeto 3D a partir de todos los *voxels* generados, por eso el nombre de Marching Cubes porque el algoritmo solo recorre y genera *voxels* sin saber el resultado final.

2.2. Sistema de *Chunks*

El problema a la hora de tratar con terrenos gigantes o infinitos es que no se pueden estas cargados dentro del juego en su totalidad. Por esa razón se recurre a cargar áreas separadas usando pantallas de carga o algún tipo de sistema que permita cargar y descargar partes del terreno en tiempo real.

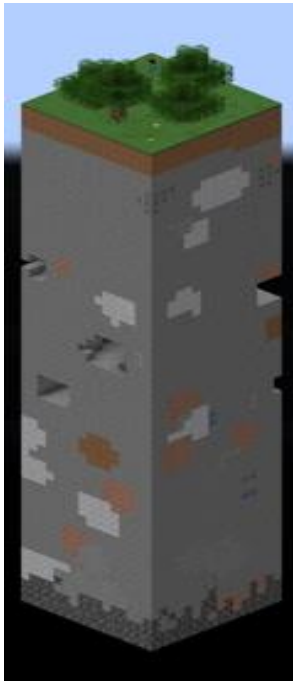

Figura 7 - Chunk Minecraft. Fuente: https://minecraft.gamepedia.com/Chunk

Este proyecto usa la carga y descarga en tiempo real como solución, recurriendo al *chunk system*. Éste consiste en dividir el terreno en porciones (cuyo término en inglés es *chunk*). Los *chunks* permiten cargar el terreno solo alrededor del jugador, cargando el terreno hacia el que se dirige o encuentra y descargando el terreno del que se aleja. Esta simple forma de descargar y cargar los *chunks* da un efecto de terreno infinito, puesto que el jugador no tiene pantallas de carga y siempre tiene terreno a su alrededor. La otra ventaja que aporta es que al cargar solo el terreno cercano al jugador la cantidad de recursos computacionales que utiliza es menor que si se cargara un terreno muy grande, mejorando la eficiencia del juego.

Como ejemplo de juego popular que integre este sistema tenemos el Minecraft [11], desarrollado por Mojang en 2011. Este juego entiende que un *chunk* es una entidad de 16x16 bloques de anchura y con 256 bloques de altura, visible en la Figura 7. Conforme el jugador se mueve por el mundo va cargando los *chunks* guardados en memoria y los que no están creados se generan de forma aleatoria. De esta forma el jugador puede moverse por el mapa infinito, generando más terreno y evitando que el juego colapse, ya que el juego solo mantiene cargado el terreno que el jugador tiene alrededor.

2.3. Terrenos en los videojuegos

Hay que entender que en todo momento nos referimos a juegos 3D, puesto que el concepto de Marching Cubes tiene sentido solo cuando hablamos de las tres dimensiones. A continuación, se explica los terrenos que se dan en videojuegos 3D, con sus ventajas e inconvenientes.

2.3.1. Terreno con un plano

Este tipo de terreno es el más extendido en el desarrollo de videojuegos, tanto por su uso histórico como por su reducido coste de renderización y sumando a modelos 3D, permite generar todo tipo mundos o terrenos. La inmensa mayoría de juegos usan un terreno plano en el que existen diferentes alturas, en el cual cualquier cavidad o agujero es un modelo 3D que no forma parte de este.

Este terreno suele bastar para la mayoría de los juegos que no necesitan modificación del terreno, ya que es terreno es un elemento estático, como se da en la mayoría de los juegos. En el caso de desear modificar el terreno, este solo nos da la posibilidad de subir o profundizar zonas en el eje Y de su topología.

Su uso habitual a lo largo de la historia de los videojuegos y la facilidad de trabajar con él, han conseguido que este tipo de terreno este soportado por todos los motores de videojuegos de última generación, así como herramientas en los propio motores para la edición de este.

Las mayores ventajas que presenta este tipo de terreno son la facilidad de su edición y la eficiencia (poco coste renderizado y cálculo de físicas). Y aunque luego sea necesario complementar este de otros modelos 3D, sigue siendo más eficiente que las opciones veremos a continuación.

2.3.2. Terreno 3D

Los terrenos con tres dimensiones reales son aquellos que pueden presentar agujeros o cavidades que no son un modelo aparte, si no que forman parte del terreno. En la mayoría de los casos, la razón de usar este tipo de terreno se debe a que se ha implementado un sistema de edición de terreno.

Existen juegos que han solucionado el problema sin el uso de Marching Cubes, como por ejemplo muchos juegos de estética *voxel*, que usan cubos únicamente para representar el terreno, como puede ser Minecraft (Figura 8A) o Trove (Figura 8B), donde todo el mundo está dividido en cubos.

La ventaja de este tipo de terrenos es la flexibilidad, ya que permiten generar mundos aleatorios, eliminar las pantallas de carga y dar libertad al jugador en temas como construcción y exploración. Aunque el coste computacional de estos es elevado, por lo que suelen ser parte de la mecánica principal del juego, en caso contrario sería una implementación contraproducente y por lo tanto seria omitida.

Para nuestro proyecto este tipo de juegos son un buen ejemplo del uso de la carga procedural de los terrenos y del uso de un sistema de *chunks*.

Figura 8 - A) Minecraft (Mojang, 2011). Fuente: https://www.minecraft.net/en-us/about-minecraft. B) Trove (Trions Worlds, 2016). Fuente: https://trovesaurus.com/biome=31/forbidden-spires.

2.3.3. Marching Cubes

Con el incremento de la potencia de los ordenadores, ha sido posible el uso de terrenos con Marching Cubes. Estos permiten crear terrenos similares a los terrenos 3D basados en cubos vistos en punto 2.3.2, pero con una estética más orgánica. Este tipo de terrenos están presentes en pocos juegos, ya que los desarrolladores que implementan Marching Cubes a su juego lo hacen de forma privada y partiendo desde cero.

En cuanto a los resultados obtenidos por este tipo de videojuegos hay diferentes variaciones:

1. Por un lado tenemos los que tienen una estática propia del sector indie, también llamado lowpoly, donde todo el universo del videojuego presenta una baja cantidad de polígonos que son visibles al jugador, como podrían ser Castle story (Figura 9A) o el Astroneer (Figura 9B), estos tipos de juegos presentan esta estética indicada antes y en cuanto el uso de los Marching Cubes, implementan la tecnología de la forma más simple, donde todo el terreno tiene la misma densidad de *voxels*, un ejemplo es la Figura 10, donde se ve que la triangulación del terreno es homogénea. Este tipo de proyectos son un claro ejemplo de cómo debería lucir y comportarse el proyecto de Unity 3D, que busca generar este tipo de terreno, ya que la estética busca ser similar, así como su uso homogéneo de la densidad de polígonos en el terreno.

Figura 9 - A) Castle story (Sauropod Studio,2017). Fuente: https://store.steampowered.com/app/227860. B) Astroneer (System Era Softworks, 2016). Fuente: https://store.steampowered.com/app/361420.

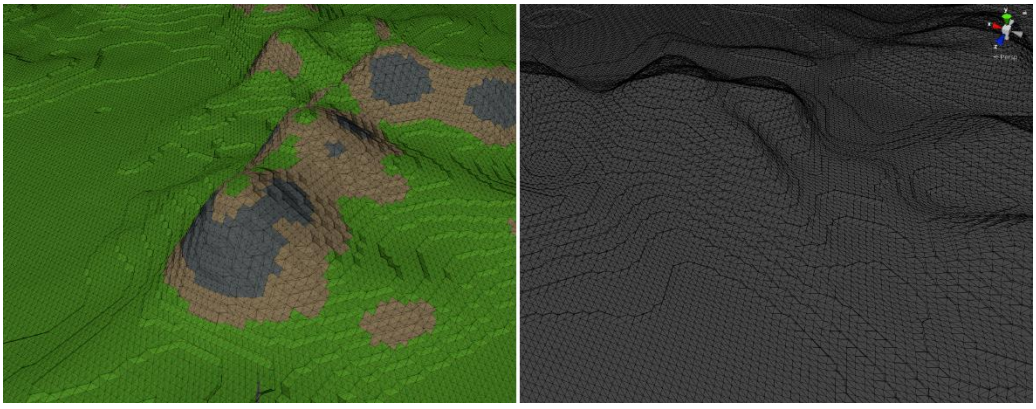

Figura 10 - Terreno poligonalmente homogéneo, imagen del motor del proyecto.

2. Por otro lado, tenemos un uso más avanzado de los Marching Cubes, donde en el propio terreno existen diferentes densidades de *voxels* a lo largo de este, permitiendo definir mucho mejor algunas secciones del terreno, así como su inversa, haciendo que la flexibilidad y eficiencia ofrecida por esto permita tener un terreno que sea mucho más orgánico, lo cual resulta en un terreno más realista, ya que solo se usa más densidad donde es necesario. Un ejemplo de este terreno es la Figura 12, donde se pueden apreciar partes del terreno donde la triangulación es mayor que en otras. Una disertación muy interesante que trata el uso de Marching Cubes a este nivel es la de Eric Stephen Lengyel [12].

   Este tipo de uso de Marching Cubes hace que sea más difícil de desarrollar este tipo de terreno, ya que su dificultad de desarrollo es superior a el indicado anteriormente, por lo que es más raro ver videojuegos con él, un ejemplo podría ser Space engineers (Figura 11), para ser precisos la compañía utiliza un motor propio, el cual mantiene bajo total secretismo por lo que no se pude confirmar totalmente su uso, aunque la comunidad defiende que el motor es resultado del uso de Marching Cubes [3].

Figura 11- Space engineers (Keen Software House, 2019) Fuente: https://www.spaceengineersgame.com/media.html

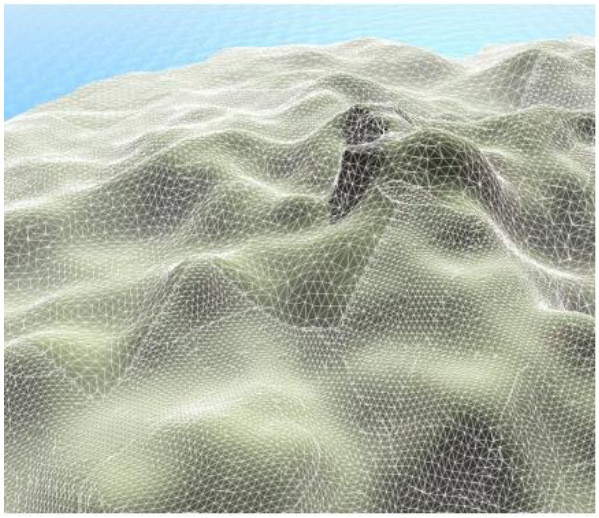

Figura 12 - Terreno diferentes densidades. Fuente: Voxel-based terrain for real-time virtual simulations

3. Objetivos

Antes del inicio del proyecto se establecieron unos objetivos, enviados con la propuesta de proyecto (Disponible en el anexo página 77). Los objetivos, siendo los mismos que en la propuesta del proyecto, se han reescrito para que queden de forma más clara los resultados buscados de cada uno, es decir no se han modificado ni en número ni en alcance.

Los objetivos que se definieron para el proyecto fueron:

1. Estudio del algoritmo Marching Cubes, así como las evoluciones dadas hasta su actualidad y los videojuegos que lo han implementado.
2. Implementar el algoritmo de Marching Cubes en Unity 3D, con un ejemplo sencillo, una representación de las configuraciones del *voxel*.
3. Usar la implementación de este algoritmo para crear un terreno sencillo que permita la implementación de un sistema de *chunks*.
4. Implementar un sistema de edición de terreno que sumado a lo anterior permita tener un terreno editable en su totalidad.
5. Implementar un sistema que permita generar terreno aleatorio usando todo lo conseguido en los puntos anteriores.

4. Metodología

Previo al inicio del desarrollo del proyecto, se realizó una fase de administración que permitiera organizarlo para definir todas sus tareas, así como el orden de estas o la duración de las mismas. Esto facilitaría su desarrollo dado que en todo momento se sabría que tareas realizar y cuanto deberían costar al alumno.

Fue en las primeras reuniones donde se estableció con el tutor la estructura que seguiría el proyecto y donde se depuró la misma, hasta obtener la aprobación del tutor. Las características que se definieron y demás elementos serán vistos continuación.

4.1. Administración del proyecto

Cuando hablamos del desarrollo del proyecto tenemos que hablar de la dualidad que tiene este, puesto que no solo trata de un proyecto informático (la implementación del algoritmo Marching Cubes), sino que también abarca el desarrollo de la memoria, con un gran peso en el proyecto, por lo que se debe tener en cuenta a la hora de organizar el mismo.

4.1.1. Elaboración de la memoria

La gestión de la memoria fue supervisada por el tutor académico que designó la universidad para el óptimo desarrollo de la misma. Se ha seguido el modelo de revisiones periódicas, mediante email y reuniones periódicas. Estas reuniones fueron registradas en unas actas (disponibles en el anexo, página 87) como se indica en el manual ofrecido por la universidad para el desarrollo del TFG.

4.1.2. Elección metodología desarrollo

La administración del proyecto se ideó pensando en ser organizado mediante una metodología ágil. Sin embargo, había que decidir sobre qué tipo de metodología ágil elegir, donde se destacaron Scrum y XP (Extreme Programming). Sus diferencias se pueden observar en la Tabla 1, las cuales ayudaron a seleccionar una de las dos.

La flexibilidad ofrecida de Scrum sobre XP y que la adaptación de Scrum para el tamaño del grupo también era más sencilla, así como que las buenas prácticas de XP eran innecesarias o difíciles de aplicar a un proyecto de una única persona, resultó en la elección de Scrum como metodología ágil elegida. Aunque sería Scrum bajo un par de modificaciones debido al tamaño del grupo.

Scrum	**Extreme Programming**
Los *sprints* tienen una duración de 2 - 4 semanas	Las iteraciones tienen una duración de 1 - 2 semanas
Cada miembro de del equipo trabaja de forma individual.	Los miembros deben seguir unas buenas prácticas, por ejemplo, programan en parejas.
Cambios en las tareas del *sprint* durante el desarrollo de este no están permitidas.	Se puede sustituir tareas de la interacción siempre que no se hayan empezado a realizar y la nueva tarea tenga un tamaño similar
Trata de seguir el orden de prioridades que marca el *Product Owner* en el *sprint backlog*, pero pueden cambiarse el orden si es para mejorar el desarrollo del proyecto.	El equipo de desarrollo sigue estrictamente el orden de prioridades definido al inicio por el cliente.
Una vez terminado un *sprint* las tareas del *backlog* terminadas y que cuenten con la conformidad del *Product Owner* ya no se retocaran.	Las tareas que se van terminando pueden ser modificadas durante el trascurso del proyecto, aunque funcionaran correctamente de acuerdo con lo que se estableció.

Tabla 1 - Tabla diferencias entre Scrum y Extreme Programming.

4.1.3. Aplicación de scrum

Al decidirse por trabajar usando Scrum se empezó a organizar el proyecto bajo las características propias de este, con ciertas modificaciones que veremos más adelante, debido a que el proyecto era desarrollado únicamente por una persona. Lo primero que se realizó fue el desarrollo de un *product backlog*, dividido en dos tipos de tareas:

- Tareas relacionadas con el código:
  - Realizar las 15 posibles caras del *voxel*.
  - Crear sistema de *chunks* (*chunks* y *regions*).
  - Modificar los *chunks* para generar un terreno simple (Generación de terreno llano y cargar/descargar *chunks* en función de la distancia).
  - Aplicar edición *in-game* de los *chunks* (Eliminar, añadir y cambias el material de este).
  - Sistema de guardado de terreno (Guardar cambios realizados y poder cargarlos posteriormente).
  - Sistema de ruido para generación de terreno aleatorio

    - Sistema de biomas (Diferentes tipos de biomas/ruido y la integración entre estos).
    - Revisar y pulir el código.
    - Adaptar el código para poder realizar una demo en la presentación del mismo.
- Tareas relacionas con la memoria:
    - Buscar lista de usos previos del algoritmo Marching Cubes.
    - Buscar documentación tanto teórica como de código relacionada con el algoritmo.
    - Desarrollar el punto de introducción.
    - Desarrollar el punto estado del arte.
    - Desarrollar el punto de objetivos.
    - Desarrollar el punto de metodología.
    - Desarrollar el punto de implementación.
    - Desarrollar el punto del estudio económico.
    - Desarrollar los puntos de resultados y conclusiones.
    - Realizar el resumen, anexos, comprobar bibliografía y pulir la totalidad de la memoria.

Para la administración de este *product backlog* se usó la herramienta online "Trello" [13], de la cual hay una imagen en la Figura 13. Esta herramienta permitió organizar los elementos del *product backlog* en cuatro diferentes columnas:

- ToDo: Refiriéndose a todas las tareas que quedan pendiente de trabajar.
- Trabajando: Las tareas en las que se están trabajando en estos momentos.
- Review: Las tareas que están pendientes de revisión. Estas debían ser aceptadas por el tutor (en caso de la memoria) o pasar las pruebas de aceptación (en caso del código).
- Completado: Todas las tareas que han sido cerradas de forma correcta formarían parte de esta columna y ya no deberían ser modificadas.

La herramienta también permitió indicar el carácter de cada tarea usando una etiqueta de color, donde la etiqueta verde correspondía a las tareas relacionadas con la memoria y la etiqueta azul correspondía a las tareas relacionadas con el código, como se ve en la Figura 13.

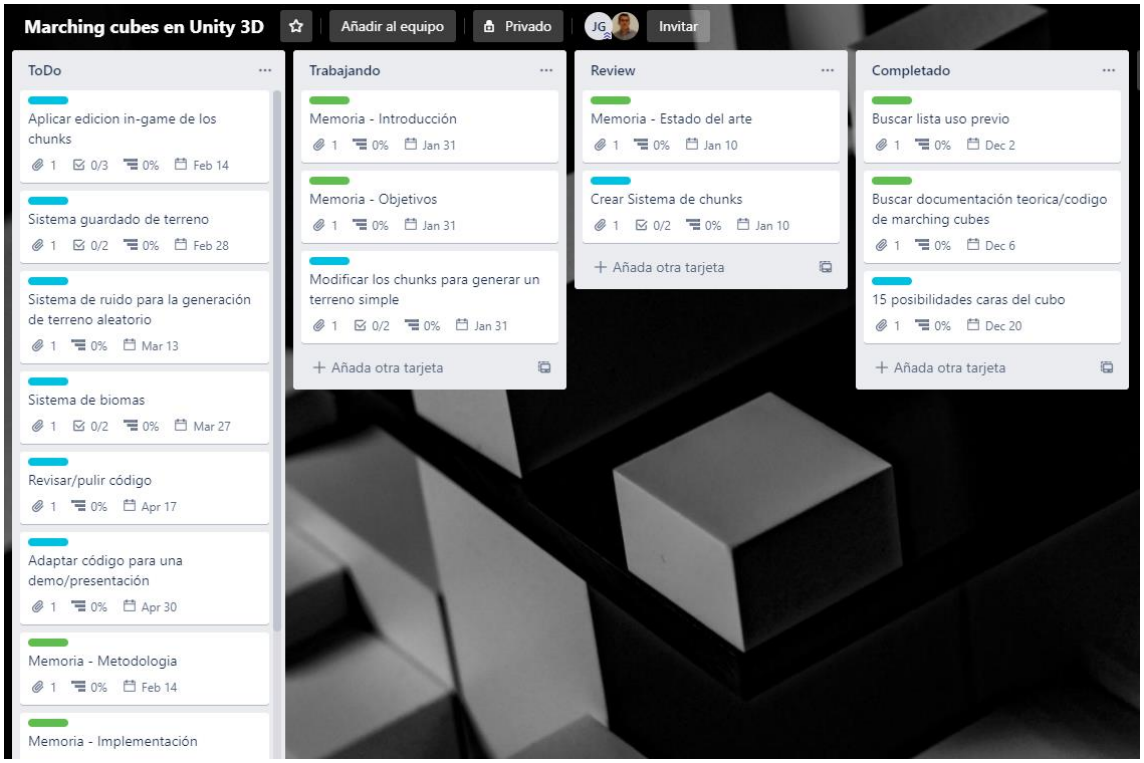

Figura 13 - Proyecto de Trello.

En cuanto la forma de organizar los *sprints* consistía en seleccionar el *sprint* correspondiente del conjunto de *sprint* backlog (explicados en el siguiente punto). Una vez se comprobaban las tareas que se habían seleccionado en ese *sprint*, se generaban unas pruebas de aceptación relacionadas con el código. Estas pruebas de aceptación eran una forma de sustituir al *product owner* que es el encargado de indicar si las tareas se han realizado de forma correcta y cerrarlas por completo, al no existir en el proyecto se sustituye por las pruebas de aceptación. Teniendo las pruebas de aceptación destinadas a la aprobación del código, la memoria contaría con el tutor como *product owner* para confirmar la correcta finalización de estas. Una vez terminado el *sprint* si se hizo alguna observación importante debía ser indicado junto a la tabla de *sprint* y las pruebas de aceptación. De esta forma se iniciaban y se daban por concluidos los diferentes *sprints* del proyecto.

De forma adicional y como sistema de copias de seguridad se decidió implementar un repositorio de GitHub (Figura 14) para subir el resultado de los *sprints*. Al terminar el proyecto, se debería generar un ".readme" y cambiar su visibilidad a público para que cualquier pudiera hacer uso del proyecto.

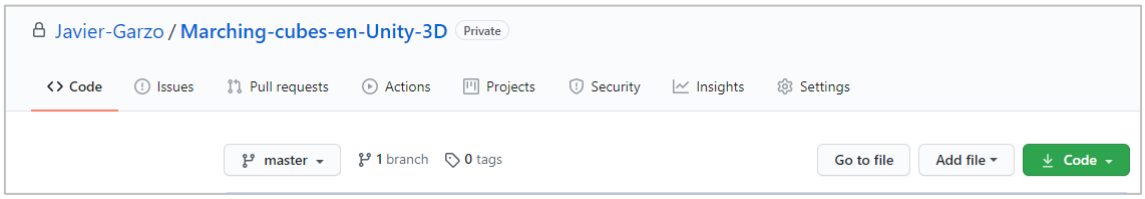

Figura 14 - Repositorio en GitHub del proyecto.

4.2. Planificación

Se estableció la forma de trabajar (*sprints*) y las tareas a realizar (*product backlog*), quedaba pendientes organizar y agrupar todas las tareas. Esta agrupación conocida en Scrum como *sprint backlog* consiste en agrupar las tareas en los *sprints* de dos a cuatro semanas de duración. En nuestro caso se usó la Tabla 2 como modelo, la cual contaba con un título, un ID, las tareas y se les debía asignar una prioridad, una dificultad y las horas estimadas de coste. Indicar que las horas estimadas de coste eran de 30 a 45 en función de las semanas que durara el *sprint* (2 -3 semanas). Por último, todos estos *sprints* están recogidos en el en el Anexo, página 78.

Título:		
ID:	Prioridad: (Alta, media, baja)	Dificultad: (Alta, media, baja)
Tareas - - -		
Horas estimadas: X horas		

Tabla 2 - Modelo de sprint.

Además de organizar el *sprint* backlog que dejaba estructurado el proyecto, se decidió realizar un diagrama de Gantt (Figura 15), esto permitía ver de forma visual el desarrollo del proyecto.

Todo este trabajo, previo al desarrollo del proyecto dejaba, cerrada la estructura del mismo, dejando claro el orden de las tareas y el conjunto de tiempo que debería destinarse a cada *sprint*.

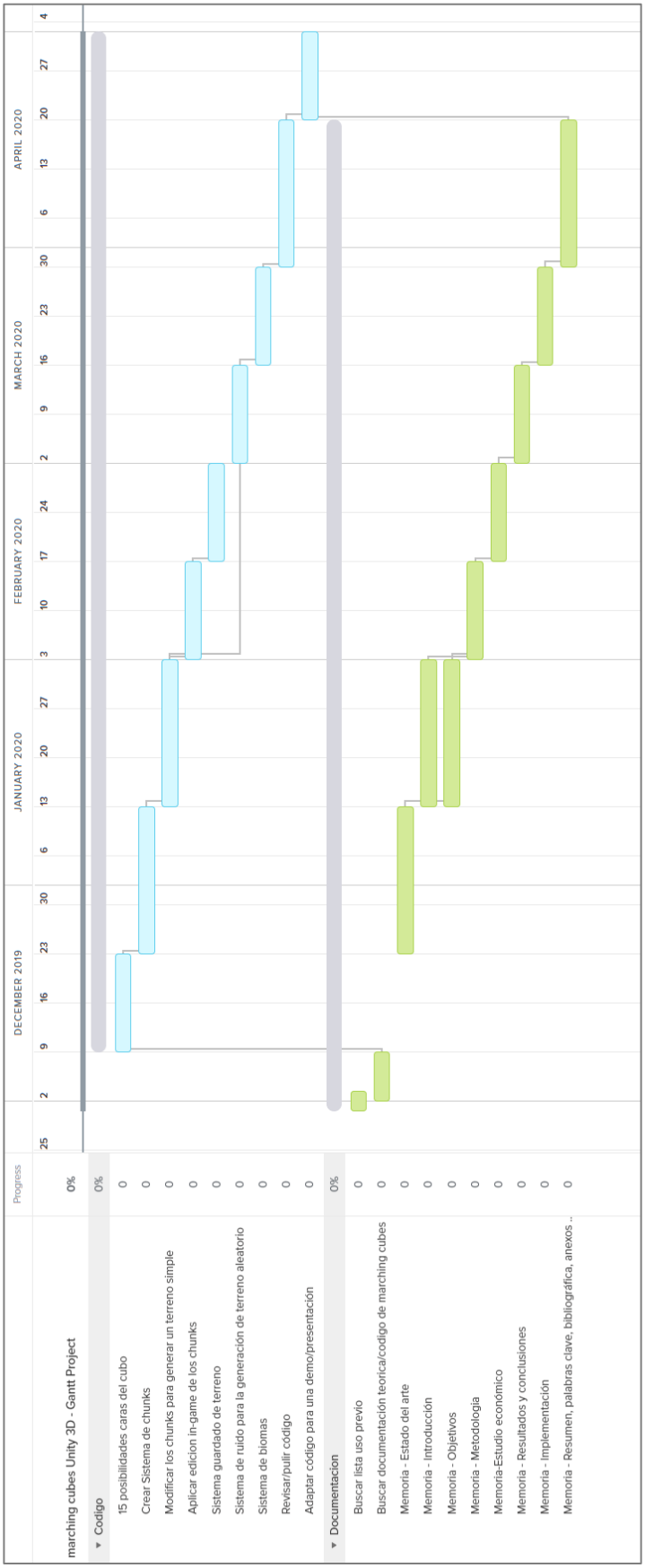

Figura 15 - Diagrama de Gantt del proyecto.

