Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Desarrollo de un entorno para la creación de videojuegos mediante Unity3D

Develompent of an enviroment for game development using the Unity 3D game engine

Germán Alfonso Teixidó

La Laguna, 11 de septiembre de 2020

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43826207Y, Profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Desarrollo de un entorno para la creación de videojuegos mediante Unity3D"

ha sido realizada bajo su dirección por D. **Germán Alfonso Teixidó**,

con N.I.F. 42238605W.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de septiembre de 2020

Licencia

Resumen

El objetivo de este trabajo ha sido la materialización de un entorno para la creación de un juego de estrategia por turnos desarrollado en el motor gráfico Unity3D.

El objetivo principal producir un entorno dotado con herramientas que permitan a diseñadores de niveles y artistas crear sus propios mapas, tipos de unidades y editar reglas específicas del juego, como condiciones de victoria, el ataque u movilidad de algunas unidades, etc.

Principalmente, el entorno de desarrollo debe de estar dotado, como mínimo, con un editor de mapas y un editor de unidades jugables.

Los equipos de desarrollo enfocados al apartado artístico deben involucrarse con código la menor cantidad de tiempo posible, preferiblemente, pudiendo realizar todo su trabajo a través de las herramientas desarrolladas por el programador.

Asimismo, dichas herramientas no pueden romper u afectar al rendimiento del gameplay del producto final: unas herramientas no sirven para nada si no se puede hacer un juego con ellas, u son nocivas para el desarrollo a largo plazo.

Palabras clave: Videojuegos, entorno, Unity, editor.

Abstract

The objective of this work has been the materialization of an environment for the creation of a turn-based strategy game developed in the Unity3D game engine.

The main objective is to produce an environment equipped with tools that allow level designers and artists to create their own maps, types of units and edit specific rules of the game, such as victory conditions, the attack or mobility of some units, etc.

As a minimum, the development environment must be equipped, at least, with a map editor and an editor of playable units.

The artistic branch of the development team should not be involved with code, thus, an important flag on the making of this tools would be the fact of artists and designers being able to realize all their work across the tools developed by the programmer.

Of course, these tools cannot break or affect the performance of the gameplay of the final product: tools are useless if you cannot make a game with them or are harmful to long-term development of such game.

Keywords: Videogames, environment, Unity, editor.

Índice general

Índice de figuras

1 Introducción

1.1 Planeando Objetivos

Nuestro objetivo es desarrollar un juego de estrategia por turnos con una estética similar a *Endless Legend*, aunque con una jugabilidad mas simplificada, como la que podría tener el videojuego *Wargroove*.

Figura 1.1: Comparativa de Endless Legend y Wargroove.

Si bien nuestro enfoque durante el desarrollo de este anteproyecto será meramente técnico, es necesario entender tanto la complejidad de *gameplay* a la que aspiramos como la estética que nuestro equipo de desarrollo busca para poder implementar las herramientas que le permitan desarrollar el juego.

1.1.1 Estética

El juego por desarrollar se resolverá en **un tablero hexagonal** similar al de juegos de rol de mesa, pero con un modelado 3D que contendrá detalles como altura, decoración, caminos, ríos, mares, diferentes terrenos, murallas, etc. Las unidades se visualizarán con un modelado 3D que será de un color u otro dependiendo del equipo al que forme parte.

1.1.2 Jugabilidad

El juego consistirá en una batalla **1vs1** jugador contra jugador en un ***TBS*** (*turn based strategy* – **juego de estrategia por turnos**). Para simplificar el desarrollo de este trabajo, nos centraremos únicamente en un modo de juego offline, omitiendo las inmensas complejidades que supondría añadir un modo multijugador en línea. Cada mapa constará con dos jugadores controlando unas unidades. El objetivo de la victoria será controlar cierta cantidad de casillas antes que el rival, o destruir todas las unidades del oponente.