5. Implementación

Una vez se terminó de definir la metodología en las primeras reuniones y organizar la administración del proyecto, se inició la fase de desarrollo. Esta fase es la que se tratará en este punto, el cual abarcará una gran parte de la documentación.

A lo largo de los siguientes puntos se podrá ver el desarrollo de cada *sprint* de forma individual (centrándose en el desarrollo del proyecto informático), tanto la parte inicial de tareas y pruebas de aceptación, como el desarrollo producido a lo largo de este o los resultados finales del *sprint*. Por último, antes de continuar, indicar que los *sprints*, pruebas de aceptación y observaciones que se verán en adelante, se encuentran disponibles en el anexo, página 78.

5.1. Recogida de información e inicio del proyecto

La primera iteración destinada al desarrollo del proyecto, suponiendo el primer contacto con el código del algoritmo. El *sprint* (id: 0) estaría compuesto por una parte de investigación inicial, que se materializaría al final de este como una representación de las 15 configuraciones del *voxel*.

5.1.1. Tareas y pruebas de aceptación

La primera tarea fue una investigación inicial. Esta investigación se realizó para ver donde se había usado los Marching Cubes, campos en los que se usó y videojuegos que lo implementaron, y buscar cualquier documentación teórica o de código. Una vez terminada investigación se realizaría una sencilla implementación de los Marching Cubes en Unity 3D, los 15 posibles estados del *voxel*.

La prueba de aceptación de la iteración se basó en la visualización correcta de los 15 estados del *voxel*, similar a la Figura 5 del estado del arte. Hay que indicar que una vez terminada la fase de investigación inicial también se decidió añadir otra prueba de aceptación. Concretamente se propuso comprobar el correcto funcionamiento del sistema de interpolación en los *voxels*. Aunque las pruebas de aceptación se debían de definir al inicio del *sprint*, esta excepción se dio ya que no se había empezado a desarrollar la parte práctica (el código) del *sprint*, parte a la que van destinadas estas pruebas de aceptación.

5.1.2. Desarrollo

El desarrollo empezó con la parte de investigación, la cual era una búsqueda orientada a todo tipo de documentos e información. Esta búsqueda podía dar con documentos académicos [6] [12], documentos más sencillos [14], proyectos similares [15] o consultar comunidades al

respecto [16]. Todos estos lugares aportaron información valiosa y variada para el desarrollo del proyecto.

En la investigación de los campos en los que se ha usado el algoritmo, se descubrió lo que ya se sabía, este era usado para pasar de capas 2D a modelos 3D y cualquier campo relacionado con la construcción de mallas en tiempo real (los videojuegos principalmente). En cuanto el campo de los videojuegos se descubrieron juegos que los implementaban desconocidos, en especial el primero en usarlo "Mine Wars 2081" [1].

Una vez se finalizó esta investigación inicial, se empieza la parte de programación en Unity 3D basándose en el documento [14], el cual aportaba un código básico en C++ que implementaba los Marching Cubes. Aunque hubiera que adaptarlo para que fuera soportado por Unity 3D (c#), este código supuso una base sobre la que empezar el proyecto. Esto posibilitó realizar la clase "MeshBuilder", que permitía generar las mallas correspondientes al realizar el algoritmo Marching Cubes. También se decidió buscar una plantilla de *singleton* para que los códigos de tipo *manager* (se encargan de administrar elementos, en este caso la construcción de la malla) heredaran de esta y fueran *singleton* en el proyecto.

Durante el desarrollo del "MeshBuilder" también se implementó el sistema de interpolación, el cual consiste en que en vez de que los puntos sean contenidos o no contenidos (2 estados), a estos se le asigna un peso. Este peso digamos entre 0 y 100 indica la cantidad de volumen en ese punto. Para esto se asigna un isoLevel, un valor donde los vértices con un peso por encima de este están contenidos dentro de la figura. La diferencia de peso entre estos sirve para aumentar el tamaño de los triángulos generados o reducirlo, lo que crea terrenos más orgánicos.

Con el código básico realizado, únicamente se necesitó crear un nuevo *script* para comprobar el correcto funcionamiento de los Marching Cubes, por lo que se desarrolló la clase "cubeConfigurations", que era capaz de generar los 15 estados del *voxel*. Una vez puesto a funcionar sirvió para corregir todos los errores presentes tanto en la clase mencionada previamente como en el "MeshBuilder", puesto que se observó que aun sin errores de compilación no funcionaba correctamente en la primera instancia.

Esto permitió desarrollar una versión muy básica de Marching Cubes, que únicamente renderizaban un *voxel*, por lo que en realidad no estaría el algoritmo completo hasta ser capaz de renderizar agrupaciones de *voxels*.

5.1.3. Diagrama del proyecto

Se generaron dos clases durante esta fase inicial del proyecto ("cubeConfigurations" y "MeshBuilder"). Igualmente se obtuvo una plantilla de *singleton* usada en el "MeshBuilder". Estas tres clases están representadas en el diagrama de la Figura 16.

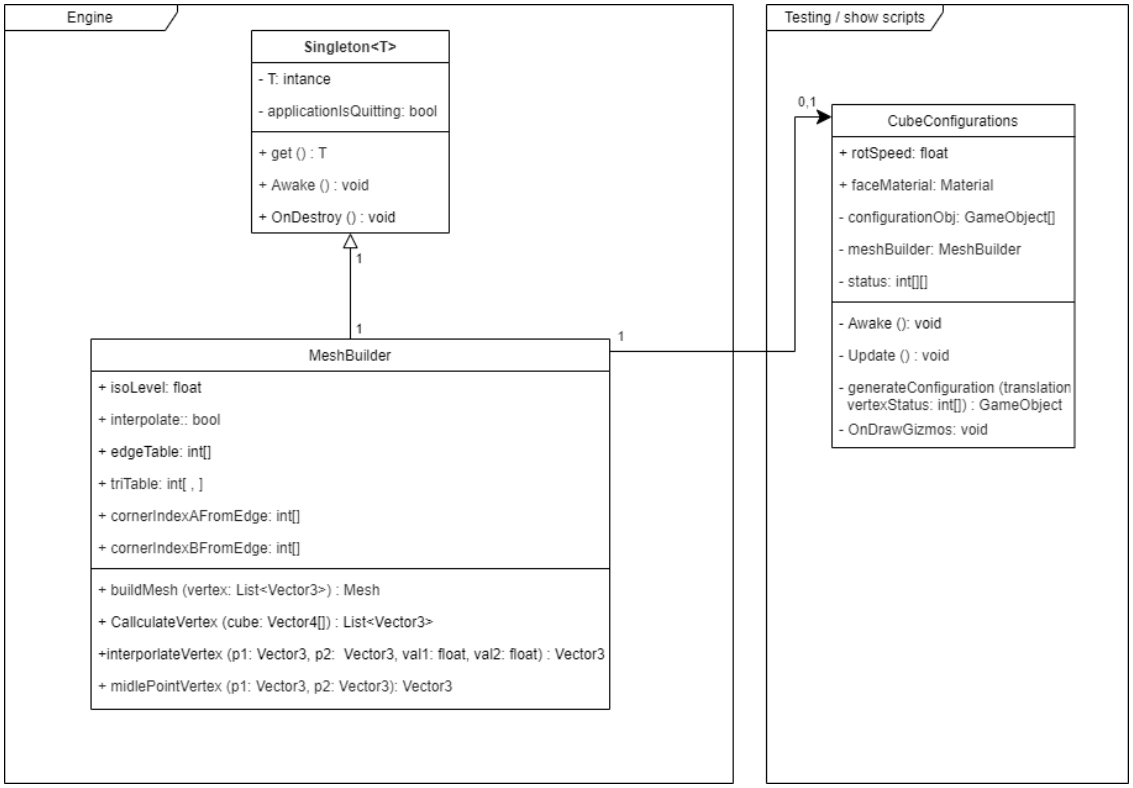

Figura 16 - Diagrama del Proyecto en el sprint id: 0.

5.1.4. Resultados

El *sprint* estuvo formado por dos tipos de tareas muy diferentes entre sí, unas destinadas a la investigación y otras destinadas al desarrollo del código. Los resultados de la investigación son difíciles de indicar de forma concreta, ya que fueron conocimientos que fueron usados a lo largo del proyecto tanto en la parte teórica como práctica. Pero sí que es cierto que los conocimientos teóricos adquiridos en esa investigación inicial están condensados en la introducción y el estado del arte.

En cuanto la parte destinada del código se obtuvo una implementación inicial del algoritmo, que permitía renderizar *voxels*. Aun sencilla, era un gran paso ya que al aplicar el mismo método a grupos de *voxels* tendríamos el algoritmo completado. Si hablamos de los resultados más visuales que se obtuvieron (Figura 17), fueron similares a Figura 5 de las 15 configuraciones del *voxel* de

wikipedia, presentada en el estado del arte. Si aplicamos la interpolación y otro *shader*, este deja las caras que deberían renderizarse en rojo y las que deberían estar ocultas en azul, el resultado de la Figura 18.

La interpolación provoca que los vértices de los triángulos no cojan el punto intermedio y se aproximen más a un vértice que otro en función del peso de estos. En la figura de ejemplo (Figura 18) los vértices marcados son más pesados, lo que hace que los triángulos generados sean mayores, por decirlo de otra manera, los vértices de la malla generada estén más lejos de estos puntos.

La creación de las diferentes configuraciones del voxel demostraron la superación de las pruebas de aceptación pensadas para este *sprint*, tanto del resultado con interpolación o sin ella.

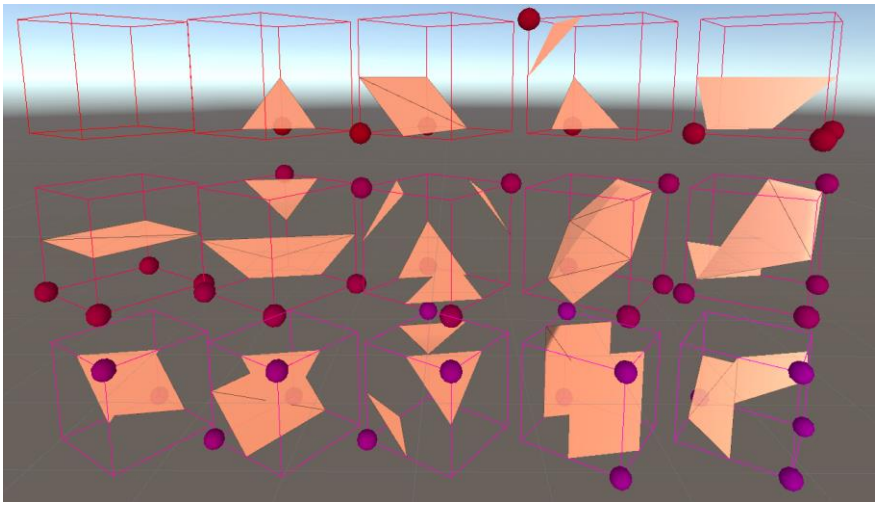

Figura 17 - 15 configuraciones del voxel del proyecto, versión no oclusion y con shaded wireframe.

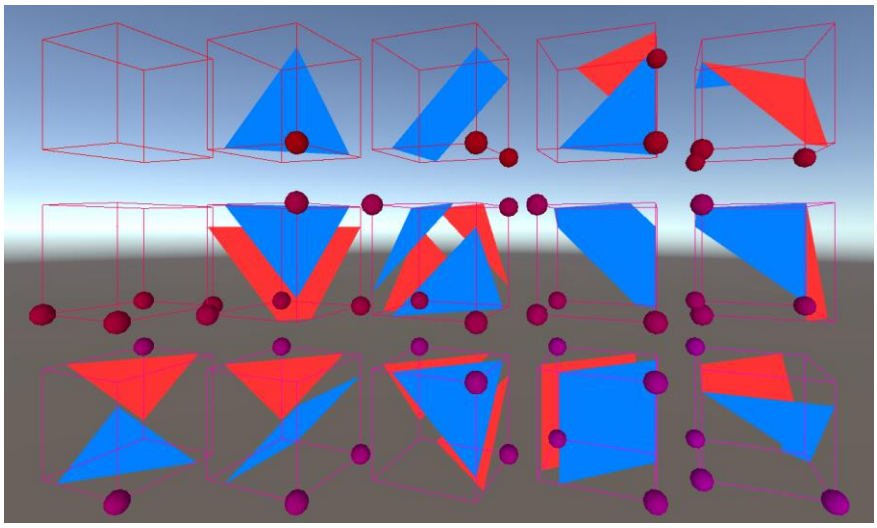

Figura 18 - 15 configuraciones del voxel con interpolación, caras ocultas (azul) y caras visibles (rojo).

5.2. Sistema de chunks

El segundo *sprint*, el cual poco tiene que ver con la ampliación del algoritmo Marching Cubes, ya que sienta las bases del sistema de *chunks*. Este *sprint* (id: 1), consistiría en desarrollar el sistema de *chunks*, aunque simplemente el código base sin poder llegar a tener una visualización del mismo.

5.2.1. Tareas y pruebas de aceptación

La tarea consistió en realizar el "esqueleto" de un sistema de *chunks*, refiriéndonos a desarrollar todo el sistema en la medida de lo posible para que en el siguiente *sprint* fuera combinado con el código de los Marching Cubes y poder generar un terreno llano. Por lo tanto, la tarea era crear todas las clases y métodos que permitieran cargar los *chunks*.

En cuanto la prueba de aceptación que se generó fue la de cargar un conjunto de *chunks* sin errores de ejecución, para comprobar el correcto funcionamiento del sistema. Esto se haría mediante un *GameObject* con un *manager* de *chunks* asignado.

5.2.2. Desarrollo

El desarrollo comenzó con idear de forma mental las características del sistema de *chunks*. Se decidió basarse en el sistema que poseía Minecraft [17][18] a la hora de dividir su mundo (visto más en profundidad en el punto 2.2). Este sistema consistía en *chunks* de dimensiones 16 x 16 y con una altura de 256 *voxels*, estos a su vez se agrupaban en *regions*, las cuales contenían 32 x 32 *chunks*.

Basándose en este sistema se crearon las clases "Chunk" y "Region", donde la segunda serviría para generar los *chunks* necesarios. La clase "Chunk" fue pensada para ser adherida a un GameObjet y se le añadió un método que permitiera inicializarlos pasándoles un *array* byte[]. Estos bytes serian usados para pasarle los datos y generar el mesh, solo que esto último se reservaría para el siguiente *sprint*. El array byte[] contendría los datos de todos los vértices del *chunk*, siendo cada dos byte un vértice. Este par de bytes tendría dos usos, el primer byte sería destinado a designar el peso del vértice (usado en la interpolación) y el segundo byte indicaría su material o el tipo de elemento que es.

Una vez terminada la clase "Chunk" se inició el desarrollo de la clase "Region", esta sería una clase que no heredaría de "MonoBehaviour" (editor de Unity 3D). Esta clase sería usada como un almacén de bytes de todos los *chunks*. Al generarse un *region* nuevo, este cargaría todos sus datos almacenados, pero esto estaba destinado al sistema de guardado, por lo que se llena el *region* con datos de prueba (mitad de los vértices con peso alto y la otra mitad con un peso bajo).

Para terminar se le asignó un método que permitía recoger los datos de un *chunk* en específico y devolverlos.

Esto dejó las clases del sistema de *chunks* terminadas, pero aun sería necesario un *manager*, por lo que se creó la clase "ChunkManager". Esta clase se encargaría de inicializar un *region* y a partir de este generar un conjunto de *GameObjects* y asignarles una clase "Chunk" e iniciarlos (llamar al método pasándole los datos extraídos del *region*).

Por último indicar que durante el desarrollo del *sprint* se decidió mover todas constantes a una clase "Constants", por ejemplo la altura de un *chunk*. Esto permite una configuración del sistema de *chunks*, únicamente con el cambio de las constantes. También se decidió mover las tablas de triangulación de la clase "MeshBuilder" a esta nueva clase, aunque estas no se pudieran convertir en constantes, para mantener la clase "MeshBuilder" más limpia.

5.2.3. Diagrama del proyecto

Se generaron tres clases nuevas, relacionadas con el sistema de *chunks*, y una clase para mantener el código más limpio ("Constants"). Los métodos y clases de la anterior iteración se han cambiaron de color a gris y las clases que no sufrieron cambios en sus métodos o variables se han comprimido. El estado que quedó después del *sprint* con id 1 se pude observar en la Figura 19.

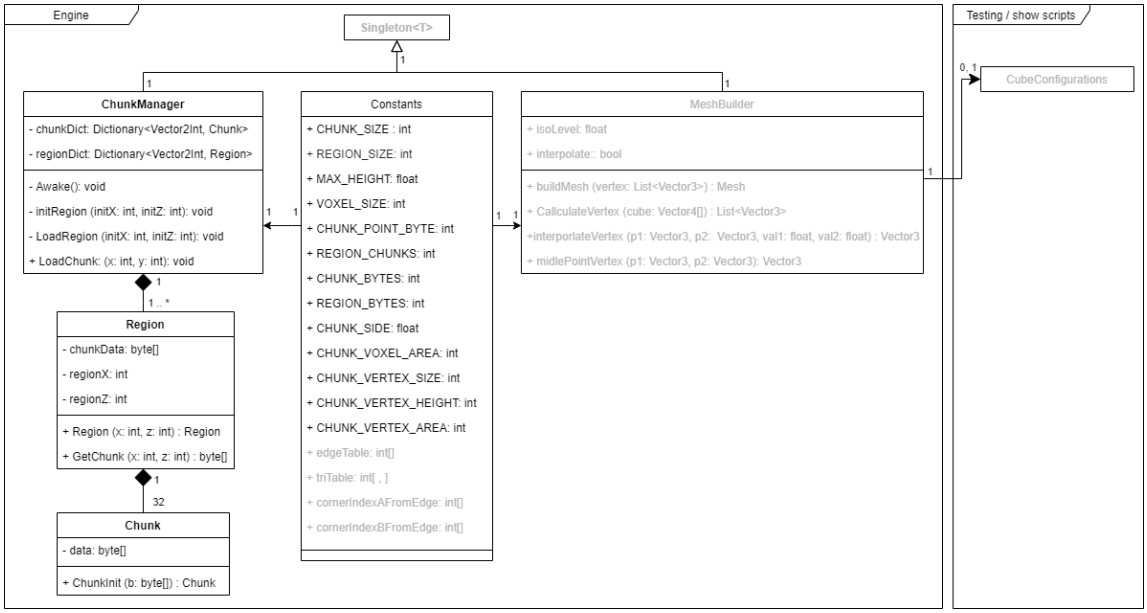

Figura 19 - Diagrama del Proyecto en el sprint id: 1, en gris los métodos y clases de sprint anteriores.

5.2.4. Resultados

El *sprint* consiguió de acabar el "esqueleto" del sistema de *chunks*, puesto que este sistema estaría sujeto a nuevas implementaciones conforme algunos *sprints* se realizasen. Para comprobar el resultado del correcto funcionamiento se hizo uso de la clase "ChunkManager" que obtuvo los resultados presentes en la Figura 20, donde se pude ver en la figura B el grupo de *GameObjects* generados y como cada *GameObject* tiene su propio *script* "Chunk" (figura A). Aunque los *chunks* no fueron visibles en el propio nivel, el sistema cargaba de forma correcta el conjunto de *chunks*, superando la prueba de aceptación que se designó en el *sprint*.

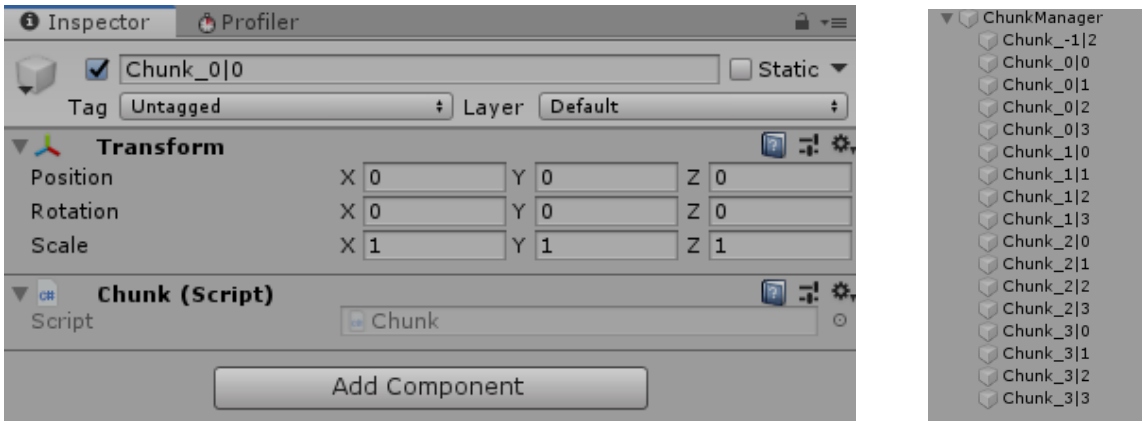

Figura 20 - A) GameObject con el Chunk asignado e iniciado. B) Carga de 3x3 chunks por el ChunkManager.

5.3. Terreno simple

El tercer *sprint* se encargó de realizar una unificación de los dos *sprints* anteriores y expandir un el sistema de *chunks* haciendo que estos se carguen y descarguen en función de la distancia al jugador. Combinar el sistema de *chunks* con el "MeshBuilder" permitió que se generasen unos resultados visibles del sistema de *chunks*, cosa que se echó en falta en el *sprint* anterior. Este *sprint* está recogido en la plantilla con id 2.

5.3.1. Tareas y pruebas de aceptación

Las tareas se dividieron principalmente en dos partes. La primera consistió en unificar los Marching Cubes creados en *sprint* con id 0 y el sistema de *chunks* del *sprint* 1, estos darían como resultado un terreno simple que permitiría visualizar los *chunks*. La segunda parte radicó en ampliar el sistema de *chunks*, cargando y descargando el terreno en función de la distancia al jugador.

Las pruebas de aceptación irían relacionadas con las tareas. Una consistiría en la generación correcta de un terreno llano y la otra en comprobar que el sistema de carga y descarga de *chunks* funcionara correctamente.

5.3.2. Desarrollo

El desarrollo comenzó juntando el sistema de *chunks* con el "MeshBuilder" por lo que se crearon unas nuevas funciones destinadas a la visualización del *chunk*. Estas nuevas funciones fueron generadas en el "MeshBuilder", las cuales consistieron generar toda la malla del *chunk* a partir de los datos del propio (byte[]), mediante Marching Cubes. Estas nuevas funciones se ejecutaban al crearse un nuevo *chunk*. Se paso de solo almacenar los datos del *chunk* en la clase a que la función "ChunkInit()" también utilizaba estos datos para obtener una malla para su *chunk*.

La unión de estas dos clases debía demostrar el correcto funcionamiento del "MeshBuilder" para generar una malla de más de un *voxel*, gracias a las nuevas funciones. Pese a ejecutarse sin errores y generar una malla, los resultados no eran los correctos, ya que la triangulación generada era errónea (fallaba la dirección de las caras de la malla y también la altura a la que se generaba). Esto se debía a que el número de *voxels* (16x16x256) en un *chunk* era diferente al de los vértices (17x17x257). Esta diferencia complicaba el código y generaba bugs difíciles de encontrar, por lo que se decidió crear unas funciones para visualizar los vértices del *chunk*. Con un pulido constante y gracias a las funciones generadas se consiguió una triangulación correcta, permitiendo generar la malla esperada en un principio. Como resultado se obtuvo el terreno llano de forma correcta y también unas funciones que permitían mostrar los vértices del *chunk* que se encontraban contenidos en el terreno.

Al tener terminada la forma de visualizar el terreno se comenzó con la segunda tarea, la de cargar y descargar los *chunks*. Esta implementación constaba de una distancia a partir de la cual nuevos *chunks* eran generados (extrayendo sus datos del *region* e inicializando el *chunk*). Los que superaban esa distancia eran ocultados (no se renderizan, pero ocupan RAM). Al incrementar aún más la distancia de los *chunks* ocultos, estos se terminaban eliminando (liberado memoria RAM). Los resultados fueron los previstos, aunque existía un inconveniente. El sistema tardaba demasiado en crear los *chunks*, lo que generaba una caída de FPS, algo que no es aceptable. Por lo que se optimizó el código para no cargar más de un *chunk* en el mismo FPS. Pese a haber una mejora, esto no llegó a solucionar el problema, por lo que se decidió aplazar su solución al *sprint* con id 7, dedicado a pulir el código. Además del sistema de *chunk* también se creó un pequeño *script* dedicado al movimiento de un jugador para testear la carga y descarga de los *chunks* en función de la distancia al jugador.

Luego de extender las clases de las dos tareas indicadas, se realizó un pequeño *script* para mejorar la comprensión de la reducción 256 casos a 15 de la metodología, llamado "CubeCaseReduction". Este era una modificación del "CubeConfigurations" para mostrar unos determinados *voxels* en pantalla, por lo que no necesitó demasiado tiempo.

5.3.3. Diagrama del proyecto

Como se puede observar en la Figura 21 la única clase nueva fue la de "CubeCaseReductions". Sin embargo, los cambios que se realizaron en el *sprint* se encuentran dentro de las clases del sistema de *chunks* y el "MeshBuilder", donde se extendieron las nuevas funcionalidades. En cuanto al *script* del movimiento de jugador "PlayerFlyMovement" no se incluye el diagrama, ya que no interactúa con el motor de Marching Cubes.

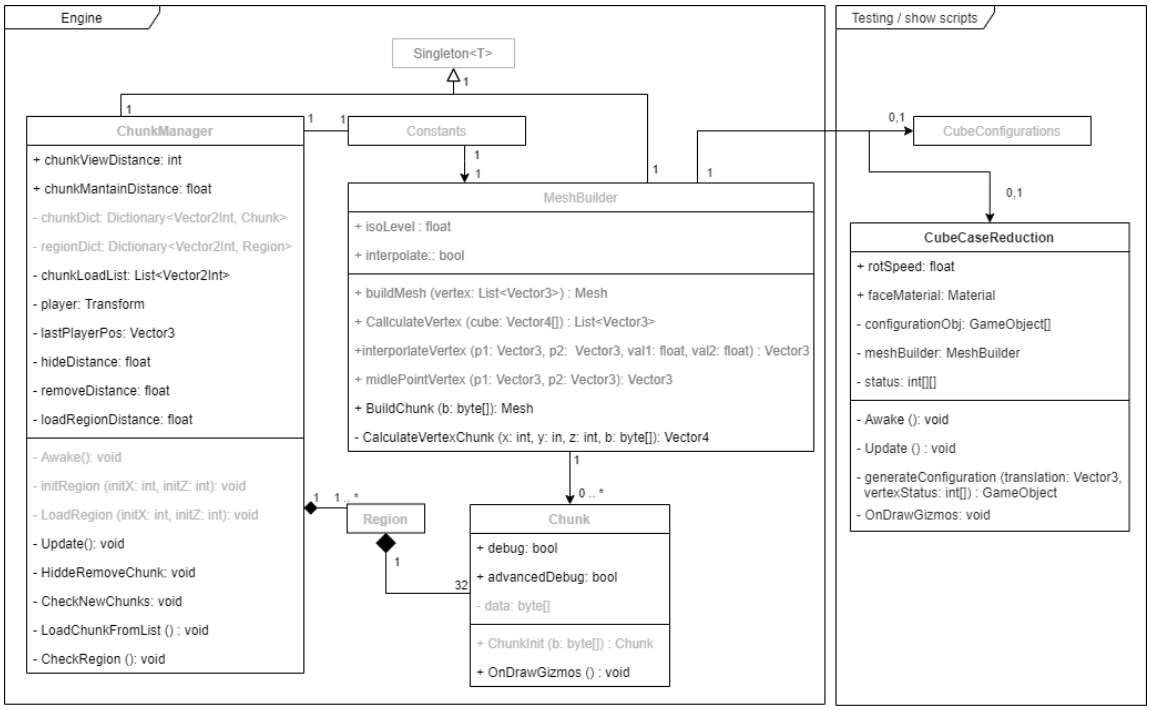

Figura 21 - Diagrama del Proyecto en el sprint id: 2, en gris los métodos y clases de sprint anteriores.

5.3.4. Resultados

Los resultados de la tarea agregaron la visualización del sistema de *chunk* disponible en la Figura 22, donde cada *chunk* coge un color aleatorio, que permitió generar un terreno llano con datos de prueba en los *chunks*. La otra implementación, la de carga y descarga de *chunks* en función de la distancia, era difícil de demostrar en una imagen, por lo que la única forma de comprobarlo era mediante la demo. También se recogió el *profiler* de Unity 3D en la Figura 23 (una ventana que permite de visualizar y registrar en tiempo real el rendimiento de los *FPS* de la escena de

Unity), donde se puede ver los picos que generan la caída de *FPS*, debidos a la carga de nuevos *chunks*. En cuanto al resultado de la clase "CubeCaseReduction", se encuentra en el punto del estado del arte, siendo la Figura 3 y la Figura 4.

Las pruebas de aceptación de la visualización del sistema de *chunks* se dio por superada. Sin embargo, la de carga y descarga de *chunks* se dio por superada, pero con problemas. El sistema funcionaba bien pero el tiempo de carga era inaceptable. El problema de la carga de *chunks* se derivó al *sprint* con id 7, destinado a pulir el código.

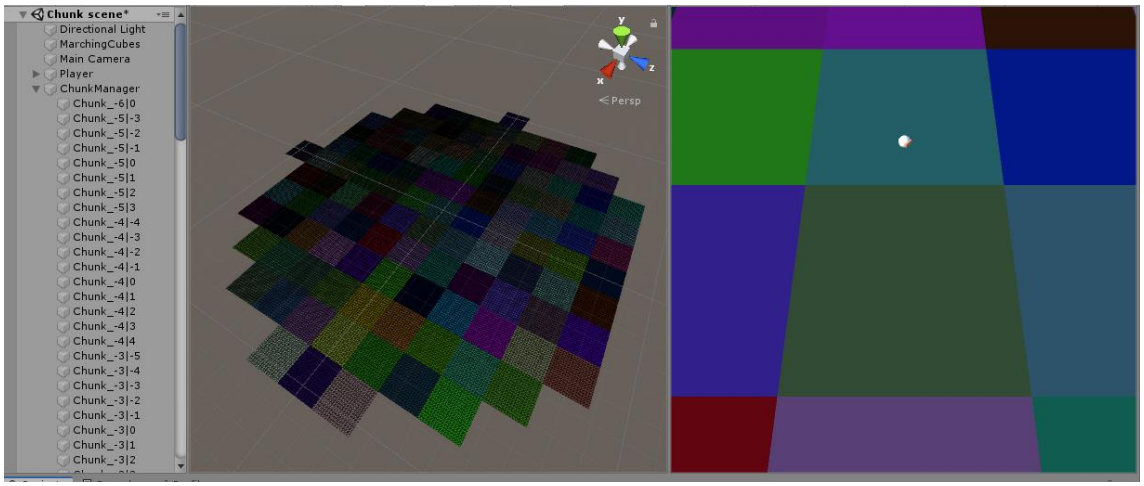

Figura 22 - Visualización del sistema de chunk (GameObjects, Scene y Game).

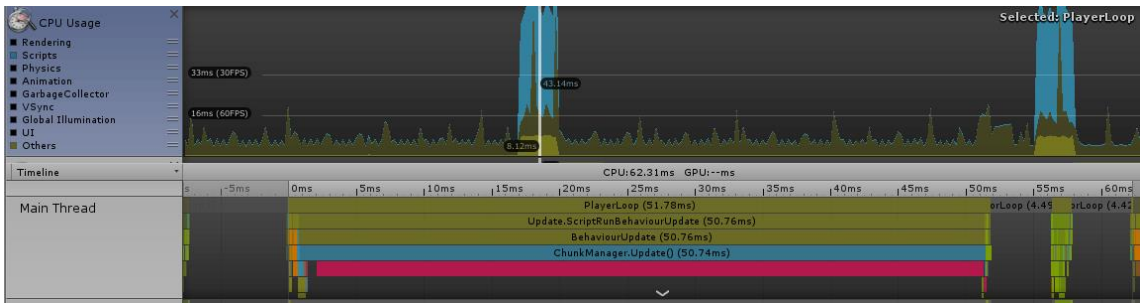

Figura 23 - Problema de la caída de FPS cuando se cargan nuevos chunks, picos azules en el profiler.

5.4. Edición de terreno

El cuarto *sprint* es donde ya se empezó a exprimir los Marching Cubes, permitiendo una edición en tiempo real del terreno. En este *sprint* también se añadió la texturización del terreno en función del tipo de material de los vértices del *voxel*. Este *sprint* con id 3, fue el primero que permitió una demo jugable del motor de Marching Cubes.

5.4.1. Tareas y pruebas de aceptación

La tarea principal consistía en implementar la edición de terreno, cuyas funciones eran eliminar, añadir y cambiar el color de los *voxels*. Pese a haber una única tarea planteada para el *sprint*, editar el terreno (cambiar los datos de los vértices del *chunk*), otra tarea se generó por la necesidad de representar el color de los *voxels*, debido a que aún no estaba implementado. Finalmente había dos tareas finales, edición de los datos de los vértices del *chunk* y texturizar los *voxels* en función del material de sus vértices.

Las pruebas de aceptación para demostrar la edición de terreno consistían en crear una estructura añadiendo terreno y crear un socavón eliminando el terreno. En lo que refiere al correcto funcionamiento del sistema de texturas, era necesario demostrar que la textura cambiara con la profundidad y que el nuevo terreno añadido tuviera otra textura.

5.4.2. Desarrollo

Siguiendo las dos tareas, arriba indicadas, se comenzó a crear la edición de terreno. Sin embargo, debido a que cada vez que se edita el terreno se debe recalcular la malla del *chunk* que es editado, se decidió reducir el tamaño del *chunk*. Se cambió la altura del *chunk* de 256 a 40, de forma temporal, lo que permitió eludir la caída de FPS a la hora de crear *chunks* (ya que los cálculos necesarios para crear un *chunk*, lo que generaba el problema, se redujeron a un 16%).

Se continúo realizando un código que indicara el punto donde se aplicaba la edición de terreno ("CameraTerrainModifier"). El funcionamiento era muy sencillo, un *raycast* era lanzado desde la cámara y se podía saber el punto de edición, los demás parámetros eran configurables en el inspector de Unity 3D y añadir o eliminar terreno dependía de si se pulsaba el botón izquierdo o derecho. Todos los parámetros anteriores eran pasados a una nueva función ("ModifyChunkData") creada en el "ChunkManager", esta función se encargaba de calcular todos los vértices en un área alrededor de un punto que se le pasaba como parámetro. Se calculaba el *chunk* al que pertenecía cada vértice y también su posición dentro del mismo (x,y,z), para así poder llamar a una función dentro del *chunk* e indicar la modificación que debía de aplicarse a ese vértice con el resto de parámetros de la función "ModifyChunkData". Las funciones del *chunk*

destinadas a modificar los datos no tenían mucha dificultad, cogían la posición del vértice en el byte[] de datos del *chunk* para añadir o restar en el primer byte del vértice (recordar que un vértice se formaba por dos byte, el primero destinado al peso y el segundo a la textura). Esto dejaba la edición de terreno terminada exceptuando la sección de cambiar la textura de los vértices, es decir, cambiar el color del terreno.

La texturización de terreno requería aplicar una modificación a la generación del mesh, indicando los *uv map* de los vértices (la posición que tiene cada vértice dentro de la textura). Para lograr esto fue necesario modificar las funciones encargadas de realizar los cálculos para generar la malla resultante de los Marching Cubes. Ahora estas funciones deberían soportar también otro elemento más sencillo, que sería el material de cada vértice. Se modificaron en general todas las funciones, el principal cambio viene en el cálculo de los vértices y el *voxel*. Cuando se calcula la posición real de los vértices de un *voxel* en la función "CalculateVertexChunk", en esta se registra el material de valor más bajo. El valor de cada material va en función del *grid map* (Figura 24*A*), donde la fila superior vale la que menos y dentro de una fila el material de la izquierda tiene menor valor. Pongamos un ejemplo, un *voxel* que tiene en los vértices superiores el material del césped con un valor de 0 y en los inferiores el material de la tierra con un valor de 1 (Figura 24B). El valor que se elegiría sería el 0, el material del césped. Luego a la hora de generar los vértices del *voxel*, se les asignarían a todos ellos el material 0, es decir el *uv map* del vértice apuntaría al material césped (0.2f, 0.8f). Este proceso se repetiría con todos los *voxels* y el resultado daría un terreno que presenta un material diferente en sus áreas.

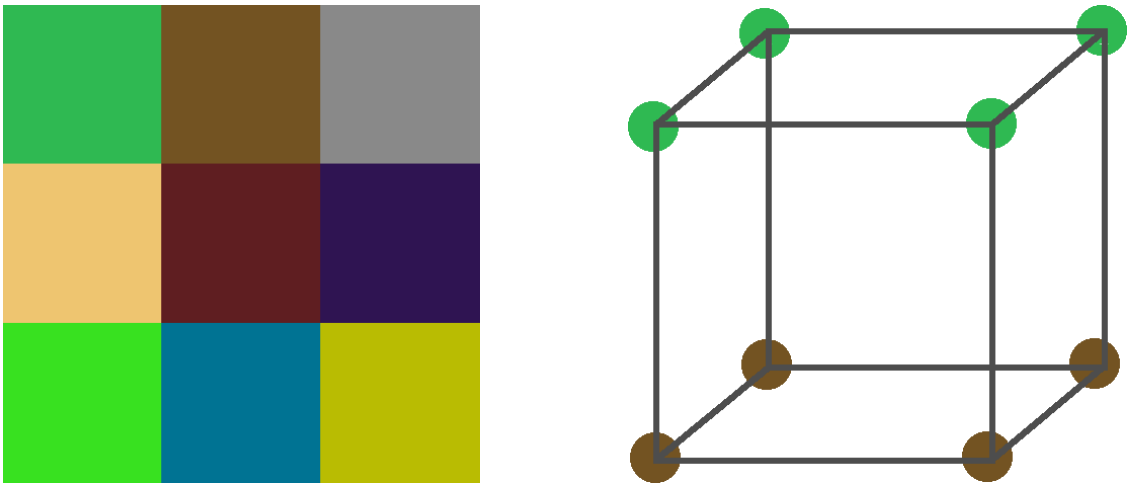

Figura 24 - A) Textura flat shader (grid map). B) Voxel cuyos vértices tienen el color de su material.

Este proceso era sencillo con el *flat shading* donde el material es un color plano. Usar una texturización estándar requirió de un proceso más complejo, debido a que los *uv map* de los

vértices tienen que colocarse en los diferentes puntos dentro de un material. También se implementó esta otra forma de texturizar el terreno, sin embargo, la solución no fue perfecta. Esta segunda implementación presentaba texturas en direcciones diferentes o pequeñas deformaciones en función de la triangulación que realizaba el *voxel*. Texturas como de césped o tierra no dejan muy visibles estos errores, texturas más geométricas (ej: texturas de ladrillos) dejan claro que no funciona perfectamente esta segunda implementación. Por cuestión de tiempo se decidió que ya era suficientemente optima la segunda implementación y se decidió terminar el sistema de texturizado.

Aunque se terminó el sistema de texturizado y la edición de terreno, quedaba pendiente un punto relacionado con este segundo, editar el color de los *chunks*. Este se solucionó con unas pequeñas modificaciones en la edición de terreno. Principalmente que cuando se superara el isoLevel al añadir terreno (el vértice ahora estaría incluido en terreno, es decir no sería aire), cambiar el byte consiguiente del del peso del vértice, es decir el del material. Esto permitió que al añadir terreno se pudiera elegir el tipo de terreno que se añade.

5.4.3. Diagrama del proyecto

Se realizaron cambios en casi todas las clases del proyecto, las cuales se pueden ver fácilmente en el diagrama (Figura 25). Debido a que el *sprint* tiene que ver con la extensión del sistema *chunks* y los Marching cubes, es normal estos cambios. Aunque hay que aclarar que las implementaciones ya realizadas en los anteriores *sprints* no se modifican, sino que se añaden nuevas implementaciones sobre estas.

Las funciones del "meshGenerator" que únicamente tenían un nuevo parámetro en negro, fue debido a que se añadió un parámetro nuevo a la función y modificado lo que realiza esta, pero la función ya estaba creada en *sprints* anteriores. También en la misma clase se pude observar cómo nuevas funciones solo son sobrecargas de otras funciones de *sprints* previos con más parámetros.

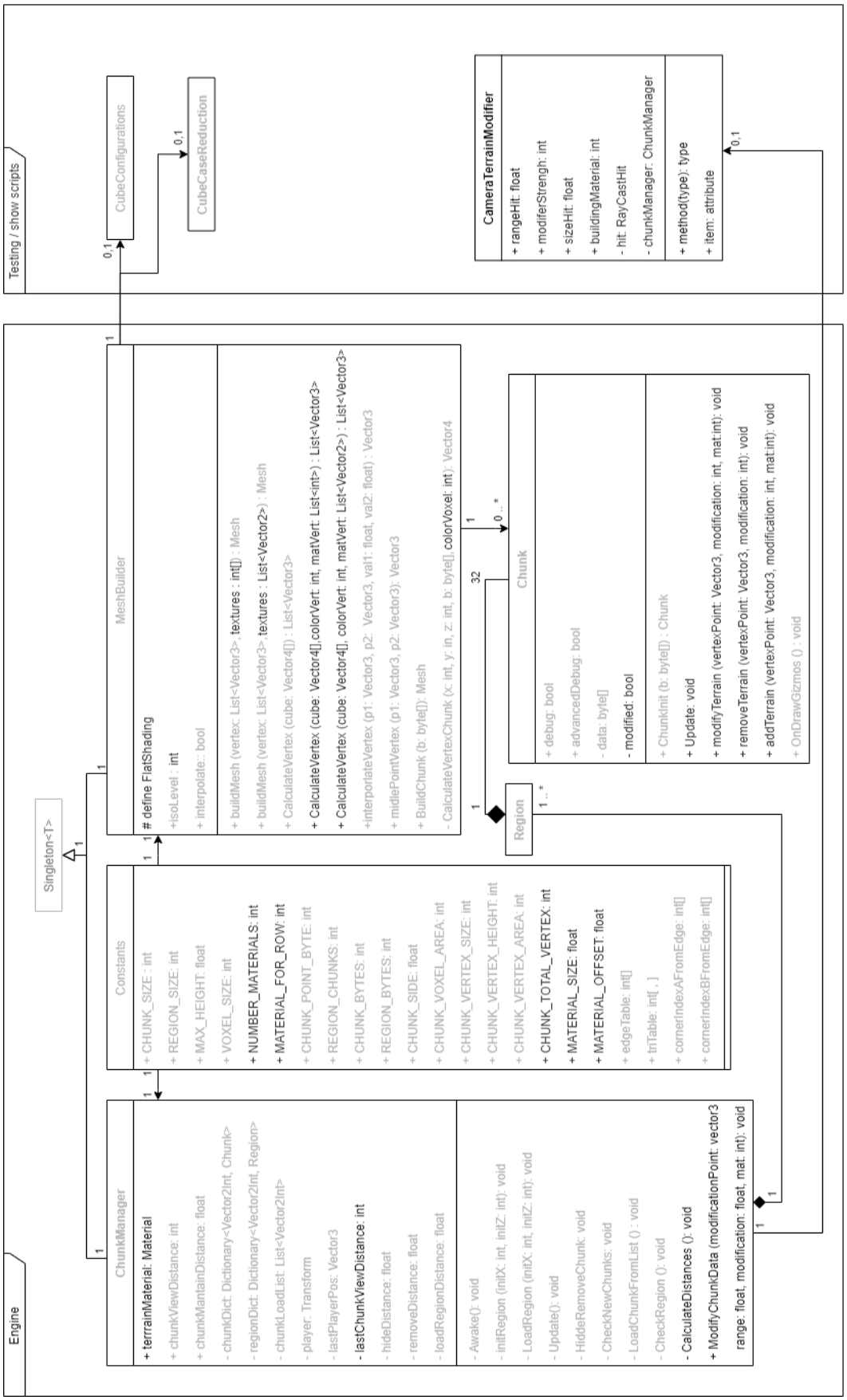

Figura 25 - Diagrama del Proyecto en el sprint id: 3, en gris los métodos y clases de sprint anteriores.

5.4.4. Resultados

Los resultados permitieron tener el primer nivel interactivo del proyecto, el cual permitía editar terreno. La Figura 26, permite ver un ejemplo de la edición de terreno donde se creó un arco y un socavón, previo a la texturización del terreno (donde cada *chunk* tiene un color). Los resultados de aplicar la texturización se pueden ver en la Figura 27 (tipo *flat shading) o en la figura* Figura 28 (tipo normal).

En cuanto a pruebas de aceptación, la de edición de terreno puede verse demostrada en cualquiera de las imágenes del *sprint*. La texturización del terreno con diferentes capas de material en función de la profundidad se superó para ambos tipos de casos, ya que se puede ver como el terreno cambia en función de la profundidad o las nuevas construcciones tienen otro tipo de material en los dos tipos terreno (Figura 27 y Figura 28). Indicar que en el caso de la texturización de tipo normal (Figura 28) se presentaban irregularidades dependiendo de la triangulación del *voxel*, sobre todo era visible en materiales geométricos como la madera o los ladrillos del ejemplo.

Es destacable que el *shader* usado para el terreno en ambos casos (Figura 27 y Figura 28) no era el definitivo, ya que presenta problemas con la iluminación. El *shader* solo tenía en cuenta la dirección de la luz por lo que no tiene sombras y proyecta luz en subsuelos cerrados. Este problema fue aplazado al *sprint* con id 8.

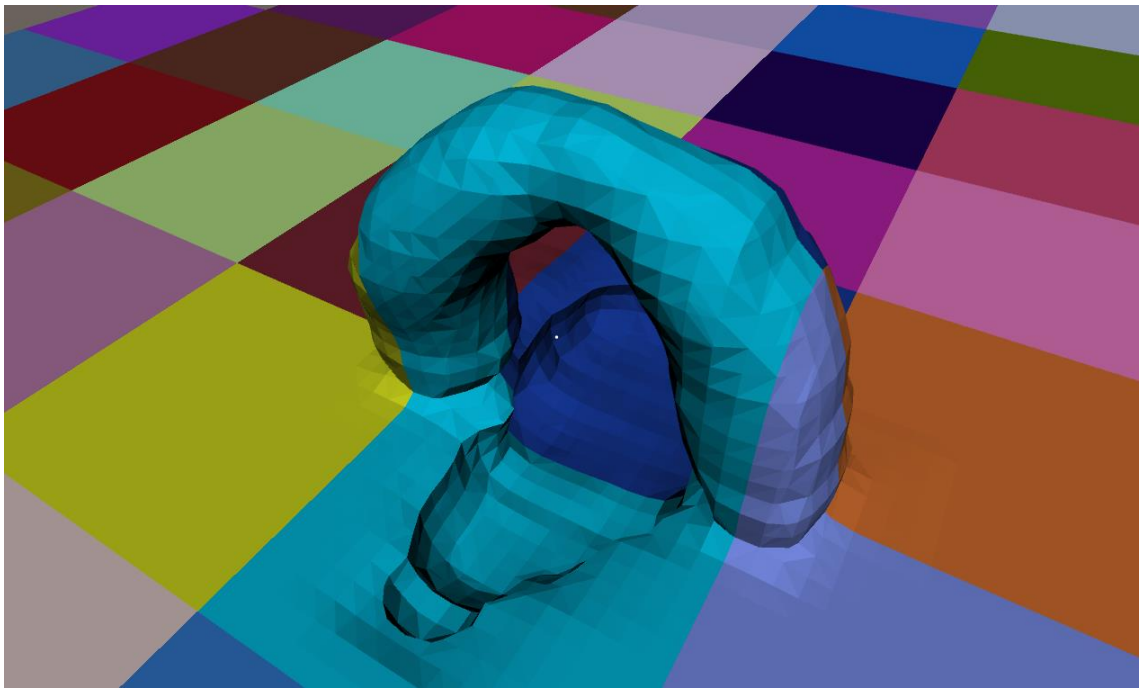

Figura 26 - Terreno editado, previo a la texturización el terreno (cada color es un chunk diferente).

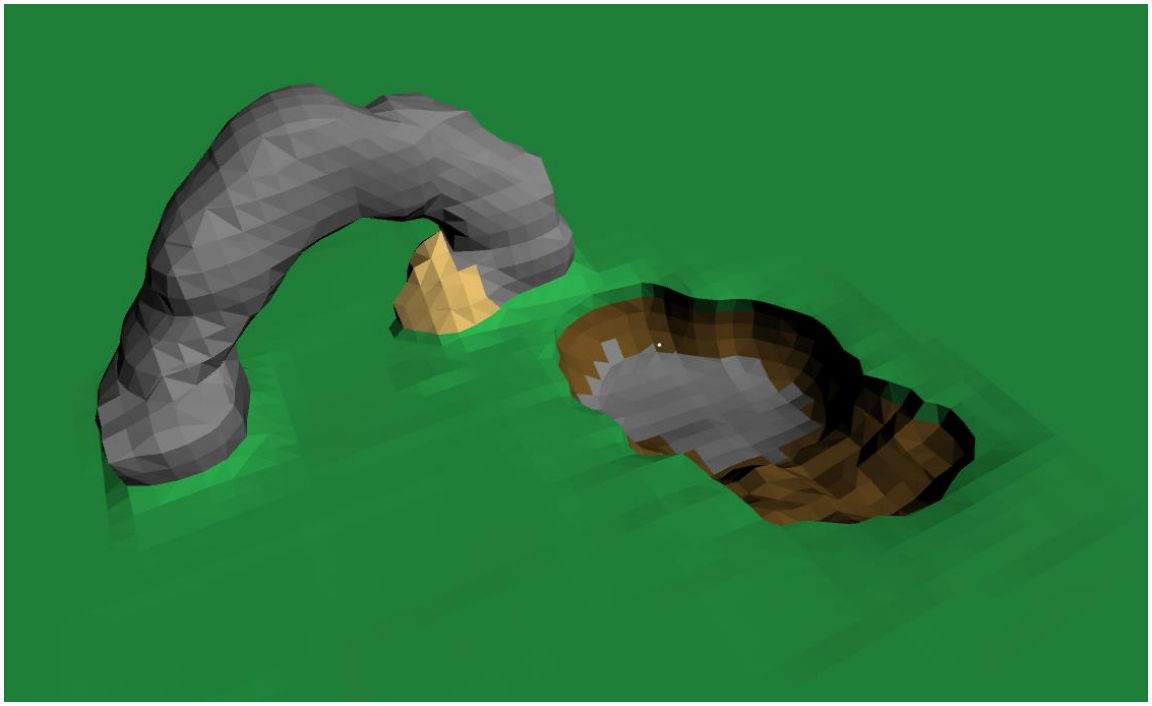

Figura 27 - Terreno modificado, texturización de tipo flat shading.