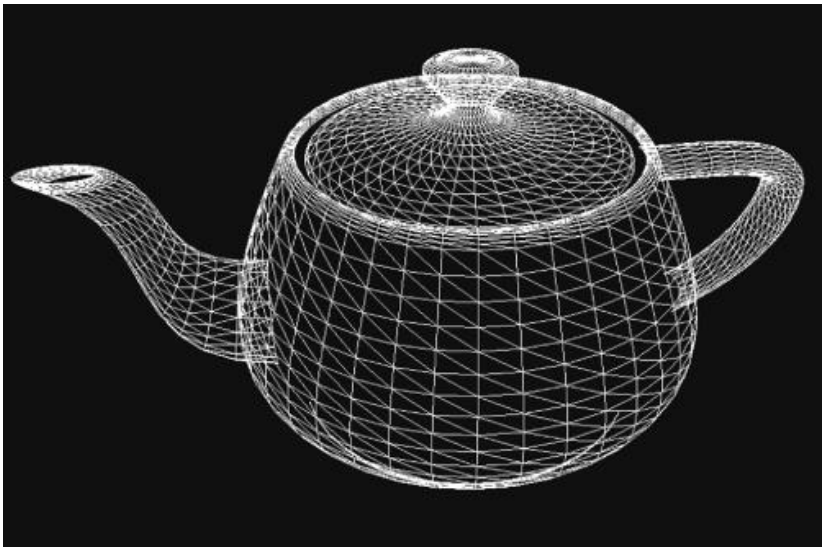
1.2 Planificando herramientas

1.2.1 Editor de mapas

El primer objetivo, y el más evidente, es el desarrollo de un **editor de mapas**. Es menester que este compuesto por diferentes casillas que puedan ser modificadas para mostrar una altura y un tipo de terreno, así como poder ser decorada con elementos decorativos. Cada casilla debe de almacenar información extra, como su posición, si tiene unidades o edificios, conexiones a otras casillas, etc. Por supuesto, estos mapas deben poder ser guardados para ser reeditados o jugados por los jugadores en un futuro.

Debido a la gran complejidad y flexibilidad necesaria para producir un mapa orgánico de este estilo, no podemos trabajar directamente con *Prefabs* (objetos predefinidos en Unity) de hexágonos en 3D a los que editar, ya que necesitamos representar transiciones de un tipo de terreno a otro, diferencias de alturas (rampas, escaleras), ríos…

Por ende, trabajaremos creando nuestra propia malla triangular para modelar nuestro mapa en 3D. Las herramientas de desarrollo de mapas irán destinadas a modificar esta malla (la cual llamaremos ***mesh*** al partir de ahora) a la vez que la información almacenada en cada casilla.



1.2: Una mesh modelando una tetera.

1.2.2 Editor de unidades

El segundo objetivo será implementar un **editor de unidades** que facilite a los desarrolladores la creación (y eliminación) de unidades *ingame*, así como modificar sus propiedades, modelado, atributos y estadísticas dentro del juego, con el objetivo de facilitar tareas de balance y similares.

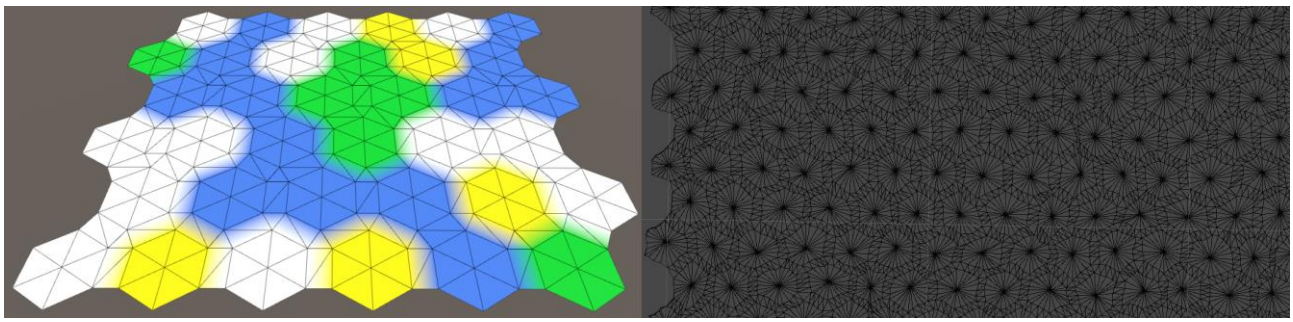
Estas unidades deben poder ser implementadas sin necesidad de escribir una sola línea de código por parte del diseñador, poseer una flexibilidad aceptable a la hora de crear y editar unidades, ser sencilla, rápida y comprensible de utilizar.

El proceso de introducir nuevas unidades desde el editor al *gameplay* debe de estar automatizado o, en su defecto, ser lo más sencillo posible.

2 Editor de Mapas

Como se ha comentado en el capítulo de Introducción, un editor de mapas es necesario. Sabemos que cada casilla es hexagonal y debe estar conectada al resto de casillas, o bien directamente, o bien usando un puente. Decidir como triangularemos nuestro tablero es esencial para saber que tipo de propiedades podremos introducirle o no en un futuro. Esta es sin lugar a duda la herramienta más exhaustiva y que mas trabajo requiere para funcionar.