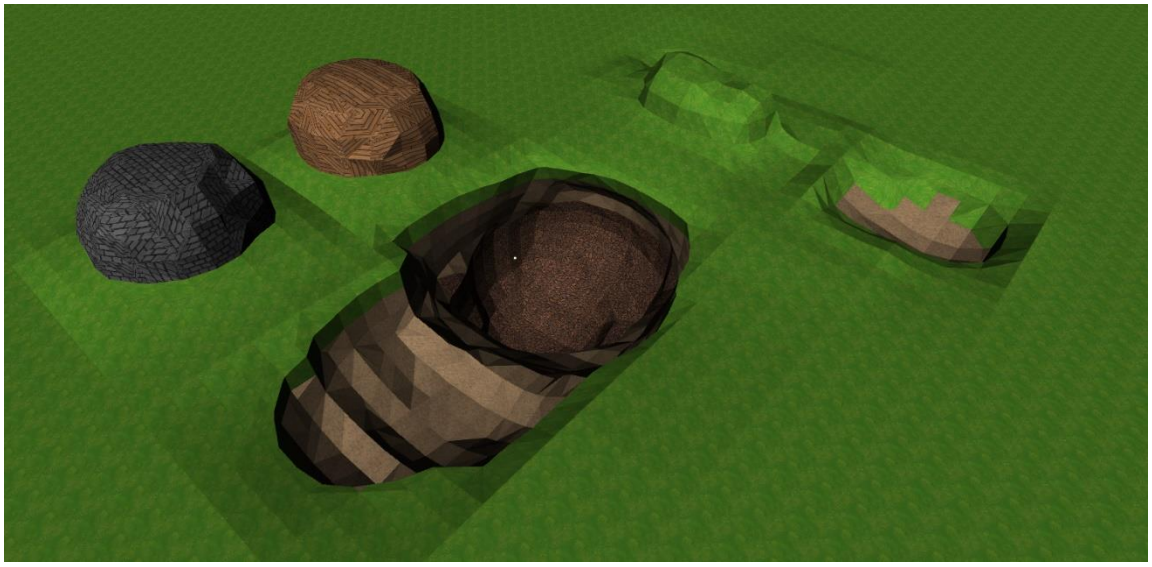

Figura 28 - Terreno modificado, texturización de tipo normal.

5.5. Sistema de guardado del terreno

Este *sprint* fue dedicado a hacer permanente cualquier cambio realizado en el terreno, mediante un sistema de guardado en archivos externos. Este *sprint* con id 4, fue el que terminó de pulir la edición de terreno, permitiendo hacer totalmente persistente el estado del nivel.

5.5.1. Tareas y pruebas de aceptación

La única tarea que se estimó para este *sprint* fue la de crear un sistema de guardado para el sistema de *chunks*. Esta tarea se podría subdividir en dos más específicas. Se deseaba guardar el terreno editado por el usuario en un archivo externo y cuando se iniciara el nivel de nuevo, los *chunks* que hubieran sido editados se extraerían de los archivos de guardado, para mantener el estado de la anterior vez.

La prueba de aceptación trataría únicamente de editar un terreno en una escena y parar la ejecución de la misma, después al iniciar de nuevo la escena los *chunk* generados deberían cargar los cambios realizados la última vez. La dificultad radicaba en que Unity 3D pierde los datos al parar e iniciar de nuevo la escena, por lo que estos deberían permanecer en unos archivos externos.

5.5.2. Desarrollo

El desarrollo empezó de forma similar al que se realizó en el sistema de *chunks* (inicio del punto 5.1.2), donde se cogieron de referencia el sistema de guardado de Minecraft [18] y en este caso también un blog [19] que explicaba cómo crear un sistema de archivos para un *voxel game*.

La tarea comenzó con guardar los *regions* en archivos externos, empezando por identificar qué *chunks* habían sido modificado. Detectar estas modificaciones era fácil, ya que los *chunks* que son editados sufren un cambio en un *bool*, esto sería igual solo que el *nuevo* bool no se volvería a cambiar a *false* al actualizar la malla del *chunk*. El único inconveniente era saber el *region* y la posición que ocupaba dentro de este cada *chunk*, por lo que se decidió guardar estos datos al crear el *chunk*. De este modo al cerrar la aplicación ("OnAplicationQuit") o al descargar *regions* (cuando el usuario se desplaza a unos nuevos) se correspondería a guardar los datos de estos. Al guardarse los *chunks* estos aplicarían cambios en los datos del *region* y al guardar este último se guardarían en un archivo "x.z.reg" (ej: "0.0.reg").

En referencia a almacenar los datos que contenía el *region* (regionData: List<byte>) consistía en escribir directamente estos bytes a un fichero externo. Los datos que contenía el *region* se modificaron para soportar dos partes diferenciadas entre sí, disponibles en Tabla 3. En el caso de

256 de altura y sin comprimir este fichero tendría un tamaño máximo de 145MB, sin embargo con compresión y guardando únicamente los *chunks* editados difícilmente se superaría los 10MB.

Bytes	0-2049	2050…
Sección	Tabla de posiciones	Tabla de *chunks*

Tabla 3 - Tabla datos del region.

Para entender esta estructura de guardado es necesario una explicación seguida de un ejemplo. La tabla de posiciones indica donde empieza cada *chunk* en el conjunto de la tabla de *chunks*. Su objetivo es que no tengan que estar todos los *chunks* inicializados y guardados en el *region*. Los primeros dos bytes (estimamos que estamos trabajando con un *region* de 32 *chunks*, 32 x 32 +1= 1025 ~= 2 bytes) indican la última posición asignada a un *chunk* y a continuación sucesivamente se indica la posición de cada *chunk* en la tabla de *chunk*. Hay que indicar que el valor 0 está reservado para indicar que el *chunk* no está creado, esto hace que el valor 1 indique la posición 2050 del *region* (posición inicio en la tabla de *chunks*). En cuanto a la tabla de *chunks*, simplemente son los byte[] de cada *chunk* consecutivamente.

Un ejemplo para entenderlo: Se desea guardar el *chunk* x:2 z:0, por lo que se busca su posición en la tabla de posiciones, (2 +0 x 32) x 2 + 2 = 6. Se lee los dos bytes correspondientes a las posiciones 6 y 7, donde se extrae el valor 0. Este valor indica que el *chunk* aún no existe por lo que se leen los dos primeros bytes de la tabla de posiciones obteniendo un 6, última posición en la tabla de *chunks* asignada, por lo que sumamos 1. El valor 7 se guarda en los bytes 6 y 7 de la tabla de posiciones, que indicará la posición del *chunk* en la tabla de *chunks*. Además se guardará en la tabla de *chunks* los bytes del *chunk* que deseamos guardar por lo que para calcular la posición a partir de la cual guardar estos nuevos datos, siendo (recordemos que el valor 0 está reservado y el 1 indica la posición 2050), 2050+ CHUNK_BYTES(147968) x (7-1) = 889858. Esa posición seria donde hay que empezar a escribir el byte[] de datos del *chunk*.

Después de definir la estructura que tendría el sistema de guardado se necesitaba cargar esos datos. Por lo que al inicializarse un *chunk* se comprobaba si existía su respectivo fichero de tipo .reg, en caso de encontrarse se cargaban los datos que estaban dentro del él, generando así un *region* con datos cargados de la anterior partida. Si no existía el archivo .reg se creaba una tabla de posiciones vacía y con ningún byte en la tabla de *chunks*. Este punto fue donde se pudo comprobar si funcionaba la parte de cargar como descargar y donde se realizó toda la corrección de *bugs* que no eran detectados al compilar.

Una vez acabado el sistema de carga y descarga seguía existiendo el problema de que los .reg podrían a llegar a ser enormes (145MB), por lo que se decidió implementar un sistema de compresión. Se buscaron librerías externas y posibles soluciones, .NET ofrecía su propia clase dedicada a este propósito ("System.IO.Compression"). Por lo que basándose en un post de stackoverflow [20] se creó la clase "CompressHelper" que permitiría llamarse a la hora de cargar y descargar archivos, ya fuera para comprimir o descomprimir los datos. Esto daba como resultado durante las pruebas que un archivo de 15MB se transformara en uno de solo 33KB, aun así, el ratio de compresión sería menor en un terreno aleatorio.

La correcta implementación del sistema de carga y descarga del terreno con la optimización del sistema de compresión dejaron por cerrada la parte de edición de terreno.

5.5.3. Diagrama del proyecto

En este *sprint* las clases relacionadas con el sistema de *chunks* han sufrido cambios y se ha generado la nueva clase "CompressHelper", la mayoría del trabajo se ha realizado en la clase del "Region" (Figura 29). La clase "Region" sufrió los mayores cambios ya que el sistema de guardado y la nueva estructura del *region* se han creado sobre ella. En cuanto la clase "Chunk" también almacena ahora unos datos extra para poder saber su posición en el *region* correcto, pese a eso los cambios de la clase "Chunk" han sido mínimos comparado con los de la clase "Region". En cuanto la clase "ChunkManager" solo se ha modificado la función "LoadRegion" para que guarde los *regions* que se van a descargar y también se ha añadido la función "OnApplicationQuit", la cual se llama al cerrar el juego o el editor de Unity 3D y guarda todo los *chunks* y *regions* activos. En referente a la clase "CompressHelper" solo tiene dos funciones usadas para comprimir o descomprimir un *array* de tipo byte[].

Se añadió un nuevo color en el diagrama de la Figura 29, el color rojo, que indica que esas funciones o variables han sido eliminadas. Pese a ser eliminadas, también están adyacentes a una función o variable parecida que las sustituye. Un ejemplo sería "chunkData: byte[]" que es sustituida por "regionData: List<byte>", su objetivo es el mismo almacenar los datos del *region*.

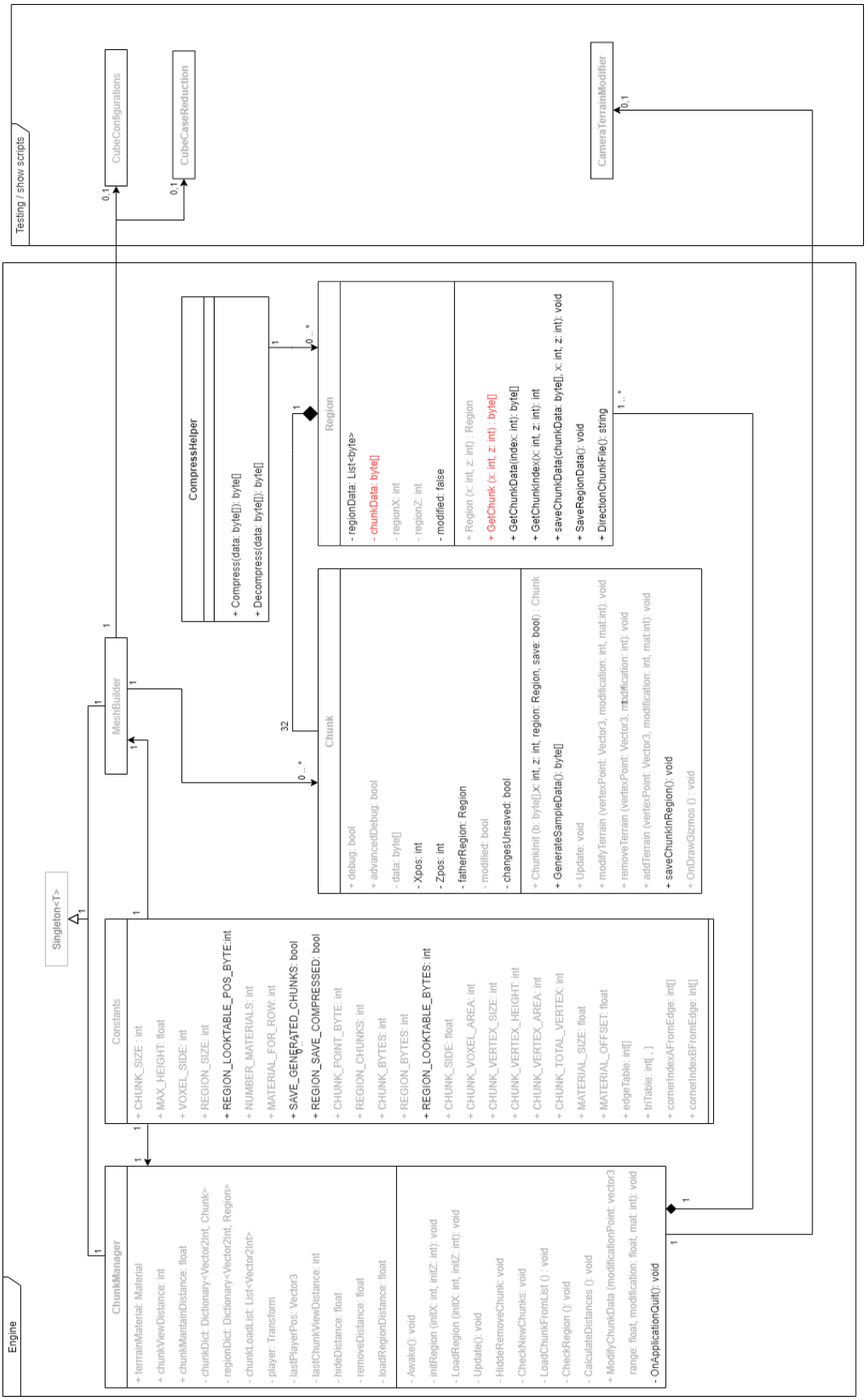

Figura 29 - Diagrama del Proyecto en el sprint id: 4, en gris los métodos y clases de sprint anteriores, en rojo los métodos o variables eliminados.

5.5.4. Resultados

La finalización del *sprint* con id 4 permitió crear terrenos persistentes, guardando los datos de los *regions* en ficheros con extensión .reg. Los resultados son difíciles de demostrar con una imagen, aun así, la Figura 30 muestra el resultado de cargar los ficheros visibles en el explorador de Windows. Este ejemplo se ha hecho sobre unos 8 *chunks* con la altura temporal de 40, que asignamos para mejorar la velocidad de creación de los *chunks*. El sistema de compresión ha reducido lo que serían aproximadamente 193KB a solo 15KB lo cual es bastante buena compresión (por lo que el tamaño de máximo de 145MB en un chunk de 256 de altura quedaría alrededor de 11,3 MB después de la compresión).

La prueba de aceptación era únicamente guardar un terreno y cargarlo al iniciar de nuevo Unity 3D, como se hace en la Figura 30. Aunque no se incluyera en un principio en la prueba de aceptación el sistema de compresión, sin la realización de este seguramente el sistema de guardado habría quedado incompleto, ya que no es admisible que los datos de los Marching Cubes llegase a ocupar un par de GB.

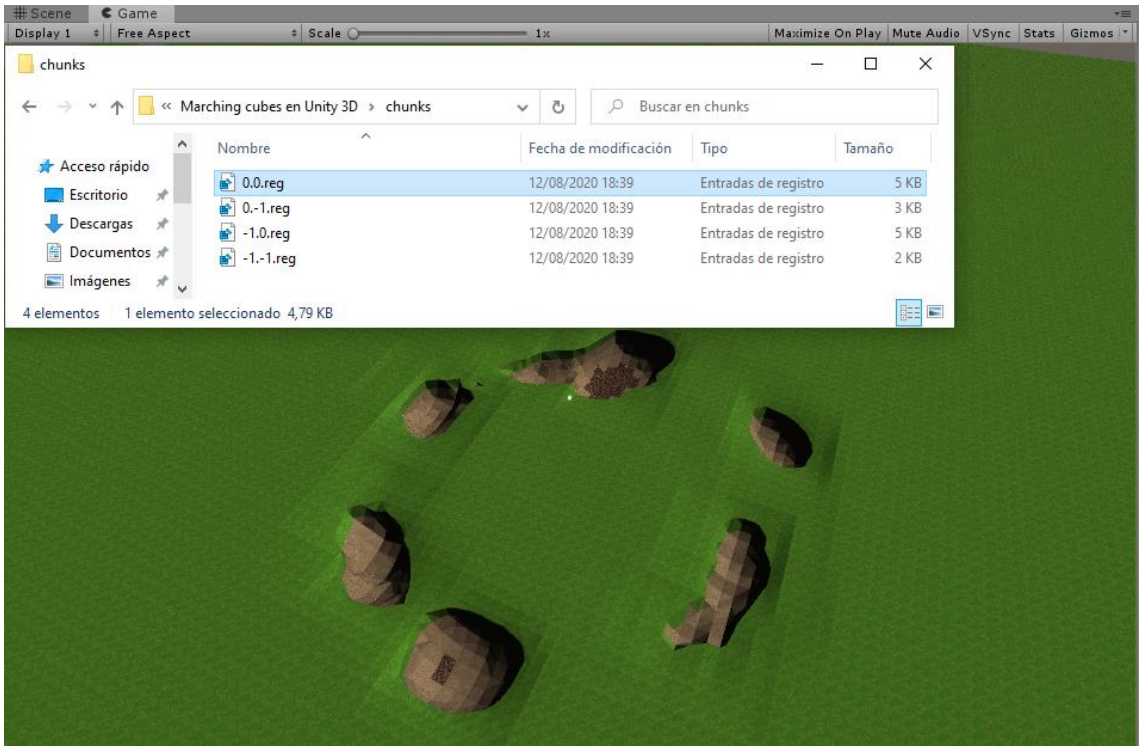

Figura 30 - Agujeros realizados en el terreno cargados desde ficheros .reg (Unity 3D) y sus respectivos ficheros (explorador windows).

5.6. Terreno aleatorio

El *sprint* se fundamentaba en desarrollar un terreno más orgánico mediante un sistema de ruido. Por lo que se pasaría de un terreno totalmente plano, usado para probar el sistema de edición y guardado en los anteriores *sprints*, a tener un terreno orgánico. Este *sprint* con id 5, permitió ya poder disfrutar de un terreno visualmente más agradable.

5.6.1. Tareas y pruebas de aceptación

En este *sprint* se buscaba desarrollar un terreno orgánico y acabar con el terreno plano arrastrado de los anteriores *sprints.* Por los conocimientos que se poseía se sabía que un sistema de ruido era la mejor forma de conseguir un terreno aleatorio y que a la vez pareciera algo real. Por lo que la única tarea que existía era la de desarrollar este sistema de ruido y usar este para desarrollar un terreno aleatorio e infinito (las técnicas usadas y otras más avanzadas están explicadas en [21]).

La prueba de aceptación se basaba en comprobar que este sistema de ruido permitiera generar un terreno aleatorio diferente al terreno plano actual, el cual debería presentar diferentes relieves y texturas a lo largo del mismo.

5.6.2. Desarrollo

El desarrollo de este *sprint* se dividió en cinco fases diferentes: El desarrollo del sistema de ruido, la creación del relieve, la texturización del terreno, la configuración de los parámetros y su integración al sistema de *chunks*. Estas cinco fases descritas permitieron la creación de un terreno montañoso que se generaba de forma pseudoaleatoria (se usa una semilla para su generación). Todo el sistema de ruido iría en un nuevo manager ("NoiseManager").

La primera parte, la del sistema de ruido, era la parte de la que se partía de cero y que después habría que integrar con los Marching Cubes. Esta parte se basaba en usar un *Perlin noise* [22] (un algoritmo que permite generar un gradiente pseudoaleatorio) para generar un conjunto de *floats* que representarían las alturas del relieve, de 0 a 1. Este *Perlin noise* se replicaría un par de veces con diferente peso generando un relieve más orgánico, el primer *Perlin noise* seria las montañas, el segundo los desniveles en el terreno y el tercero las irregularidades en el terreno (cada uno genera una modificación de menor altura en el terreno).

Partiendo del *Perlin noise* anterior, se implementó el sistema de relieve. Este consistía en coger el valor entre 0-1 del *float* del *Perlin noise* y multiplicarlo por la altura máxima del *chunk*, de este modo todos los vetices x,z en ese punto estarían contenidos en el terreno hasta la altura calculada. Aunque la implementación ya generaba un relieve de acuerdo con el *Perlin noise*, se

añadieron otras variables para facilitar la adaptación del terreno, como la altura máxima del relieve o la altura mínima de la superficie. Para realizar el *debug*, se crearon dos nuevas clases ("NoiseTerrainAutoUpdate" y "NoiseTerrainViewer"), que permitían generar un terreno y que ese se actualizara cuando los valores del inspector se modificaran, por lo que se visualizaban diferentes relieves modificando los datos del inspector. En este punto se decidió que el terreno generado sería un terreno montañoso, puesto que el sistema de Marching Cubes no soportaba fluidos, por lo que playas, océanos o ríos no podrían representarse en el terreno.

Teniendo el relieve generado, se comenzó el desarrollo de la texturización. Empezando por una idea muy básica como era texturizar en función de la altura del terreno, pero esta se descartó rápidamente, ya que eso generaba un terreno muy laminado con tres tonos diferentes. Descartando la idea anterior se pensó en realizar el texturizado basándose en el desnivel del relieve. Este tuvo una dificultad aumentada y se notó sobre todo a la hora de realizar la función de calcular el relieve, la cual necesitó de múltiples versiones diferentes hasta dar con la final. El principal inconveniente era la forma de calcular las diferencias de altura entre los puntos adyacentes de un vértice, donde se calculaban desniveles que no correspondían correctamente a lo deseado en ese punto. Además, otro punto que también incremento el tiempo necesitado en esta parte fue el de ampliar la anchura del ruido generado para que abarcase también los vértices adyacentes del *chunk*, lo que requirió de cambios en todas las funciones realizadas.

Las últimas dos fases fueron las que menos se tardaron en realizar, dado que solo era necesario cambiar un par de líneas en el "ChunkManager" y configurar los parámetros en el inspector. El desarrollo de las diferentes partes permitió dotar al sistema de *chunks* de un terreno infinito y aleatorio.

5.6.3. Diagrama del proyecto

Todo el trabajo recayó en el nuevo manager generado "NoiseManager", donde se implementaron todos los pasos del desarrollo (Figura 31). También se realizaron unos pequeños *scripts* ("NoiseTerrainAutoUpdate" y "NoiseTerrainViewer") que permitieron visualizar en tiempo real los cambios realizados a los parámetros del "NoiseManager", usado para realizar el *debug*. Hay otras clases modificadas como son el "ChunkManager" que añade el uso del "NoiseManager" para generar los datos de los *chunks* no existentes en los .reg. Esto último causó que se eliminara la función de la clase "Chunk" que creaba unos datos de testeo para el *chunk* (el terreno plano), porque estos datos se sustituían por el terreno generado en el "NoiseManager".

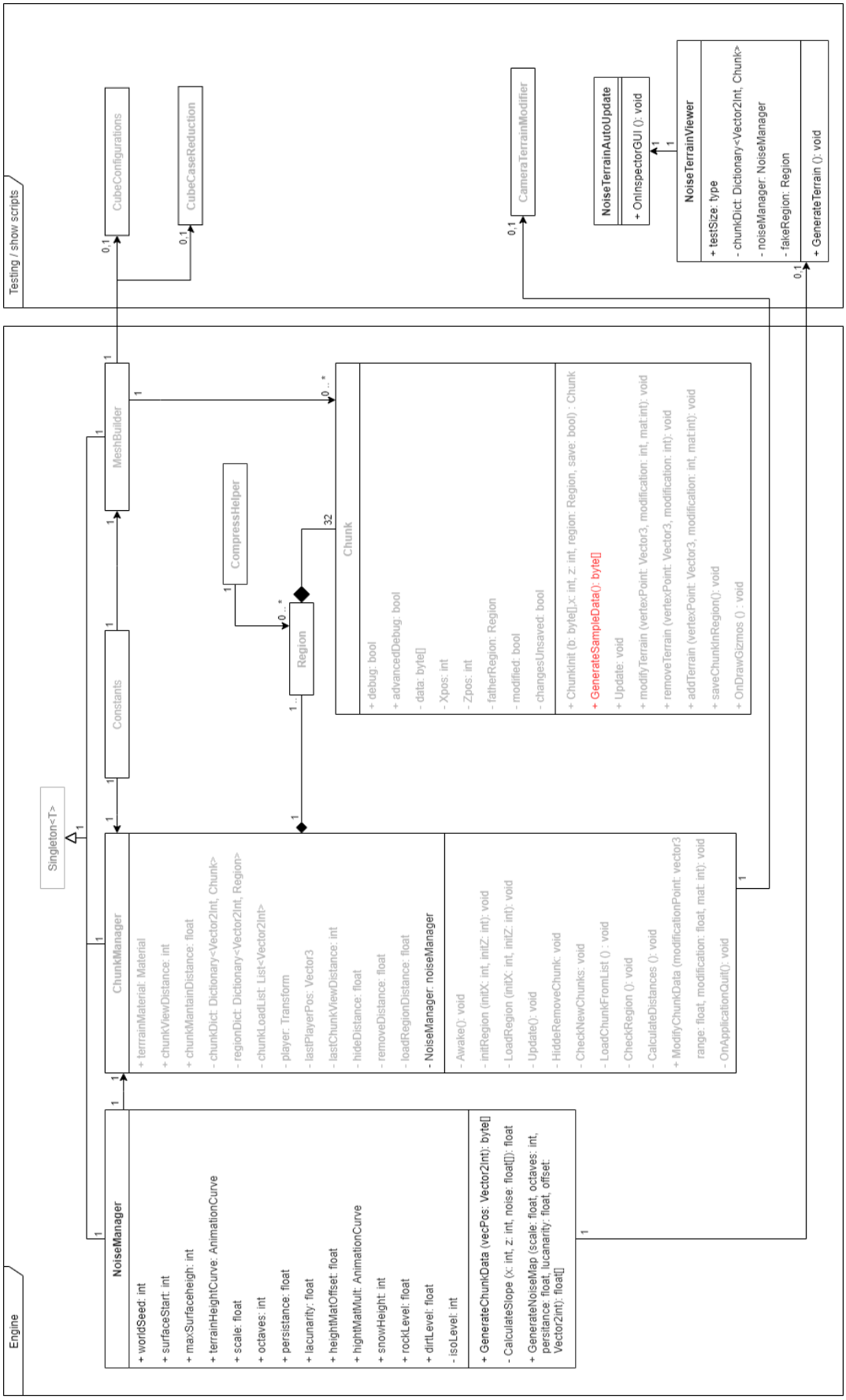

Figura 31 - Diagrama del Proyecto en el sprint id: 5, en gris los métodos y clases de sprint anteriores, en rojo los métodos eliminados.

5.6.4. Resultados

Este *sprint* trajo unos resultados muy visuales como se puede ver en las Figura 32 y Figura 33. A partir de ese momento el sistema de *chunks* contaría con un terreno infinito y configurable usando únicamente el inspector (ejemplo del inspector en la esquina superior izquierda de la Figura 32). Además, se creó un nivel para probar en tiempo real distintos parámetros en el "NoiseManager" y elegir así la configuración más deseada.

En cuanto a la única prueba de aceptación, quedó superada como se puede ver en las imágenes, ya que se generó un terreno diferente al plano, en este caso uno montañoso.

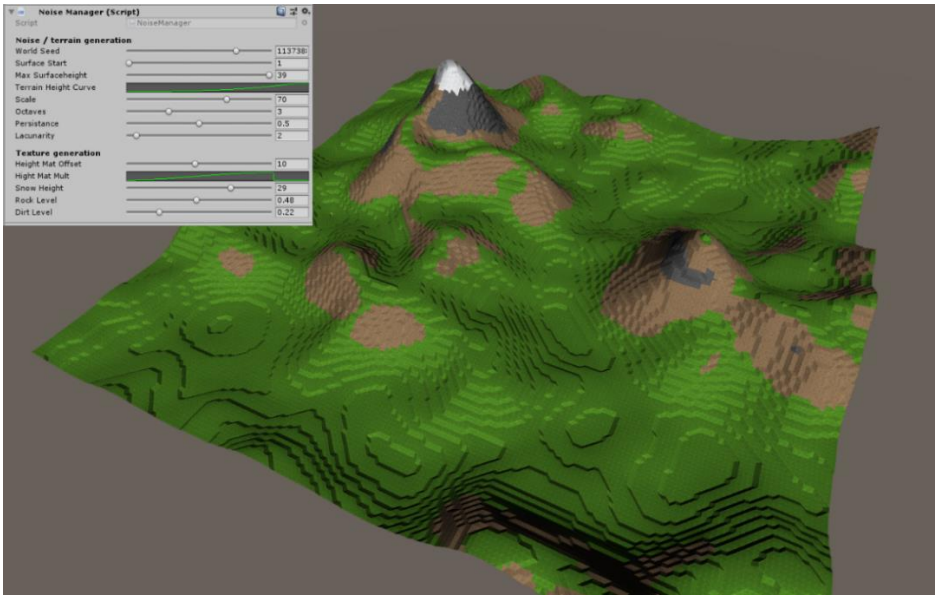

Figura 32 - Sección de terreno generada con el sistema de ruido y los parámetros usados.

Figura 33- Demostración de edición de terreno construyendo una "mina" en unas colinas.

5.7. Biomas

Este *sprint* fue diseñado para traer la variedad al sistema de terreno aleatorio, evitando que solo existiera un tipo de terreno. Este *sprint* con id 6 implemento los biomas, entendiendo estos como terrenos con distintas características. El desarrollo terminó la parte de terreno aleatorio en el proyecto, permitiendo generar un terreno capaz de ser montañoso, desértico, llano o gélido y que los cambios entre estos distintos terrenos no fueran bruscos.

5.7.1. Tareas y pruebas de aceptación

La tarea que se describió en el *sprint* se centraba alrededor de la realización de un sistema de biomas o diferentes tipos de ruido. Este sistema de biomas debía soportar distintos tipos de terreno y que los cambios de un terreno a otro no fueran bruscos desniveles o líneas muy demarcadas, estos cambios deberían ser orgánicos.

Al inicio del *sprint* se estableció que las pruebas de aceptación serían dos. La primera consistiría en desarrollar nuevos biomas, sin incluir el bioma montañoso generado en el anterior *sprint*. La segunda consistía en que estos biomas debían tener cambios entre si orgánicos.

5.7.2. Desarrollo

La primera parte del desarrollo consistió en extraer la generación de terreno y texturización del "NoiseManager" creado en el anterior *sprint*, ya que la generación del terreno montañoso se hacía dentro de esta clase. Por lo que se generaría una nueva clase llamada "Biome", que sería aquella de la cual heredarían todos los tipos de biomas diferentes. La clase "Biome" definiría las variables genéricas usadas para generar el relieve por todos los biomas y una función de tipo *abstact* (todos los hijos deben implementar la suya propia) que sería usada para la texturización de cada terreno en específico. Después de definir la clase "Biome" se creó un hijo de esta, "B_Mountains", la cual definiría sus propias variables usadas para la texturización del terreno e implementaría la función "GenerateChunkData" y "CalculateSlope", estas dos eran el código usado previamente en el "NoiseManager". Todo este proceso se centró en restructurar toda la arquitectura del sistema de ruido, por lo que la mayoría de nuevas clases extrajeron el código del "NoiseManager" y lo hicieron suyo, reduciendo el código de esta. Al terminar la restructuración la clase "NoiseManager" llamaba a la función "GenerateChunkData" de la clase "B_Mountains" que daba los mismos resultados que el inicio del *sprint*. En esta parte también se actualizó la parte del "NoiseTerrainViewer" que necesitó unas nuevas clases para auto actualizarse cuando los valores de un bioma fueran modificados.

El siguiente paso fue el de realizar otro bioma, que sería usado para la realización del sistema de combinar biomas. Este nuevo bioma fue un desierto, debido a que la diferencia de desnivel que se daría (montañas contra llanuras de las dunas) y de textura (el césped contra la arena) haría fácil ver si el sistema de combinar los biomas funcionaba orgánicamente.

Una vez que se tenían dos biomas se comienza a desarrollar el sistema de combinar biomas en el "NoiseManager" el cual se basaba en usar un sistema de ruido a gran escala. Este ruido variaría de 0 a 1 y en ese segmento se dividían todos los biomas. Por lo que se estimó que por debajo del 0.5 fuera desierto y por encima fuera montañas, siendo los valores cercanos al 0.5 donde estos dos biomas se unirían. Para que los biomas encajasen de forma orgánica se decidió establecer una altura de unión ("surfaceLevel") a la que la altura de los biomas se acercaría conforme se aproximaban al valor 0.5. Esto permitió que si el terreno generado en un bioma no estaba en una frontera se generara con una altura normal para ese bioma, pero que en las fronteras a la altura del bioma se acercara al "surfaceLevel" de forma gradual, evitando cortes o desniveles bruscos en el terreno. Para calcular la altura correcta al generar cada bioma se añadió un parámetro nuevo a la función de "GenerateChunkData" de la clase "Biome" correspondiente a la cercanía entre 0 y 1 a una frontera con otro bioma y mediante un *lerp* calcular la altura correspondiente.

Cuando se acabó la integración entre biomas, se realizaron dos nuevos biomas que permitirían cerrar el *sprint*. El bioma de las llanuras ("B_Plains") siendo en una llanura verde, muy parecida al bioma desértico, y el bioma glaciar ("B_ICE") que era totalmente nevado y que presentaba bloques de hielo en la superficie.

5.7.3. Diagrama del proyecto

Lo primero que se ve al observar la Figura 34, es la cantidad de variables eliminadas en el "NoiseManager" que pasaron a formar parte de las clases "Biome" o "B_Mountains". También se añadieron algunas funciones de ruido extra y variables relacionadas con la integración entre biomas. Otro punto importante es la clase "Biome" de la que heredan muchas variables del viejo "NoiseManager" y de la cual heredaran todos los diferentes tipos de biomas. Como curiosidad, "B_Mountains" es el único que usa pendientes para calcular las texturas en su terreno y por eso es el único que posee la función "CalculateSlope". Por último, indicar que el "NoiseTerrainViewer" también se adaptó por lo que necesitó unas clases nuevas, de muy pocas líneas cada una.

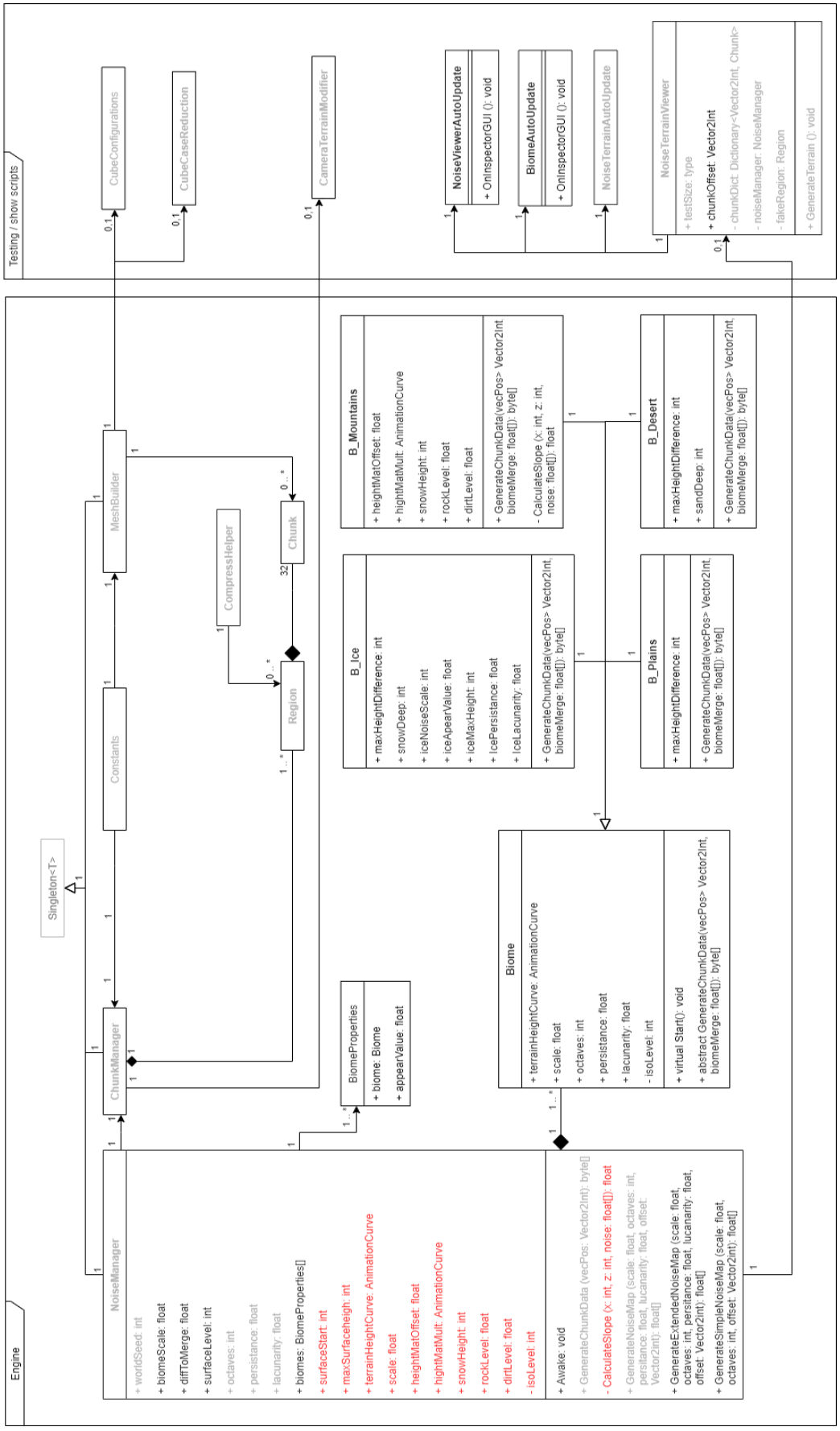

Figura 34 - Diagrama del Proyecto en el sprint id: 6, en gris los métodos y clases de sprint anteriores, en rojo los métodos o variables eliminados.

5.7.4. Resultados

Al final de este *sprint* se permitió la integración de diferentes biomas en el terreno aleatorio, lo que trajo variedad a este. El sistema de integración entre diferentes biomas se muestra en la Figura 35A, donde el bioma montañoso colinda con el desértico sin cortes en sus montañas y con una frontera orgánica entre ellos. También se crearon tres nuevos biomas: desértico (Figura 35A), glacial (Figura 36A) y llano (Figura 36B). Todo este nuevo sistema mantuvo la facilidad de edición a través del inspector de Unity 3D como en el *sprint* anterior (Figura 35B).

En lo referente a las pruebas de aceptación: la integración entre diferentes biomas y el desarrollo de los nuevos tres biomas fue superada como se puede apreciar en las figuras.

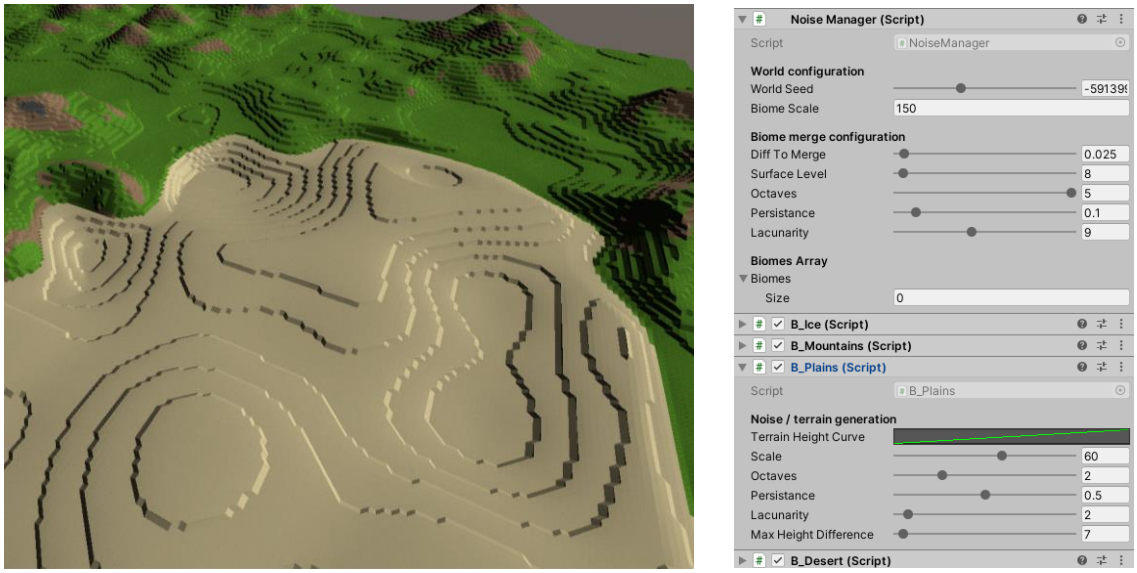

Figura 35 - Frontera biomas destico (B_Desert") y montañoso ("B_Mountains"). B) Inspector del NoiseManager, contiene los scripts de los biomas y el NoiseManager.

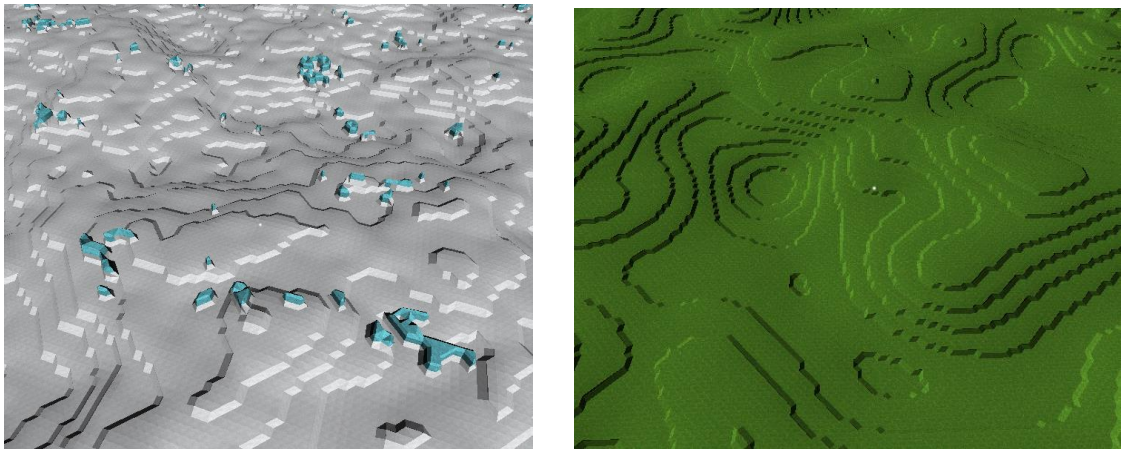

Figura 36 - A) Bioma glaciar ("B_Ice"). B) Bioma llanuras ("B_Plains").

5.8. Optimization Mediante Job System de Unity

El nombre dado al punto es diferente al que tenía el *sprint* con id 7 ("Fin del código y memoria"), ya que pese a tratarse en un principio de un *sprint* dedicado a pulir el código, acabo centrándose en la mejora de la eficiencia de la generación de *chunks*. Este *sprint* es el último que se usará para desarrollar el motor de Marching Cubes, por lo que el resultado final del motor se dio en este *sprint*.

5.8.1. Tareas y pruebas de aceptación

La tarea para la que se pensó este *sprint* era para pulir el código final, como podrían ser fallos o problemas de optimización. Como se indicó en el *sprint* con id 2 se dedicó a mejorar la eficiencia de la creación de *chunks*, que era inaceptable cuando se querían hacer *chunks* de un cierto tamaño.

La prueba de aceptación se centraba en aumentar a 256 la altura de los *chunks*, para que estos fueran costosos de generar y realizar cambios en el código para aumentar la eficiencia de generación de los mismos.

5.8.2. Desarrollo

La primera parte del desarrollo se centró en una pequeña investigación cuyo fin era incrementar la eficiencia del código en Unity 3D. Los dos métodos que destacaban para incrementar drásticamente el rendimiento en el código eran el uso de *Compute shaders* [23] o el Unity *Job System* [24], ambos permitían el uso de *multi-threading* para conseguir un incremento drástico. El *Compute shaders* se basaba en el uso de la tarjeta gráfica para realizar los cálculos, lo que incrementa la velocidad, y permitía *multi-threading*, pese a las ventajas era necesario reescribir el código para que pueda ser compilado como un *shader*. El *Job System* se basaba en la posibilidad de dividir trabajos ("Jobs") entre *threads* (procesos independientes al que se ejecuta el juego) diferentes al principal y su adaptación era más fácil en cuanto el código. En un principio se iba a elegir el *Compute shader* pero al descubrir un *package* de Unity conocido como "Burst" [25], que permitía compilar el *Job System* a código maquina ultra eficiente, se hizo un cambio de elección en el último momento. Finalmente se decidió realizar la mejora de la generación de *chunks* usando el *Job System* y *Burst*.

Una vez elegido el sistema que se iba a usar para incrementar el rendimiento, se ideó la forma de medir el resultado por lo que rodeó la función que generaba el *chunk* usando unos temporizadores. Se usó la clase "Stopwatch" [26] de .NET, que permitió registrar los milisegundos que costaba cada *chunk*. Se realizó el incremento la altura de los *chunks* a 256, como cuando se

detectó el problema de eficiencia en el *sprint* con id 2 y se comenzó la medición de la generación de *chunks* usado el código de los *sprint*s previos (cargando 111 *chunks*). El resultado visible en la Figura 37, muestra el *profiler* de Unity sobrecargado, superando los 60 ms en la carga (menos de 15 FPS). El problema radicaba en que se tardaba una media de 61,5 ms en cargar un *chunk*, dado que se carga un *chunk* por *frame* (sistema que se implementó en el *sprint* con id 2 para que no se congelara al cargar varios *chunks*). Como curiosidad se comprobó la eficiencia del código de texturización de flat shading, revelando que el rendimiento era similar. Se decidió que al final del *sprint* se limpiaría el código relacionado con realizar el *flat shading*, ya que mismos resultados se podían obtener usando la texturización normal y un *shader* específico que consigue *flat shading* (haciendo que la texturización *flat shading* fuera innecesaria).

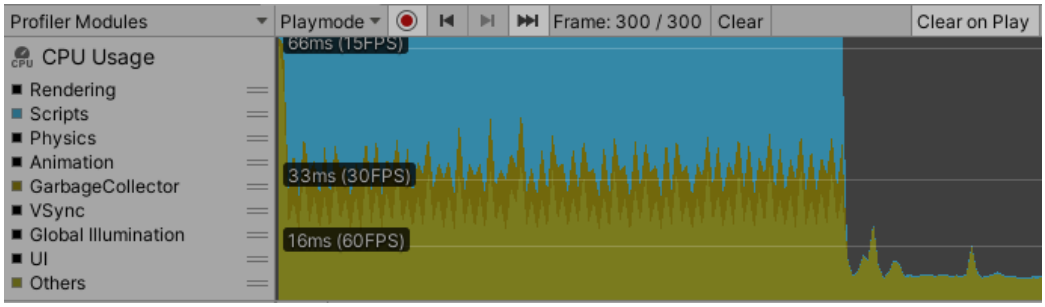

Figura 37 - Profiler usando el código single-thread de sprints previos. (61,5 ms/chunk)

Establecido el test inicial (cargar 111 *chunks*) y realizadas las mediciones del código original, comenzó la implementación del *Job System*. El código era bastante similar y el proceso consistió en realizar una copia de las funciones usadas en el "MeshBuilder" pero dentro de un *struct* que heredaba de *IJob* ("BuildChunkJob"). También se debió cambiar la función que se encargaba de ejecutar el código del construir el *chunk* ("BuildChunk(byte[] b)"), ya que el *IJob* debía tener una función ("Execute()") que era a la que se llamaba al iniciar el trabajo. Se realizaron cambios en los parámetros de las funciones, debido a que las variables se guardaban en el *struct* y eran accesibles por todas sus funciones. Por último se cambió el *input* y el *outpus* de "BuildChunkJob" este recibiría un *array* bytes[] y devolvería un *array* de tipo *Vector3*[] (posiciones vértices) y *Vector2*[] (posiciones del *uv map*), a causa de que usar la clase "Mesh" no estaba permitido dentro del *struct IJob*. Esto hacía posible realizar todos los cálculos de la generación del *chunk* usando el *Job System* y usar los vértices y los *uv map* generados para crear el *mesh* una vez se terminaba de realizar el proceso. Sin embargo, el resultado fue lo contrario a lo esperado, dando una media de 130,25 ms por *chunk* (Figura 38). La razón de estos malos resultados era que cargar solo un *chunk* por *frame*, en realidad lo único que se hacía era ejecutar el mismo proceso, pero esta vez fuera del *thread* principal, bajando la eficiencia.

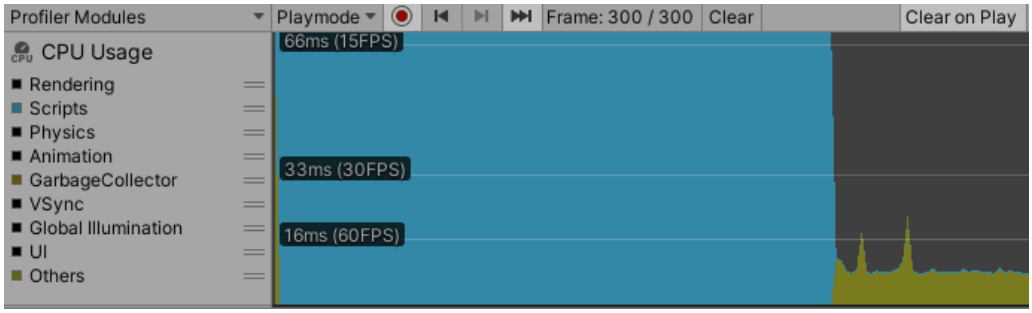

Figura 38 - Profiler usando únicamente el Job System de Unity 3D. (130,25 ms/chunk)

Pese a los resultados contradictorios de la implementación anterior se tenía confianza en el *Burst*, por lo que se decidió implementar este en busca de una mejora sustancial. El *Burst* constía principalmente en añadir un *flag* al *struct* "BuildChunkJob", que se encargaría de compilar este en código maquina ultra eficiente. Sin embargo, para poder compilar el código con *Burst* era necesario que este no usara clases propias de Unity como podría ser *Vector3*, *Vector2*, *Math* o *arrays* de C#. Usando "Unity.Mathematics" se sustituyeron los *Vector3*, *Vector2* por *float3* y *float2,* mientras que los *arrays* se sustituyeron por *NativeArray* en caso de los inputs y en el caso de los *outputs* por *NativeList* (ya que se desconocía el tamaño final de los *outputs* y los *nativesArrays* no se pueden cambiar de tamaño una vez se inicia el *job* de Unity). Además se necesitó añadir una copia de las tablas de triangulación en el *struct* "BuildChunkJob". Esto permitió compilar el *job* usando *Burst* consiguiendo alcanzar los 9,2 ms por *chunk*, haciendo que los FPS no bajaran de los 30 en todo el tiempo de carga (Figura 39).

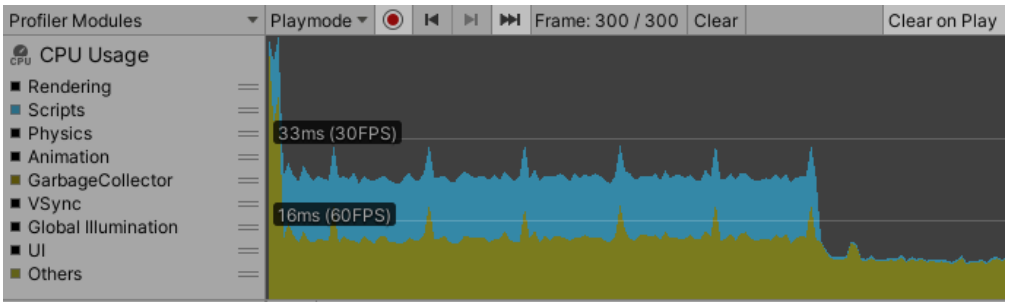

Figura 39 - Profiler usando el Job System (IJob) y el Burst. (9,2 ms/chunk)

Como última mejora se intentó pasar del *IJob* a un *IJobFor*, ya que este último permite ejecutar el trabajo en diferentes *threads* simultáneos, dividiéndolo en múltiples accesos a diferentes posiciones de un *array*. Para que se entienda mejor, permitía que se dividieran los cálculos del *chunk* en rangos de 15 vértices, ejecutando todos simultáneamente y generando el mismo resultado final. La implementación consistió exactamente en cambiar un par de líneas, ya que se sustituyó la "y" por un parámetro *int* "index" que sustituiría a la altura en los cálculos, el cual serviría para calcular los vértices del *mesh* de 15 en 15 de altura.

Esto contribuyo una mejora pequeña solo 7,5 ms por *chunk* (Figura 40). Como curiosidad si se desactivaba el *Burst* se conseguía 39 ms por *chunk*, una gran diferencia comparado con los 130,25 ms por *chunk* del *Ijob*. Seguramente la diferencia se incrementa con procesadores con más núcleos (el del ordenador de testeo tenía 4 núcleos, como se ve en la Figura 41 donde se ven los 3 *workers* trabajando, es decir el número de núcleos del procesador menos el usado como principal). En cuanto a la pequeña diferencia entre los *jobs* que usan *Burst*, quizás es debido a que el coste cae principalmente en extraer los datos del *job* y generar el *mesh*. Aun teniendo una pequeña mejora, el *multi-threading* y las *NativeList* puede dar problemas, por lo que se decidió escoger el *IJob* simple por encima del *IJobFor*.

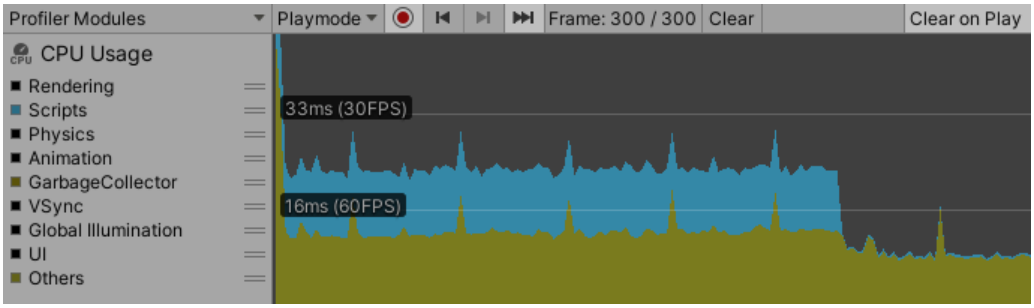

Figura 40 - Profiler usando el Job System con multi-thread (IJobFor) y el Burst. (7,5 ms/chunk)

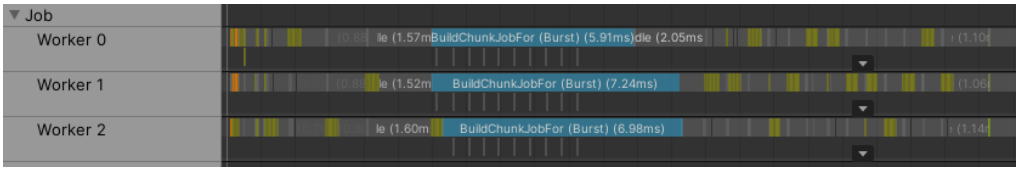

Figura 41 - Multiples workers trabajando cada parte del cálculo del chunk por separado.

Para terminar se realizó una limpieza del código, donde se guardó el código viejo del "MeshBuilder" en una región llamada "Original code (Deprecated)" para indicar que será borrado en futuras versiones (todavía algunos elementos de la demo lo usan, 15 posibles configuraciones del cubo y la reducción de casos). También se eliminaron los temporizadores dejando el código limpio y finalizando el *sprint*.

5.8.3. Diagrama del proyecto

Se añadió la clase "BuildChunkJob" al motor, muy similar a "MeshBuilder", debido a que hace lo mismo pero adaptada para ser usada por *Job System* y *Burst*. Se han eliminado algunos métodos y variables del "MeshBuilder" debido a la eliminación del código de texturización *flat shading*. En cuanto a la clase "BuildChunkJobFor", pese a ser nueva, está comprimida por ser exactamente igual que "BuildChunkJob" exceptuando a la función "Execute(index: int)".

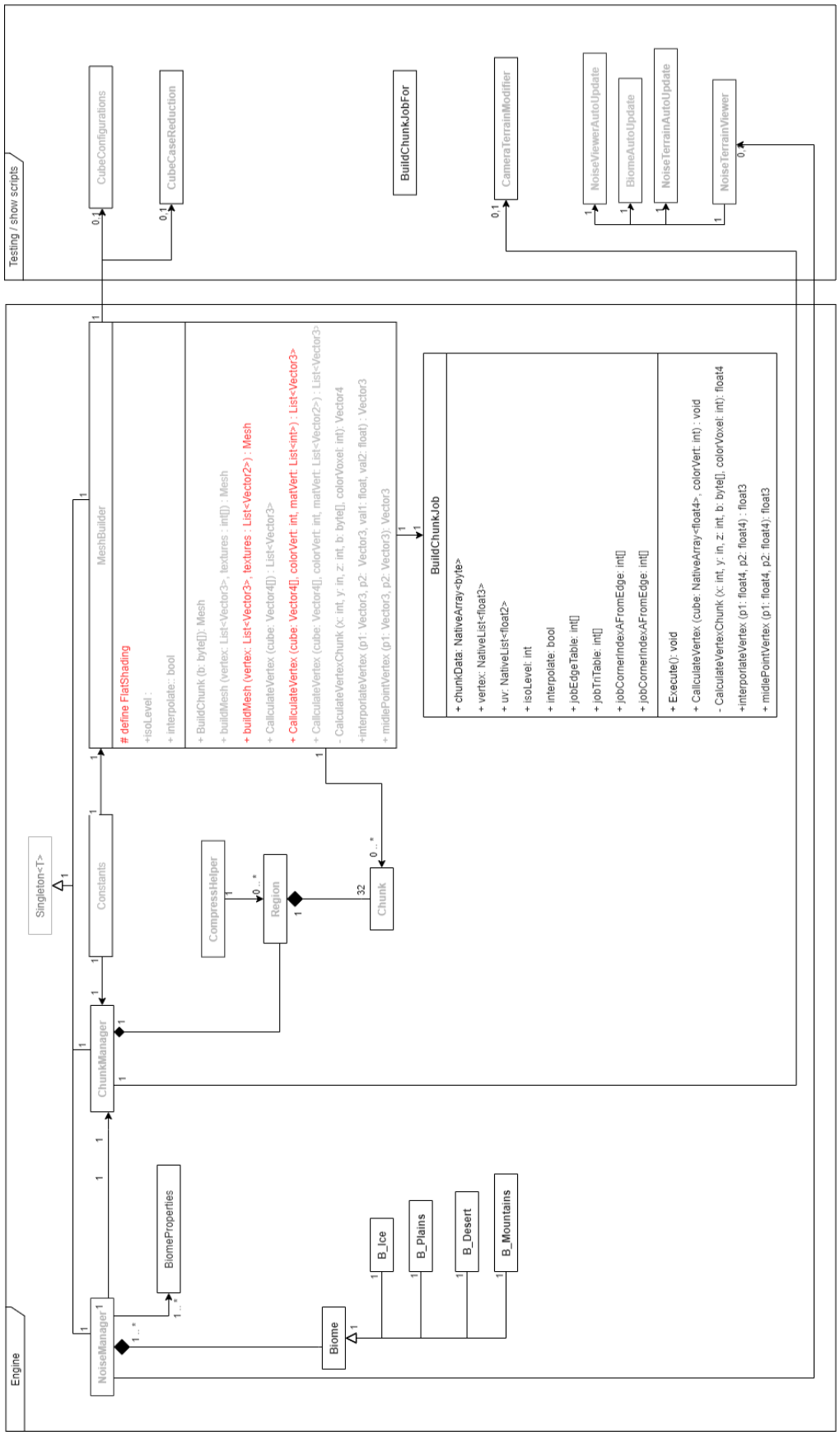

Figura 42 - Diagrama del Proyecto en el sprint id: 7, en gris los métodos y clases de sprint anteriores, en rojo los métodos o variables eliminados.

5.8.4. Resultados

Al final del *sprint* se consiguió reducir el coste de creación de *chunks* a solo un 14,96 % respecto el original, cerrando el desarrollo del motor. Durante el desarrollo se recopilaron los resultados de las implementaciones (Tabla 4). Este *sprint* también verifico la potencia del *Job System* de Unity sumado al *Burst* tanto en un único *thread* extra (*IJob*) como en *muli-thread* más real *(IJobFor)*. Aunque fuera evidente que se podía seguir mejorándose el rendimiento aplicando lo usado con los *chunks* al sistema de ruido o mejorando el código para trabajar de forma segura con IJobFor. El resultado mostrado en el *profiler* permitió demostrar que se alcanzó una eficiencia suficiente, al menos para el resultado esperado con el tiempo disponible.

La prueba de aceptación se definió de forma bastante vaga porque no se sabía que resultado se podría esperar ni como se aumentaría el rendimiento cuando se inició el *sprint*, pese a eso el resultado se consideró superado con mucho éxito.

Tipo	Original	IJob	IJobFor	IJob + burst	IJobFor+ burst
Máximos FPS	89 ms	158,2 ms	68 ms	34,2 ms	31 ms
Media FPS	70 ms	149.5 ms	58,5 ms	26,4 ms	23,4 ms
Media ms chunk	61,5 ms	130,25 ms	39 ms	9,2 ms	7,5 ms
Coste %	100%	211,78%	63,41%	14,96%	12,20%

Tabla 4 - Diferencia de eficiencia entre las implementaciones realizadas.

5.9. Preparar el Proyecto para la presentación

El *sprint* final (id: 8) se centró en preparar el proyecto para ser presentado, intentado mejorar la usabilidad de las demos y la estética de los *shaders*. Este *sprint* tiene la peculiaridad de que la estructura en la memoria es diferente al resto, debido a que no se modificó el motor (el motor se cerró en el anterior *sprint*). El punto de tareas y pruebas de aceptación no se realizó debido a que el *sprint* no tenía unas tareas fijadas en un principio y carecía de pruebas de aceptación centrada en algún punto como las anteriores (esta se basaba en destinar entre 16-20 horas al desarrollo de la utilidad y estética del proyecto), de igual manera sucedió con el punto del diagrama del motor que no se realizó, debido a que este no sufrió cambios.

5.9.1. Desarrollo

El *sprint* no tenía ninguna lista de tareas en un principio, pese a eso se sabía que se quería mejorar los *shaders* los cuales eran una parte principal de las demos. Esta mejora de los *shaders*

comenzó con el usado en el terreno, que no proyectaba ningún tipo de sombra y tenía iluminación en subsuelos. Por lo que se necesitó buscar un nuevo *shader* capaz de aplicar sombras, mediante el manual de Unity [27] y partes del código del anterior *shader* del terreno, que permitían darle una iluminación del tipo *flat shading* que estilizaban de gran manera el terreno. El nuevo *shader* permitió que las sombras se aplicaran al terreno, aunque debido a mantener la iluminación del tipo *flat shading* no se eliminó el problema de la iluminación en subsuelos, pese a eso se decidió terminar de trabajar este *shader*. Después de acabar el *shader* del terreno, se decidió implementar un nuevo *shader* que permitiera visualizar cada chunk de forma diferente. Este nuevo *shader* fue usado para generar una nueva escena ("ChunkVisualization"), que permitía visualizar el comportamiento del sistema de *chunks.* Para finalizar con la sección de los *shader* se realizó una mejora de los *shaders* de las escenas de "15Configuration scene" y "256 cases to 15 reduction" para que estos permitieran visualizar mejor la geometría de los *voxels* generados.

El siguiente paso que se pensó fue el de modificar la altura de los *chunks*, debido a que esta era innecesariamente grande. Por lo que se decidió poner la altura de 80, lo que permitía tener un terreno con una profundidad de excavación y una altura máxima aceptables. Esta modificación también requirió realizar unos cambios en las variables de los biomas, que requirieron de adaptar las variables relacionadas con la altura de los *chunks*.

Otra parte que quedaba pendiente y se desarrolló en este *sprint* fue el de la realización del ".readme" del repositorio de GitHub. Esto permitió poner en público el proyecto ya que disponía de una explicación básica de su funcionamiento [28].

Por último se realizaron un par de cambios en las escenas para una mejor usabilidad del usuario. Las escenas de "15Configuration scene" y "256 cases to 15 reduction" dispondrían de una interfaz indicando el tipo de *shader* que usaban para visualizar los voxels y la posibilidad de cambiar este por otro usando la rueda del ratón. Un proceso similar sufrieron las escenas "ChunkVisualization" y "FirstPersonLevel" que dispondrían de unos textos en su parte superior indicando el material seleccionado en ese momento y el tamaño del pincel de edición de terreno, así como la posibilidad de cambiar estas variables a través de unos *inputs*.

5.9.2. Resultados

Todos las nuevas implementaciones o modificaciones trajeron cambios visuales, aunque los más importantes son los que se dieron en las escenas "ChunkVisualization" y "FirstPersonLevel". En la escena "ChunkVisualization" se puede ver como cada *chunk* tiene un color diferente (Figura 43), lo que permite entender de forma visual cómo se comportan los *chunks*. En cuanto al nivel "FirstPersonLevel" la Figura 44 muestra una comparación del nuevo *shader* y el viejo, donde se

observa como el nuevo es capaz de proyectar sombras. Otro cambio pequeño pero importante fue el de la Figura 45 y Figura 46 que añadían esa pequeña interfaz destinada a mejorar la compresión de las escenas.

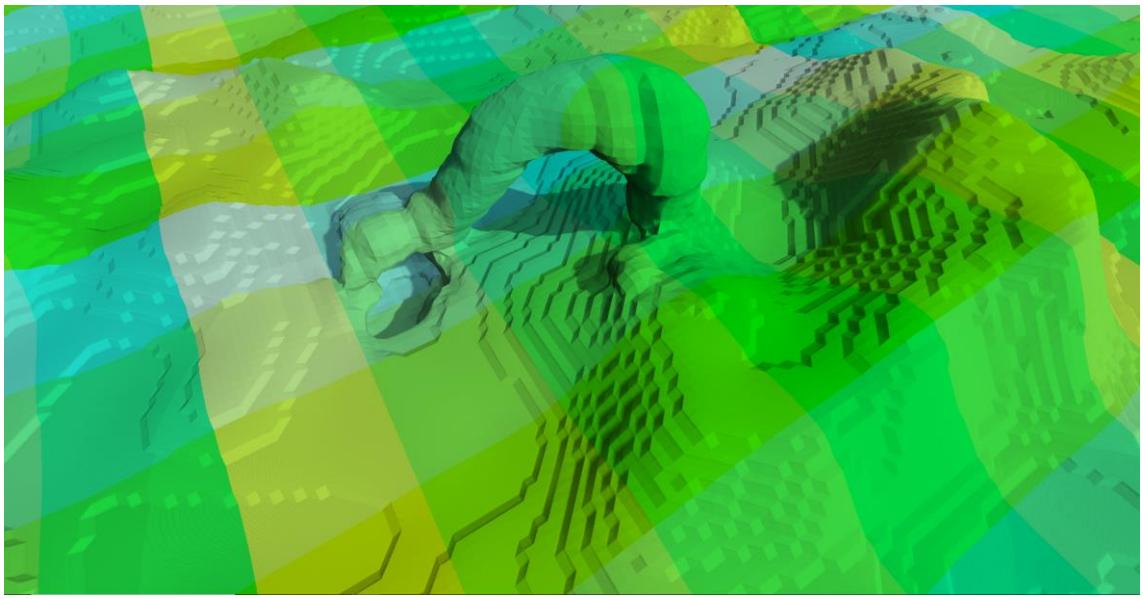

Figura 43 - Terreno de la escena "ChunkVisualization", donde cada chunk tiene un color diferente.

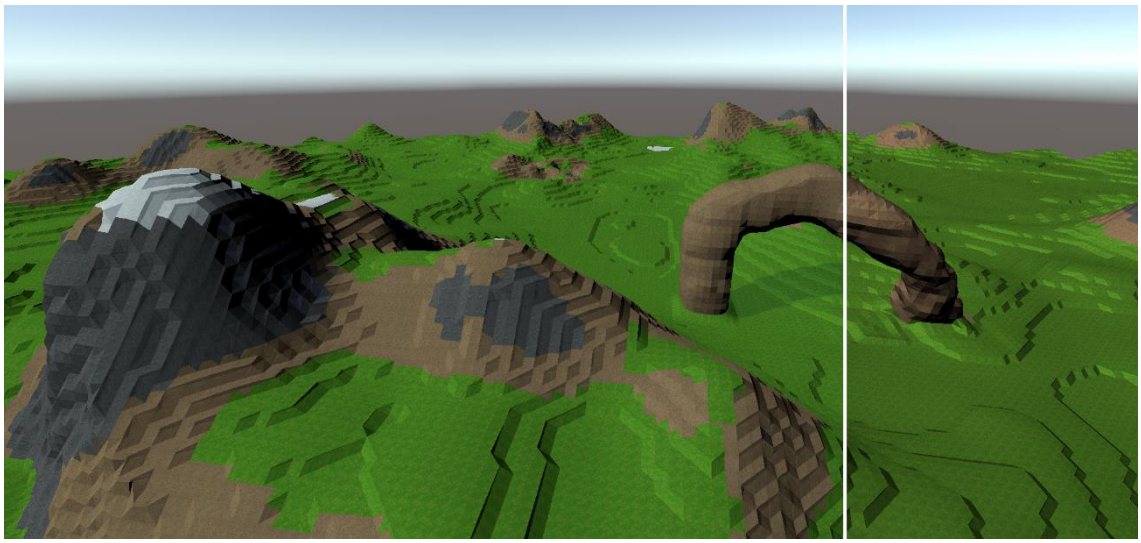

Figura 44 - Diferencia shader del terreno nuevo (izquierda) y viejo (derecha), donde se observa que el nuevo shader soporta la sombra generada por el arco del terreno.

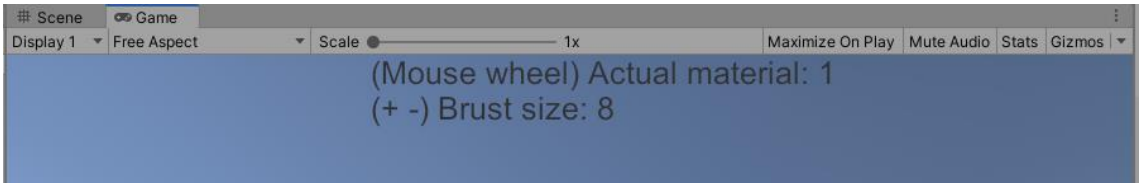

Figura 45 - Interfaz añadida a la escena "FirstPersonLevel", indicando parámetros de edición del terreno.

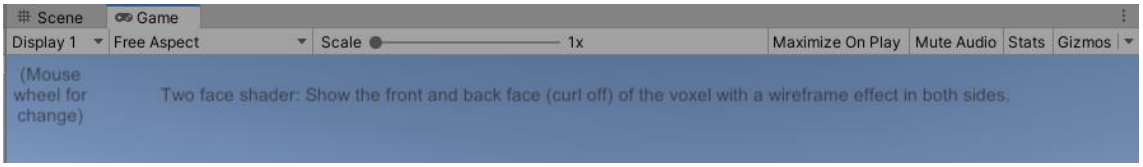

Figura 46 - Interfaz añadida a las escenas "15Configuration scene" y "256 cases to 15 reduction", que indican las características del shader usado.

6. Estudio económico

Esta sección agrupará todo lo relacionado con el importe económico del proyecto. Entendiendo como escenario el coste total en el que se hubiera incurrido en caso de ser desarrollado por una empresa, sin contar los gastos indirectos como alquiler de oficinas, suministros, impuestos, etc. El punto se va a centrar en los costes, sin contemplar el margen empresarial, ya que el producto final va a ser público y gratuito.

6.1. Coste recursos humanos

El tema más importante referido a costes cuando se desarrolla software es siempre el coste de recursos humanos, dado que es el que más peso tiene. Primero se han calculado las horas dedicadas al proyecto, disponibles en la Tabla 5. Se ha considerado oportuno descontar las horas que se dedicaron a la documentación (incluidas en los *sprints*), ya que en una empresa real u otro tipo de proyecto el documento habría sido más sencillo. Para calcular el coste de cada hora se ha utilizado la Tabla 6, donde se han recogido tanto el salario de un programador junior [29], como de un programador senior [30] y un desarrollador de software [31] (también denominado como analista de software). Estos costes salariales se han usado para calcular el gasto de personal en la Tabla 7, que extrae 15 horas del total destinadas al desarrollador de software (las equivalentes al diseño del motor), asigna el restante al desarrollador junior y añade 5 horas de programador senior (se entiende que equivalen a la necesidad de una segunda persona más profesional en las reuniones). Los datos finales están disponibles en la Tabla 7 dan un total de 2.965,25€.

Tarea	Horas de coste aproximadas
Reuniones realizadas	5
Organizar proyecto	12
Desarrollo de los *sprints*	330
Documentación	-144
Total	203

Tabla 5 - Tabla horas de trabajo del proyecto.

Se han tomado como referencia in total de 1750 horas anuales de media, según se desprende una amplia muestra de convenios colectivos [32]. Estas horas son usadas para el cálculo del salario / hora a partir del salario / año en la Tabla 6.

Empleo	Salario / año	Salario / hora	Coste empresa / hora (30% SS)
Programador junior	18.242€/m	10,42€/h	13,55€/h
Desarrollador de software	27.244€/m	15,57€/h	20,24€/h
Programador senior	30.757€/m	17,76€/h	22,85€/h

Tabla 6 - Salarios programador en España.

Tarea	Horas	Coste empresa/hora	Total
Desarrollo del proyecto	188 (203-15 de diseño del software)	13,55€/h	2.547,40€
Diseño del software	15	20,24€/h	303,60€
Coste extra de las reuniones	5	22,85€/h	114,25€
		Total	2.965,25€

Tabla 7 - Costes salariales del proyecto.

6.2. Coste software

Este es un pequeño inciso para repasar los costes de software subcontratado. Los programas usados han sido Trello, Github, Unity 3D y Visual Studio Comunity. Todos ellos tienen diferentes planes se suscripciones, pero se ha contratado en todos la versión gratuita. Dentro de estos hay que destacar que si la empresa fuera mayor que 5 programadores (5 personas usando Visual Studio), ya no se podría utilizar la versión de gratuita de Visual Studio. Algo similar pasaría con Unity 3D, que solo permite su versión gratuita si la empresa genera menos de 200.000$ al año. Se ha estimado que el proyecto sería realizado en una empresa de videojuegos pequeña de cuatro personas, por lo que el cote del software subcontratado ascendería a cero.

6.3. Coste hardware

Vamos a calcular el coste correspondiente al uso de un ordenador de gama media-alta. Para el cálculo del tiempo de uso de este, se ha considerado 217 horas correspondientes a un mes y

medio de uso a jornada completa. Para estimar el coste de amortización imputable a un mes y medio de uso (Tabla 8).

Hardware	Coste de compra	Tiempo de vida	Coste de un mes y medio de uso
Ordenador gama media-alta y periféricos	850€	5 años	23,5€

Tabla 8 - Costes del proyecto relacionados con el hardware

6.4. Costes totales

Los costes totales están basados en la suma de los costes previos, resumido en la Tabla 9. Como era de esperar el mayor importe corresponde a los recursos humanos. El resultado final no incluye los costes indirectos (alquiler oficinas, internet, etc), debido a la dificultad e inexactitud que presentarían estos, ya que variarían dependiendo de la estructura de la empresa. Al final los costes del proyecto sin incluir la documentación ni los costes indirectos sumarian un total de 2.988,75€.

Desde un principio se pensó en que sería un producto gratuito, pero si hubiera tenido un enfoque diferente podría intentarse venderse como un motor de Marching Cubes por un precio más barato que los actuales (aunque esto requería terminar de finalizar el motor, ya que el estado actual no representa el motor terminado en su totalidad). Pero tanto se intentará vender el motor como ofrecer gratuitamente, una parte con mucho valor son los conocimientos adquiridos, que permiten desarrollar juegos con el uso de Marching Cubes, los cuales son un nicho donde destacar no es difícil debido a su reducida cantidad y su popularidad relativamente alta (por ejemplo el "Astroneers" tiene entre de 1-2 millones de ventas [33], siendo de los últimos en salir con este tipo de terreno).

Tipo de coste	Coste monetario
Recursos humanos	2.965,25€
Software	0€
Hardware	23,5€
Total	2.988,75€

Tabla 9 - Costes totales del proyecto.

7. Resultados

A lo largo de este punto se tratarán si los objetivos han sido cumplidos, las propiedades que ha llegado a alcanzar el proyecto, el estado final del diagrama del motor y unas imágenes de los resultados alcanzados. Se añadió un punto extra dedicado a hablar de las diferencias que se han dado entre la organización esperada en un inicio del proyecto y la llevada en la realidad, analizando las horas destinadas a cada parte.

7.1. Objetivos completados

En este punto se tratarán los objetivos del proyecto y si estos han sido alcanzados correctamente:

1. **Estudio del algoritmo Marching Cubes, así como las evoluciones dadas hasta su actualidad y los videojuegos que lo han implementado:** Este objetivo ha sido completado en su totalidad, debido a que se ha podido realizar el motor de Marching Cube, encontrar videojuegos que lo han implementado, así como conocer las evoluciones del algoritmo.
2. **Implementar el algoritmo de Marching Cubes en Unity 3D, con un ejemplo sencillo, una representación de las configuraciones del *voxel*:** El ejemplo se realizó en el *sprint* con id 0, sin embargo no fue hasta el *sprint* con id 2 cuando se alcanzó completamente el objetivo.
3. **Usar la implementación de este algoritmo para crear un terreno sencillo que permita la implementación de un sistema de *chunks*:** El terreno simple y el sistema de chunks se finalizaron en el *sprint* con id 2, dejando por finalizado este objetivo.
4. **Implementar un sistema de edición de terreno que sumado a lo anterior permita tener un terreno editable en su totalidad:** El terreno se podría editar en su totalidad una vez acabado el *sprint* con id 5, que dejaba terminado el objetivo, aun así se mejoró el sistema de edición con el *sprint* con id 6 que permitía guardar el terreno editado (Dejando superado el objetivo por encima de lo esperado en un principio).
5. **Implementar un sistema que permita generar terreno aleatorio usando todo lo conseguido en los puntos anteriores:** El terreno aleatorio y el sistema de biomas se realizó en los *sprint* 6 y 7, generando aleatorio que presentaría diferentes tipos de biomas a lo largo de este. El objetivo se alcanzó correctamente, aunque técnicas más avanzadas se podrían añadirse en el futuro (como las disponibles en [21]).

7.2. Propiedades finales del proyecto

Si analizamos el proyecto acorde a los *sprints* desarrollados el estado final del motor de Marching Cubes ha ido adquirido las siguientes propiedades:

- Marching cubes: El punto más básico para nuestro motor, siendo capaz de realizar los cálculos necesarios para generar mallas usando el algoritmo de los Marching Cubes. Su uso permite tener mallas 3D totalmente editables y como es en el caso del proyecto un terreno editable en su totalidad.
- Uso de *Job System* y *Burst*: Esta cualidad es de destacar ya que su uso agregó gran cantidad de eficiencia, algo que siempre es solicitado por todos los desarrolladores cuando descargan un *asset* y que puede ser uno de los puntos más problemáticos del algoritmo Marching Cubes.
- Terreno aleatorio y biomas: Estas dos propiedades son capaces de generar un terreno totalmente aleatorio y que varía a lo largo de este, creando mundos diferentes en cada partida. También hay que incluir que los biomas tienen su propio nivel para ser editados y son fácilmente configurables usando el inspector de Unity (una forma de mejorar la velocidad de integración de estos en el juego del desarrollador).
- Sistema de *chunks*: Esta propiedad es obligatoria siempre que se desea hacer terrenos de enorme tamaño y se desea eliminar las pantallas de carga. En el caso del proyecto se usa para cargar todo el terreno aleatorio alrededor del jugador sin que este tenga que esperar pantallas de carga. En el caso de no querer un terreno infinito se podría usar para mejorar la eficiencia del terreno finito.
- Sistema de guardado: Una cualidad que se busca en muchos juegos es la persistencia de las acciones del jugador, siendo esta propiedad la que hace que cualquier cambio realizado en el terreno se quede registrado de forma permanente. Asimismo, el sistema de guardado cuenta con un sistema de compresión permitiendo guardar grandes cantidades de datos de terreno en el menor espacio posible.

Este conjunto de propiedades permite tener un motor de Marching Cubes lo suficientemente desarrollado para generar terrenos visualmente atractivos y demostrar el funcionamiento de la edición de terreno, permitiendo que estos puedan ser implementados por algún desarrollador en algún juego sencillo o que sirva como base para extender su propio motor de Marching Cubes.

7.3. Diagrama final

El resultado obtenido al final del *sprint* con id 7 permitió terminar el desarrollo de la parte de código relacionada con todo el motor de Marching Cubes, dando como resultado la Figura 47. En la que se muestran en profundidad los managers y el "BuildChunkJob" (teniendo una importancia similar al "MeshBuilder", ya que se encarga de construir los *chunks*). Este diagrama final permite hacer una idea de cómo interaccionan entre si los diferentes elementos, las herencias de clases existentes o las proporciones de cada clase.

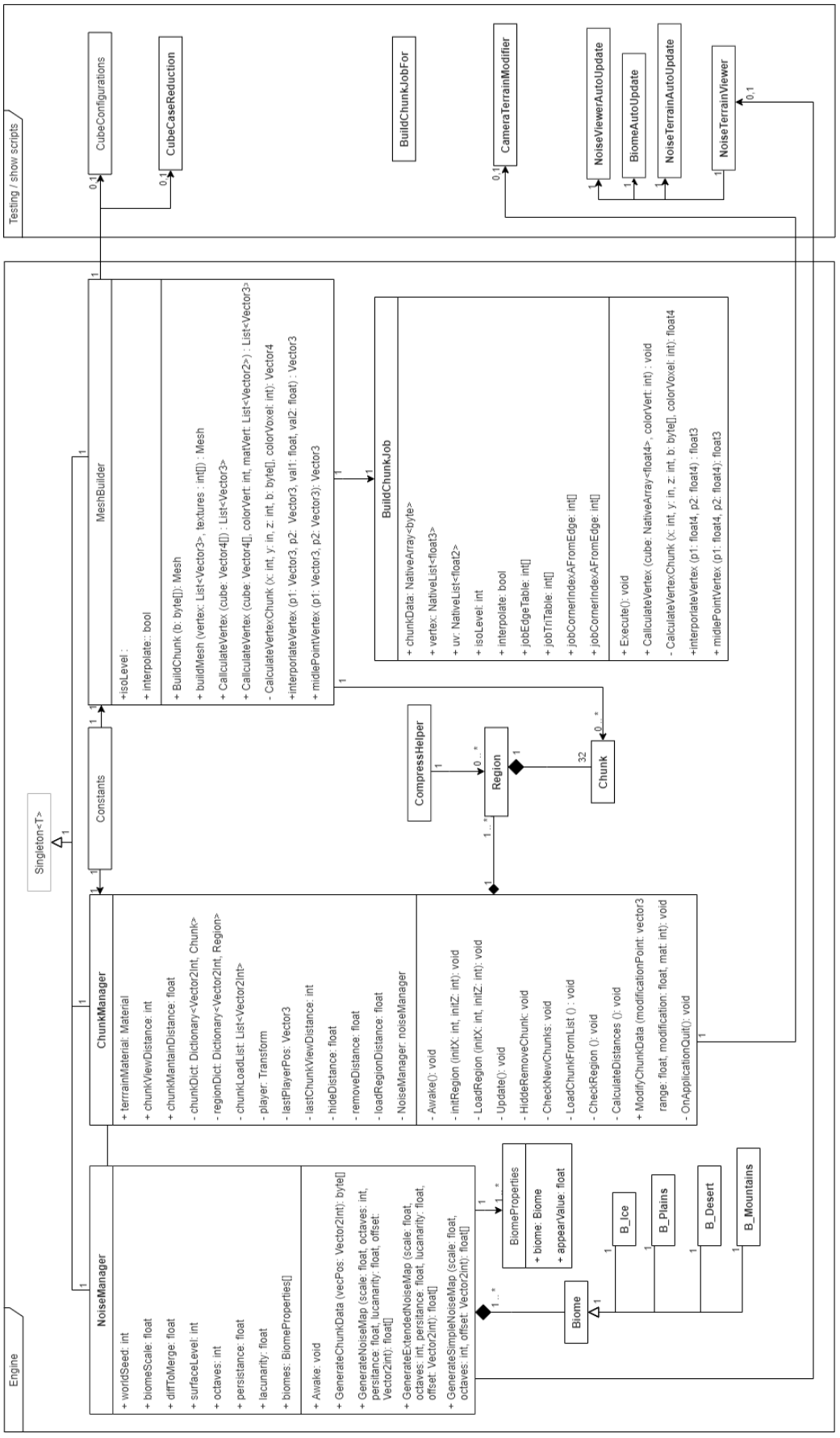

Figura 47 - Diagrama final del proyecto.

7.4. Evolución del proyecto en imágenes

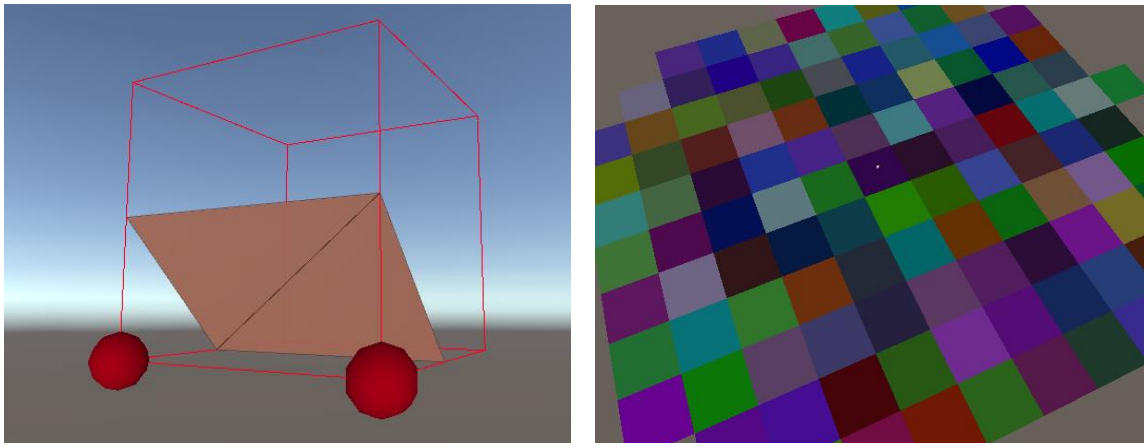

Figura 48 - A) Sprint 1/9, se puede generar la malla de un único voxel. B) Sprint 3/9, se genera un terreno plano que se carga y descarga mediante un sistema de chunks.

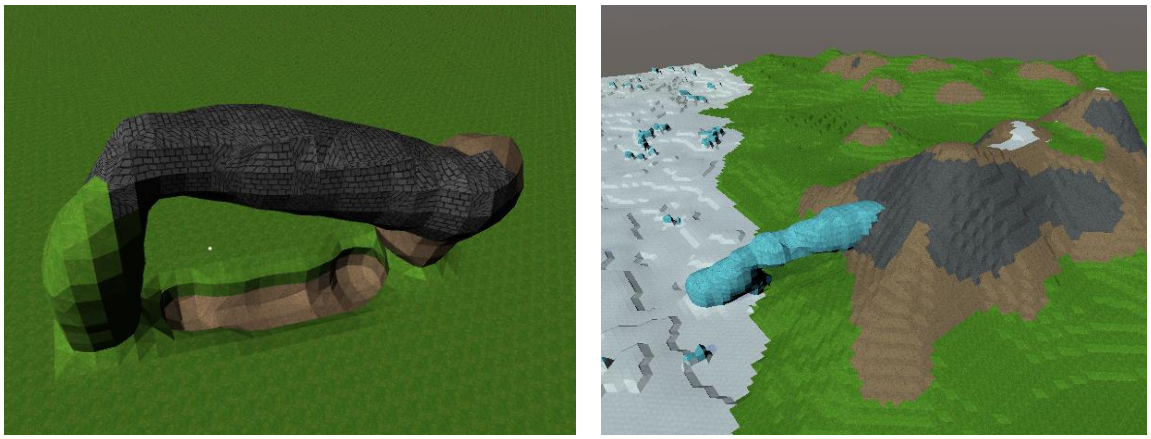

Figura 49 - A) Sprint 5/9, el terreno plano es editable, soporta texturas y sus cambios son permanentes gracias un sistema de guardado. B) Sprint 8/9, motor terminado, el terreno editable ahora presenta un relieve aleatorio y biomas con diferentes características.

7.5. Desviación respecto la planificación inicial

El punto tratara de la diferencia que se ha dado con la organización inicial, así como las causas de estas diferencias. Se remarcarán los *sprints* que han fallado en su estimación y recalcara los puntos que deberían haberse tenido en cuenta para futuros trabajos.

El proyecto previo al desarrollo y los *sprints* tuvo una fase de organización que ocupo 12 horas y que terminó cuando se estimó la duración de todos los *sprint*. La estimación presentaba la forma de la Figura 50A, donde los tres primeros *sprint* ocupaban 45 horas, pensando que los primeros *sprints* que partirían de la nada costarían un extra, y el resto costarían 30 horas. Esto nos dejaba con un total de 315 horas de *sprints*, donde se pensaba que solo que alrededor de 80 horas (25%) serian destinadas a la documentación (Figura 50B).

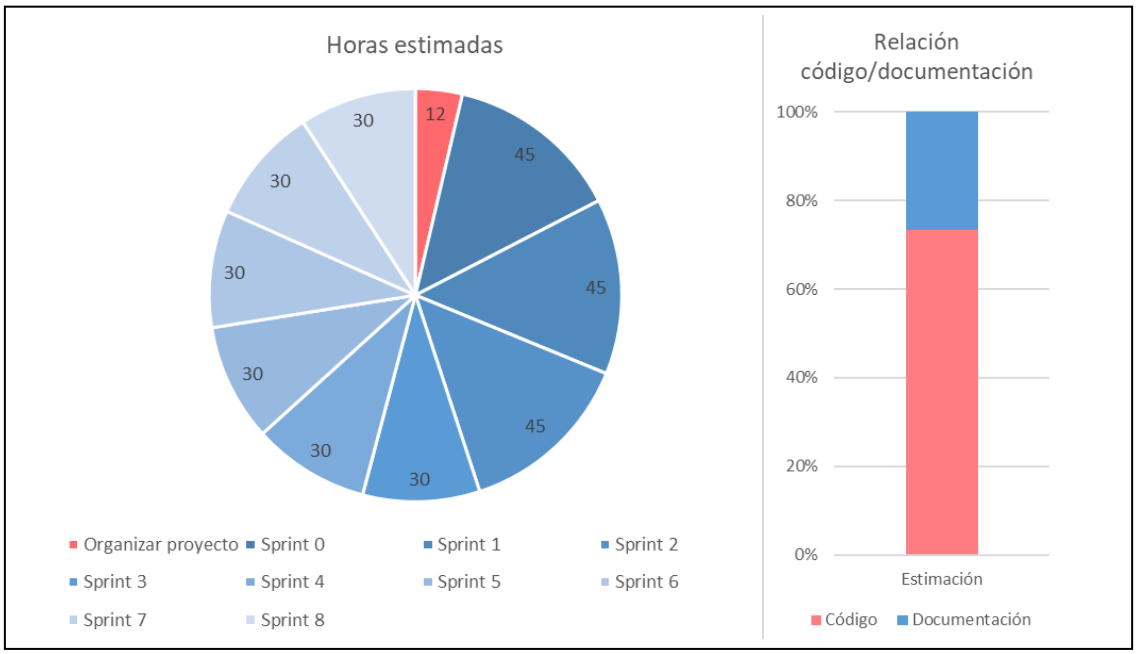

Figura 50 - A) Horas estimadas de cada parte del proyecto. B) Relación código/documentación estimado.

Pero si se observa la realidad Figura 51A, las horas estimadas suelen rondar alrededor de las 30 en la mayoría de *sprints*, exceptuando algunos que la superan por bastante. En cuanto a las horas que se esperaba dedicar a la documentación, se alcanzaron las 144 horas (41%) frente las 80 estimadas, debido en su mayoría a horas gastadas en corregir la memoria (algo que no se tuvo presente a la hora de organizar la duración de los *sprints*). Entre las diferencias con lo estimado (Figura 52) cabe destacar los siguientes *sprints*:

- *Sprint* 0: Durando alrededor de la mitad de lo esperado, principalmente porque no se realizó documentación, sumado a que realizar el algoritmo de los Marching Cubes costó menos del tiempo esperado.
- *Sprint* 1: Pese a que su duración sea aproximada a la estimada, se destaca por ser el más largo de todos y esto es debido a que en él se desarrolló el estado del arte, el cual requirió de mucho trabajo y pulido hasta su versión final (añadiendo el coste de las modificaciones al *sprint*), aumentando de gran manera el coste de este.
- *Sprint* 2: Durando la media de 30 horas compartida por los otros *sprints*, indica el coste que se le debería haber estimado en un principio (no necesitaba 45 horas).
- *Sprint* 3: Superó en medida lo esperado debido a que durante el desarrollo del código se encontraron complicaciones y *bugs* que aumentaron el tiempo necesario.
- *Sprint* 5: La razón para su duración extendida es debido a una mala valoración del peso de los puntos de resultados y conclusión, los cuales llevaron bastante más de lo esperado.

Como se puede observar en la Figura 52 el resto de *sprints*, que no se mencionan en la lista anterior, tuvieron una duración correcta o aproximada a la organización inicial.

La finalización del proyecto ha dejado visibles los principales puntos que se omitieron en la organización inicial como son: las reuniones conllevarían horas, las horas estimadas de los puntos de la memoria se debería hacer en función de la dificultad (preguntando al tutor, por ejemplo) y no solo por la extensión de los puntos, así como contar con que los puntos tienes que revisarse además de redactarse, lo que debería tenerse en cuenta en las estimaciones iniciales.

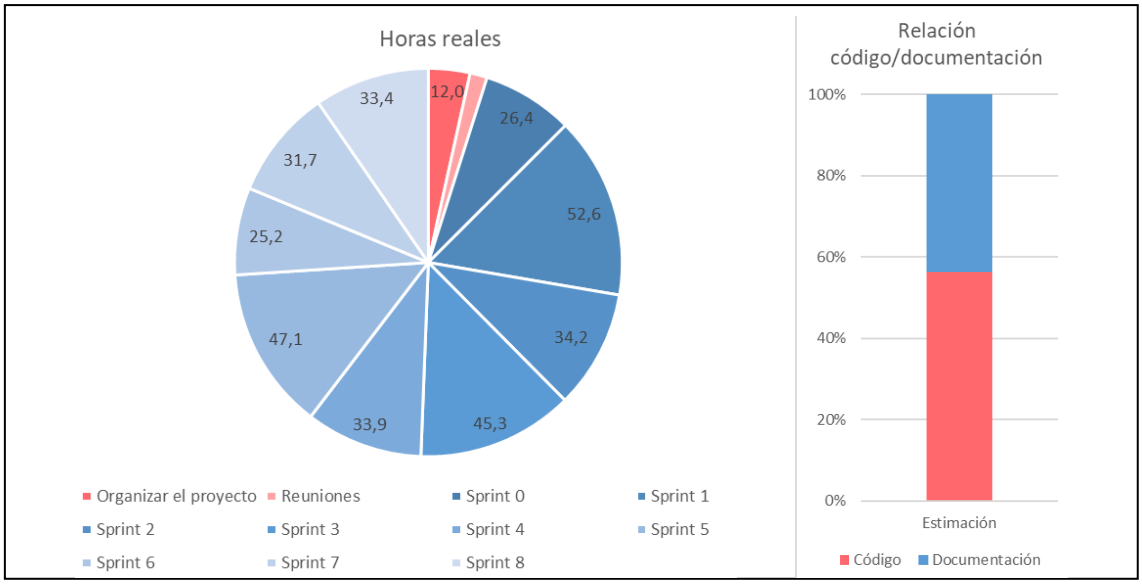

Figura 51 - A) Horas reales de cada parte del proyecto. B) Relación código/documentación real.

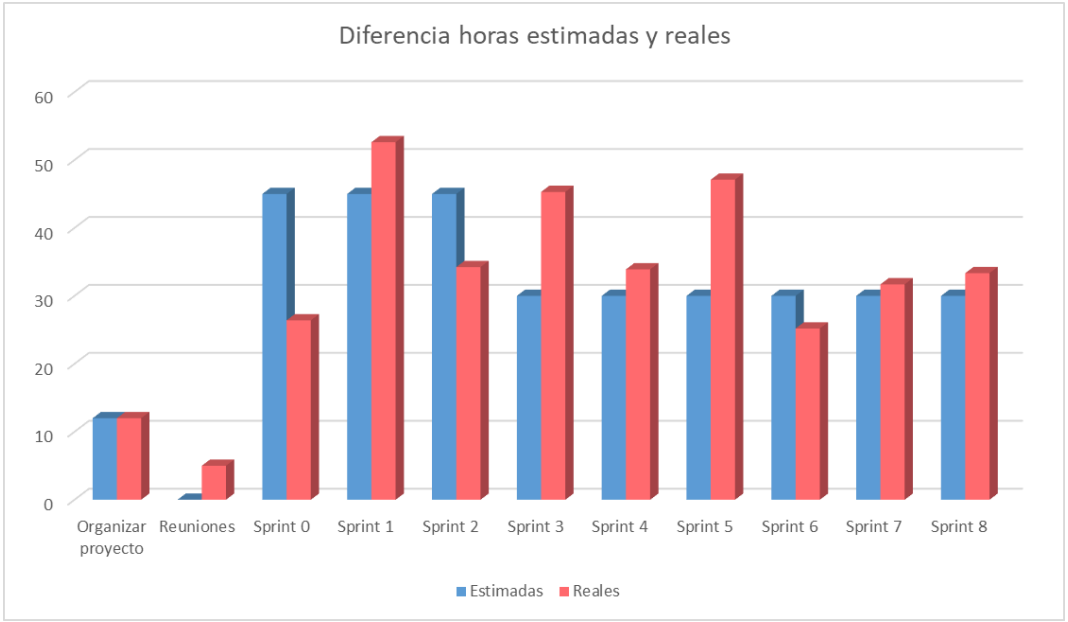

Figura 52 - Diferencia entre las horas estimadas y las reales.

8. Conclusión

El proyecto surgió bajo el interés de realizar un motor de Marching Cubes, originado por la privacidad que se tratan los motores realizados para juegos con características propias de los Marching Cubes y debido a que no existía ningún *asset* gratuito que estuviera la altura de las versiones de pago del algoritmo. Esto generó una propuesta de proyecto muy técnica orientada a realizar una implementación del algoritmo Marching Cubes en Unity 3D. Sin embargo, detrás de la propuesta técnica que se ve claramente en los objetivos, existía además una búsqueda personal y era la de conseguir desarrollar un motor de Marching Cubes que fuera fácil de usar y que ayudara a utilizar este tipo de mecánicas o terrenos en juegos *indies*.

En la parte más técnica se recogían las propiedades que debía alcanzar el motor de Marching Cubes. Desde esta visión técnica se extrae la conclusión de que el proyecto ha sido exitoso, ha tenido todos los objetivos cumplidos y presentables en una demo. Como en todos los proyectos en desarrollo, alguna de las propiedades del motor pueda extenderse en el futuro, el contenido que se esperaba de cada una ha sido el óptimo, gracias a que las propiedades interactúan bien entre ellas y no dan la sensación de que falte contenido alguno.

En referencia a la idea personal de ofrecer una herramienta a los desarrolladores *indie*, no se contempló en los objetivos, porque como se ha indicado anteriormente, el proyecto se orientó de una mantera técnica. Esta idea es susceptible de ser mejorada, ya que se buscaba ofrecer un motor de Marching Cubes para uso profesional, necesitando este de un trabajo extra para conseguir la flexibilidad y las facilidades de uso que se esperan de un *asset* de ese tipo de calidad. Esto no significa que el motor no haya conseguido su objetivo, ya que este puede ser el embrión sobre la que otros desarrolladores *indie* realicen su propio motor de Marching Cubes o un juego sencillo. Igualmente, al ser un proyecto ambicioso y ampliable a cada instante, el espacio temporal en el que se realizó ha implicado que el resultado se adapte a la realidad del producto final.

Estas dos reflexiones dejan claro que se ha conseguido implementar de forma exitosa el algoritmo Marching Cubes, siendo el resultado el óptimo al tiempo empleado. Habiendo generado un producto que puede ser ampliado en el futuro o utilizado por desarrolladores que deseen realizar su propia adaptación del motor, aportándoles una ayuda base como se planteó desde un principio.

8.1. Futura ampliación del proyecto

Pensando en las futuras ampliaciones del motor, existirían dos variantes: una destinada a facilitar o flexibilizar su uso por terceros y otra más orientada a implementar nuevas propiedades.

Implementaciones destinadas a la facilidad y flexibilidad de uso:

- Mejorar el *.readme* o realizar una wiki, debido a que es fundamental tener una buena documentación para facilitar su uso y la actual es bastante básica.
- Limpiar el proyecto de elementos que no se han eliminado debido a que son contenido potencial en la presentación de este y que agregan valor al TFG, pero son incensarios desde la visión de un *asset*.
- Añadir "pinceles extra", entiendo estos como formas de editar el terreno de forma diferente, aportando un extra de flexibilidad.
- Arreglar el fallo de texturización para texturas geométricas, ya que este fallo hace que no se puedan usar ese tipo de texturas.
- Añadir soporte para terrenos finitos para caso de que no se desee un terreno aleatorio e infinito (implementación actual). Lo que aportaría una forma rápida de conseguir terrenos finitos sin necesidad de que sea necesaria una modificación en el código de propio.

Mejora del algoritmo y añadir nuevas propiedades:

- Realizar el sistema de ruido mediante *Job System* y *Burst*, reduciendo el coste de la generación de terreno.
- Implementar un sistema de *Level Of Detail* (*LOD*), destinado a mejorar el rendimiento al cargar con menor calidad los *chunks* más alejados del jugador.
- Añadir vegetación, una de las cosas que añaden mucha vida a el mundo es la vegetación del terreno, siendo necesario un sistema de vegetación o colocación de árboles.
- Un sistema de fluidos, siendo de las partes que se echó en falta a la hora de desarrollar biomas, porque la falta de este no permitía crear ni ríos o ni mares. Aunque este sería el más difícil de implementar y muy costoso a niveles de eficiencia del motor.
- Permitir guardar fauna u objetos extra en los *chunks*, necesitando ampliar el sistema de guardado para poder almacenar estas entidades en los archivos .reg.

Estos elementos son los más destacados, siendo de mayor prioridad los primeros, puesto que estos optimizarían el uso del motor por un tercero y son fácilmente realizables, ya que solo requerirían modificaciones de características del motor ya realizadas (cosa que se hace rápido con los conocimientos ya adquiridos). Mientras que los segundos implementarían una mayor potencia y calidad del motor de Marching Cubes (siendo unos objetivos más a largo plazo).

La realización de todos los puntos dejaría un motor de Marching Cubes que permitiría competir con las *assets* de pago similares, aportando la posibilidad de que se democratizaran los Marching Cubes, debido a que se ofrecería como un *asset* gratuito.

9. Bibliografía

[1] Keen Software Hause, "Pagina web de Mine wars 2081." https://www.minerwars.com/.

[2] "Unity Asset Store: Marching Cube search," 2020. https://assetstore.unity.com/?q=Marching cubes&orderBy=1.

[3] Meshocky, "Space Engineers type of technology used discussion," *Reddit*, 2019. https://www.reddit.com/r/spaceengineers/comments/b44to7/anyone_with_programming_experience_or_knowledge/.

[4] upsidedownfaceman, "Castle Story terrain discussion," *Reddit*, 2012. https://www.reddit.com/r/gamedev/comments/o41md/castle_story_voxels_in_unity/.

[5] Geronimo53, "Astroneers terrain technique used discussion," *Reddit*, 2016. https://www.reddit.com/r/Unity3D/comments/3oftkj/astroneers_terrain_technique/.

[6] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*, 1987, pp. 163–169, doi: 10.1145/37401.37422.

[7] M. J. Dürst, "Letters: Additional reference to 'Marching Cubes,'" *ACM Comput. Graph.*, vol. 22, no. 5, pp. 72–73, 1988, doi: 10.1145/378267.378271.

[8] G. M. Nielson and B. Hamann, "The asymptotic decider: Resolving the ambiguity in marching cubes," *Proc. 2nd Conf. Vis. 1991, VIS 1991*, 1991, doi: 10.1109/visual.1991.175782.

[9] B. K. Natarajan, "On generating topologically consistent isosurfaces from uniform samples," *Vis. Comput.*, vol. 11, no. 1, pp. 52–62, 1994, doi: 10.1007/BF01900699.

[10] E. Chernyaev, "Marching cubes 33: Construction of topologically correct isosurfaces," in *Institute for High Energy Physics, Moscow, Russia,* 1995, p. 10, doi: citeulike-article-id:4505427.

[11] Mojang, "Pagina web de Minecraft," 2011. https://www.minecraft.net/.

[12] E. S. Lengyel, "Voxel-based terrain for real-time virtual simulations," UNIVERSITY OF CALIFORNIA, 2010.

[13] Joel Spolsky, "Trello página web," 2011. https://trello.com/.

[14] P. Bourke, "Polygonising a scalar field," 1994. http://paulbourke.net/geometry/polygonise/.

[15] S. Lague, "GitHub de SebLague, proyecto de Marching Cubes," 2019. https://github.com/SebLague/Marching-Cubes/tree/master/Assets.

[16] Reddit, "Reddit Voxel Game Development." https://www.reddit.com/r/VoxelGameDev/.

[17] Minecraft Wiki, "Minecraft chunk file format." https://minecraft.gamepedia.com/Chunk_format.

[18] Minecraft Wiki, "Minecraft region file format." https://minecraft.gamepedia.com/Region_file_format.

[19] Benjamin, "SoA blog: Creating a Region File System for a Voxel Game." https://www.seedofandromeda.com/blogs/1-creating-a-region-file-system-for-a-voxel-game.

[20] Lamloumi Afif, "StackOverflow compress/decompress an array of bytes," 2016, [Online]. Available: https://stackoverflow.com/questions/40909052/using-gzip-to-compress-decompress-an-array-of-bytes.

[21] Red Blob Games, "Making maps with noise functions," 2020. https://www.redblobgames.com/maps/terrain-from-noise.

[22] Unity 3D, "Unity Manual: Mathf.PerlinNoise," 2019. https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html.

[23] Unity 3D, "Unity Manual: Compute shaders," 2019. https://docs.unity3d.com/Manual/class-ComputeShader.html.

[24] Unity 3D, "Unity Manual: C# Job System," 2019. https://docs.unity3d.com/Manual/JobSystem.html.

[25] Unity 3D, "Unity Packages: Burst documentation," 2019, [Online]. Available: https://docs.unity3d.com/Packages/com.unity.burst@1.4/manual/docs/QuickStart.html.

[26] Microsoft, "Stopwatch class .NET." https://docs.microsoft.com/es-es/dotnet/api/system.diagnostics.stopwatch?view=netcore-3.1.

[27] Unity 3D, "Unity Manual: Shaders," 2019. https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html.

[28] J. Garzo, "Repositorio público proyecto Marching Cubes en Unity 3D," 2020. https://github.com/Javier-Garzo/Marching-cubes-on-Unity-3D.

[29] indeed.com, "Salarios de programador junior en España," 2020. https://es.indeed.com/salaries/programador-junior-Salaries?period=yearly.

[30] indeed.com, "Salarios de programador senior en España," 2020. https://es.indeed.com/salaries/programador-senior-Salaries?period=yearly.

[31] indeed.com, "Salarios de desarrollador de software en España," 2020. https://es.indeed.com/salaries/desarrollador-de-software-Salaries.

[32] Statista and R. Fernández, "Número medio de horas de trabajo al año acordadas en convenios colectivos en España de 2006 a 2019," 2020. https://es.statista.com/estadisticas/478441/promedio-de-horas-de-trabajo-al-ano-segun-convenios-colectivos-de-espana/.

[33] Steamspy, "steamspy: Astroneer," 2020, [Online]. Available: https://steamspy.com/app/361420.

10. Anexo

10.1. Propuesta de proyecto

Nombre alumno:	Javier Garzo Perez
Titulación:	Doble grado en Ingeniería informática y Diseño y desarrollo de videojuegos
Curso académico:	2019-2020

1.TÍTULO DEL PROYECTO

Marching cubes en Unity 3D

2.DESCRIPCIÓN Y JUSTIFICACIÓN DEL TEMA A TRATAR

El objetivo consiste en desarrollar en Unity 3D el algoritmo de marching cubes para el desarrollo de terreno 3D que permita ser modificado durante el juego y que también permita crear un mundo autogenerado e infinito.

3.OBJETIVOS DEL PROYECTO

Los objetivos del proyecto son:

1. Estudio del estado actual de las aplicaciones del algoritmo marching cube en la actualidad.
2. Crear el algoritmo de marching cubes en unity 3D.
3. Aplicar el algoritmo para crear un terreno.
4. Editar este terreno durante la ejecución del juego.
5. Generar terreno automáticamente usando el terreno generado con marching cubes.

4.METODOLOGÍA

Sera desarrollado durante la primera reunión con el profesor, así como la planificación de tareas o las observaciones adicionales.

5.PLANIFICACIÓN DE TAREAS

6.OBSERVACIONES ADICIONALES

El tutor del proyecto será Jorge Echeverria.

10.2. Conjunto sprints del proyecto

Tabla sprint

Título: Recogida de información e inicio del proyecto		
ID: 0	Prioridad: Alta	Dificultad: Media
Tareas -Buscar lista uso previo de los Marching Cubes -Buscar documentación teórica/código de Marching Cubes -Implementar los 15 estados del *voxel* en Unity 3D.		
Horas estimadas: 45 horas		

Pruebas de aceptación

- Comprobar que las caras de los *voxel* de configuración generadas corresponden con los de las imágenes del estado del arte. (Superada)
- Probar la interpolación para ver si provoca las modificaciones esperadas, por ejemplo: en los cubos de configuración. (Superada)

Observaciones

- La dificultad del uso del algoritmo de Marching Cubes radica más en su comprensión e implementación que en la cantidad de código generado
- Se ha estimado que funciona correctamente a la hora de generar un único *voxel*. Sin embargo, no va a ser hasta el *sprint* con id 2, que se va a usar agrupaciones de *voxels*, lo que podría revelar problemas que no se han comprobado ahora.

Tabla sprint

Título: Sistema de chunks		
ID: 1	Prioridad: Alta	Dificultad: Media
Tareas -Crear sistema de chunks • *Chunk* • Sistema de *chunks* - Memoria – Estado del arte		
Horas estimadas: 45 horas		

Pruebas de aceptación

- Comprobar el correcto funcionamiento del sistema de *regions* y *chunks*, que el código sea capaz de cargar un conjunto de *chunks* de forma automática, sin errores de ejecución. (Superada)

Observaciones

- El estado del arte requerirá varias iteraciones para estar completado de forma correcta. Esto seguramente también se aplicará al resto de puntos de la memoria.
- Se ha realizado una clase estática ("Constants"), que será usada para almacenar variables y constantes. Las tablas de triangulación se moverán a esta variable.

Tabla sprint

<table>
<tr><td colspan="3">Título: Terreno simple</td></tr>
<tr><td>ID: 2</td><td>Prioridad: Media</td><td>Dificultad: Baja</td></tr>
<tr><td colspan="3">Tareas
-Modificar los chunks para generar un terreno simple
• Generación de terreno llano
• Cargar y descargar chunks en función de la distancia
- Memoria – Objetivos e Introducción</td></tr>
<tr><td colspan="3">Horas estimadas: 45 horas</td></tr>
</table>

Pruebas de aceptación

- Generar de forma visual el terreno llano haciendo uso de la combinación del sistema de *chunks*, del *sprint* 1, y el "MeshBuilder", del *sprint* 0. (Superado)
- Simular mediante un jugador flotante o la propia cámara la carga y descarga de *chunks* debido a la distancia. (Superado con problemas, el sistema funciona bien, pero los FPS caen demasiado al cargar nuevos *chunks*. Se ha creado un sistema para evitar la carga de múltiples *chunks* a la vez, pero es insuficiente. Este problema se traspasa al *sprint* id:7, dedicado a pulir el código.)

Observaciones

- Durante la corrección del estado del arte se ha creado una clase ("CubeCasesReduction"), que permite visualizar la explicación del porque la reducción de 256 casos del *voxel* a solo 15.
- Las pruebas de aceptación no han sido superadas en su totalidad, ya que se presentan problemas de eficiencia no aceptables a la hora de cargar *chunks*. Solución tomada explicada en la *review* de la prueba de aceptación.
- Se ha tomado la decisión de ir realizando el apartado de implementación conforme se desarrollan los *sprints*, con el objetivo de facilitar el desarrollo de ese punto. Lo que incremento bastante el tiempo que se esperaba dedicar a la memoria.

Tabla sprint

Título: Edición de terreno		
ID: 3	Prioridad: Media	Dificultad: Media
Tareas -Aplicar edición *in-game* de los *chunks* • Eliminar terreno • Añadir terreno • Cambiar color -Memoria - Metodología		
Horas estimadas: 30 horas		

Pruebas de aceptación

- Crear una estructura usando el editor (añadir *voxels*) y un socavón o agujero (eliminar *voxels*), insistiendo en la zona de donde se unen los *chunks* (la zona más problemática). (Superado)
- Generar un terreno que conforme profundizas presenta diferentes texturas en diferentes alturas. Estas texturas van indicadas en los datos del vértice y deben indicar la posición que ocupan en el *uv map*. (Superado en ambos casos de texturización, aunque en el caso de la texturización estándar los *yv* no son correctos según la triangulación de la malla. Esto se podría arreglar en los *sprint* finales o utilizando texturas que no cuenten con estructuras geométricas, ej: ladrillos.)

Observaciones

- Se ha reducido a 40 la altura total de los *voxels* hasta mejorar la eficiencia. Esto permite solventar de forma temporal los picos al generar *chunks*, ya que el tamaño del *chunk* se reduce drásticamente (256 de altura a 40).
- El sistema de texturizar los *chunks* ha incrementado los costes de la generación de estos, una estimación de entre un 10-20%. El problema de eficiencia sigue estando en realizar las operaciones de los marching cubes más que la texturización.
- Se han realizado dos tipos de texturización estándar y *flat shading*, esto ha hecho que se exceda un poco el tiempo previsto para este *sprint*.
- El *shader* tiene en cuenta solo la dirección de la luz, pero no el sombreado, eso provoca que exista luz en cuevas y subsuelos. Hay que cambiar de *shader*, aplazo *sprint* id 8.

Tabla srpint

Título: Sistema de guardado del terreno		
ID: 4	Prioridad: Alta	Dificultad: Alta
Tareas -Aplicar edición in-game de los *chunks* • Guardar cambios en el terreno • Cargar terreno modificado -Memoria – Estudio económico		
Horas estimadas: 30 horas		

Pruebas de aceptación

- Modificar un terreno para que se guarde y parar el juego. Después de iniciar de nuevo el nivel el *chunk* que había sido modificado debería cargarse con los cambios realizados previamente. (Superado)

Observaciones

- Ha llegado el punto de la memoria que los datos son temporales, en este caso las horas destinadas al proyecto deberán modificarse antes de entregar la memoria. Sin embargo, el texto, así como los resultados seguramente no requieran modificación ya que serán próximos a los reales.
- Se ha implementado un sistema de compresión usando la clase "GZipStream" de .NET. La cantidad de compresión es enorme (15 MB = 33KB), pero es debido a que los datos de ejemplo son iguales entre sí, esto cambiara con el terreno aleatorio de los siguientes sprints.

Tabla sprint

Título: Terreno aleatorio		
ID: 5	Prioridad: Alta	Dificultad: Media
Tareas -Sistema de ruido para la generación de terreno aleatorio • Sistema de ruido -Memoria Resultados y conclusiones		
Horas estimadas: 30 horas		

Pruebas de aceptación

- Crear un sistema de ruido que permita generar un terreno diferente al terreno plano actual, con diferentes texturas y relieve a lo largo de este. (superado, se ha generado un terreno montañoso)

Observaciones

- En la memoria la estructura de resultados está terminada, pero algunas imágenes deberán cambiarse con las finales. También la sección destina a comparar el desarrollo del proyecto estimado con el real se ha dejado con unas indicaciones para ser desarrollada al final de todos los *sprints*.
- Al no tener sistema de fluidos o algo similar para crear agua, no se pueden realizar terrenos con ríos, lagos o similares. En este punto lo único que se podría hacer un bloque solido con textura de un bloque de agua, pero no funcionaría como agua o el jugador se hundiría en ella, así que se han realizado ejemplos montañosos o llanuras.
- El sistema de ruido es 2D, por lo que nos permite generar relieve, pero no soporta un sistema de cuevas como se dan en juegos del tipo. Esto es principalmente por que debido límite de tiempo, usar ruido 3D estaría fuera de las posibilidades.

Tabla sprint

Título: Biomas		
ID: 6	Prioridad: Alta	Dificultad: Media
Tareas -Sistema de ruido para la generación de terreno aleatorio • Tipos de ruido/biomas • Integración entre diferentes ruidos/biomas -Memoria – Implementación		
Horas estimadas: 30 horas		

Pruebas de aceptación

- Adaptar el "NoiseManager" para soportar diferentes biomas (tipos de terreno con distintas características). Estos deben unirse de forma orgánica, es decir nada de un *chunk* de un tipo de bioma y otro de otro, debe haber una fusión de ambos en las zonas fronterizas. (Superado)
- Realizar unos biomas extra, entendiendo que el terreno montañoso realizado en el anterior *sprint* cuenta como bioma previo. (Superado, tres nuevos biomas generados: desierto, llanuras y glacial)

Observaciones

- Para acabar la implementación se deberá incluir el siguiente *sprint* (id:7). En cuanto el *sprint* con id 8, se decidirá después de su finalización si se incluye o no a la implementación, depende si se desarrolla el sistema de *chunks*.

Tabla sprint

Título: Fin del código y memoria		
ID: 7	Prioridad: Alta	Dificultad: Media
Tareas -Revisar/pulir código. - Memoria - Resumen, palabras clave, introducción, bibliografía y anexos.		
Horas estimadas: 45 horas		

Pruebas de aceptación

- Volver a incrementar la altura de un *chunk* a 256 y reducir los milisegundos por *frame*, es decir incrementar los FPS del proyecto. (Superado, coste reducido al 14,96% respecto el original)

Observaciones

- La sección de pulir el código ha ido dedicada a incrementar la eficiencia de la generación de *chunks*, problema que se asignó en el *sprint* con id 2.
- Se han hecho cambios relacionados con la correcta maquetación, como pueden ser tablas, anexos, saltos de página entre diferentes puntos. Además, se han realizado un par de cambios en el punto de resultados.
- Como contenido final añadido a la memoria, se ha realizado el resumen final y se espera su aprobación por el tutor para realizar el abstract.

Tabla sprint

Título: Preparación presentación		
ID: 8	Prioridad: Media	Dificultad: Baja
Tareas - Preparar proyecto para una demo / presentación.		
Horas estimadas: 30 horas		

Pruebas de aceptación

- Sin pruebas de aceptación parecidas a las anteriores, ya que no se tenía en mente ningún cambio en el que se centrara en específico el *sprint*. Por lo que se iniciaría con la intención de realizar mejoras variadas orientadas a la utilidad y estética del proyecto hasta alcanzar unas 16-20 horas y dejar el resto para pulir la documentación.

Observaciones

- Se ha decidido añadir el *sprint* 8 a la implementación pese a no realizar cambios en el motor de Marching Cubes, aunque se hará bajo una estructura un poco diferente a la de los anteriores *sprint*s.
- Los puntos desarrollados en este *sprint* han consistido en mejorar los *shaders* usados, cambias la altura de los *chunks* a 80 y añadir una pequeña interfaz con inputs a las escenas de Unity, para mejorar la interacción del usuario con las demos.
- Los puntos desarrollados en la memoria son el punto 5.9, el abstract y un pulido general de esta (corregir faltas, imágenes faltantes y horas destinadas). Cerrando la memoria.

10.3. Actas realizas con el tutor

REUNIÓN: 1ºReunión

Fecha:	30/10/2019		
Hora comienzo:	17:05	**Hora finalización:**	17:40
Lugar:	Edificio rectorado, 2º planta		
Elabora acta:	Javier Garzo		
Convocados:	Javier Garzo, Jesús Lacarte, Francisco Pérez, Jorge Echeverria		

Orden del día / Acta

No.	Asunto	Acuerdo
1	Explicación actas reunión.	001
2	Utilización de herramientas administración de trabajo (Trello, diagrama de Gantt) y guardar estado inicial de estos.	002
3	Explicación realización de estado del arte inicial.	003
4	Ejemplos de proyectos anteriores de ejemplo y como obtenerlos.	004
5	Resolver dudas respecto lo hablado.	
6		
7		
8		
9		
10		

Resumen de acuerdos

Número	Acuerdo	Plazo	Responsable
001	Realización primera acta		Javier Garzo
002	Realización de Trello y diagrama de Gantt		Javier Garzo
003	Empezar el estado del arte		Javier Garzo
004	Entregar unos ejemplos de tfgs.		Jorge Echeverria
005			
006			

REUNIÓN:	**2ºReunión**

Fecha:	28/11/2019		
Hora comienzo:	17:00	**Hora finalización:**	17:50
Lugar:	Edificio rectorado, 2º planta		
Elabora acta:	Javier Garzo		
Convocados:	Javier Garzo, Jorge Echeverria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	Resumen rápido del estado actual del Trello, diagrama de Gantt e idea de cómo serán los *sprints.*	001
2	Hablar sobre modificaciones hechas en Trello.	002
3	Hablar sobre la memoria y su asignación en diagrama de Gantt.	003
4	Ejemplos metodologías.	004
5	Recomendación de creación de pruebas de aceptación.	005
6	Recomendaciones de tfgs para leer.	006
7		
8		
9		
10		

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Realizar una foto del estado actual del proyecto antes de realizar cambios.	Inmediato	Javier Garzo
002	Dividir en tarjetas Trello en función de los sprints.	Inmediato	Javier Garzo
003	Dividir memoria en Trello y empezar ya su desarrollo en este caso el estado del arte.	Inmediato	Javier Garzo
004/005/006	Revisar tfgs recomendados poniendo énfasis en las partes de metodologías, estado del arte y pruebas de aceptación.	Inmediato	Javier Garzo

REUNIÓN:	**3ºReunión**

Fecha:	31/01/2020		
Hora comienzo:	17:30	**Hora finalización:**	18:10
Lugar:	Edificio rectorado, 2º planta		
Elabora acta:	Javier Garzo		
Convocados:	Javier Garzo, Jorge Echeverria		

Orden del día / Acta

No.	Asunto	Acuerdo
1	Hablar de la corrección del estado del arte, que trato sobre temas redacción, maquetación, bibliografía.	001
2	Enseñar progreso en Unity 3D.	002
3		
4		
5		
6		
7		
8		
9		
10		

Resumen de acuerdos

Número	Acuerdo	Plazo	Responsable
001	Modificar todos los puntos hablados del estado del arte (figuras, bibliografías, algunos párrafos, añadir imágenes técnicas).	Siguiente reunión	Javier Garzo
001	Probar plugin para word mendelev.	Siguiente reunión	Javier Garzo
001	Enviar correo recordando ideas habladas.	Inmediato	Javier Garzo
002	Seguir trabajando en el código de Unity 3D.	Sprint actual	Javier Garzo

REUNIÓN:	**4ºReunión**

Fecha:	12/06/2020		
Hora comienzo:	17:00	**Hora finalización:**	17:30
Lugar:	Reunión remota a través de Microsoft teams		
Elabora acta:	Javier Garzo		
Convocados:	Javier Garzo, Jorge Echeverria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	Hablar sobre la entrega del proyecto, las fechas de entrega y como estructurar a lo largo de este.	001
2	Comentar los problemas recibidos en la corrección del email: Tener cuidado con las comas y símbolos de puntuación, revisar maquetación (Sistema A y B misma figura), realizar la mejora de algunos párrafos que no quedan claro del todo.	002
3	Hablar de la realización de la metodología para la próxima entrega.	003
4	Hablar de la mejora de la explicación del apartado del estado del arte de Marching cubes.	
5		
6		
7		
8		
9		
10		

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Realizar el proyecto en su totalidad.	11/09	Javier Garzo
002	Corregir todos los puntos dados, tanto los más específicos como los generales.	3 semanas	Javier Garzo
003	Enviar el trabajo con el nuevo punto y acordar una reunión.	3 semanas	Javier Garzo

REUNIÓN:	**5ºReunión**

Fecha:	07/09/2020		
Hora comienzo:	12:30	**Hora finalización:**	13:05
Lugar:	Reunión remota a través de Microsoft teams		
Elabora acta:	Javier Garzo		
Convocados:	Javier Garzo, Jorge Echeverria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	Hablar sobre las dudas relacionadas con la memoria (estilos, estructuras y la mejora de algunos párrafos específicos).	001
2	Hablar de la entrega del proyecto.	002
3	Tratar el tema de la autorización a la fase de demostración y defensa	003
4		
5		
6		
7		
8		
9		
10		

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Revisar la memoria	Antes de la entrega	Jorge Echeverria
002	Realizar la entrega del trabajo	11/09	Javier Garzo
003	Rellenar los datos de la autorización de la defensa y enviárselos al tutor	08/09	Javier Garzo

10.4. Documentación adjunta

Como documentación complementaria se aportan los siguientes elementos:

- Autorización del TFG: Autorización dada por el tutor para la defensa del TFG.
- Marching-cubes-en-Unity-3D: Es el proyecto de Unity 3D desarrollado en la versión de Unity 2019.4.8f1, aunque cualquier versión similar serviría para ejecutarlo (2019.4.*f*). Esta contiene diferentes niveles para probar el Motor de Marching Cubes, así como todo lo realizado durante el desarrollo del proyecto.
- Demo compilada: Una escena compilada específicamente para que en caso de no tener el editor de Unity 3D poder probar el motor de Marching Cubes. Se recomienda bajar Unity 3D y usar el proyecto, ya que la demo contiene la escena más importante, pero carece de otras escenas que también tienen su importancia dentro del proyecto.
- Readme.txt: Una explicación básica sobre las diferencias entre la demo compilada y el proyecto de Unity 3D. Se recomienda su lectora antes de proceder a utilizar tanto la demo compilada como el proyecto de Unity 3D.