2.1 Tablero y forma de las meshs.



2.1.1 Evolución de la complejidad de las casillas y forma básica de la mesh a lo largo del desarrollo.

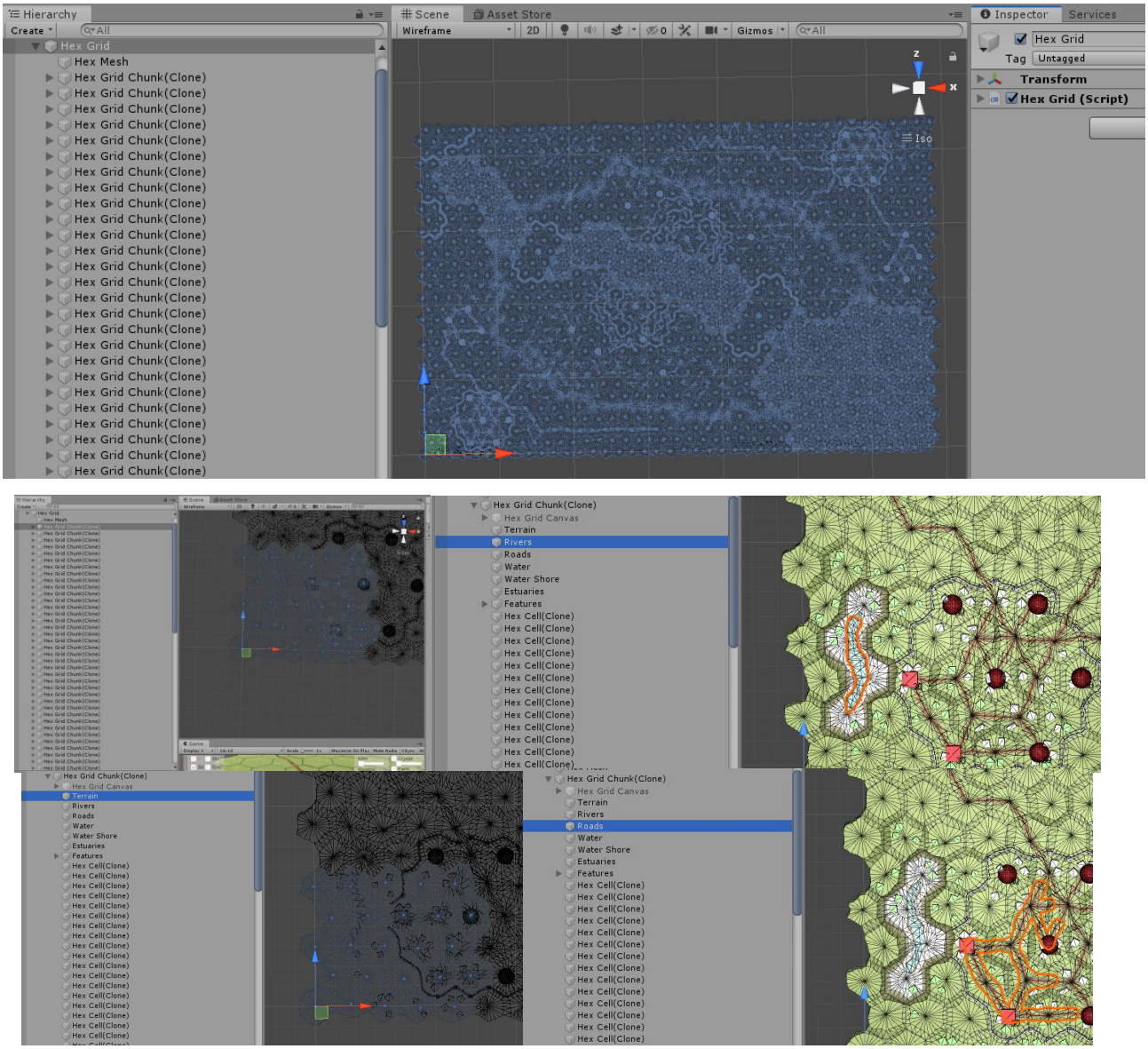
En la imagen de arriba a la izquierda podemos apreciar la forma básica de nuestra *mesh* al comienzo del desarrollo. Nos permite tener varias casillas formadas por 6 triángulos equiláteros conectadas las unas a las otras a través de un 'puente' rectangular formado por 2 triángulos rectángulos, y dichos puentes a su vez están conectados a través de triángulos, todo con el fin de obtener el modelado de un tablero hexagonal en el que se permite un degradado de colores natural, aunque el aspecto del mismo sea todavía muy artificial (debido a su aspecto totalmente hexagonal).

A lo largo del desarrollo nuestra *mesh* se irá desformando tanto con el fin de pulir este defecto (usando soluciones como Ruido de Perlin, aumentando el número de vértices necesario para triangular un hexágono, dividiéndose en diferentes *sub-meshs*), a la vez que para implementar nuevas funcionalidades decorativas. No obstante, el espíritu de la forma original, aunque difícil de reconocer, se mantendrá en el final de nuestro tablero, como se puede apreciar a la izquierda de la imagen.

En un futuro, nuestro tablero se dividirá en *chunks* (esto será explicado más adelante en el punto **2.5 *Chunks***), esto es, en lugar de una única *mesh* encargada de triangular todo el tablero, dividiremos el tablero en trozos, y cada trozo gestionará su propia *mesh* y casillas. Asimismo, cada *chunk* no constará con tan solo una *mesh*, ya que en el futuro tendremos que usar varias para representar diferentes elementos (terreno, agua, caminos, etc.).

Por ende, las **responsabilidades** de nuestro tablero son:

- Gestionar cada uno de sus *chunks* (*Hex Grid Chunks*).
  - Cada *chunk* es responsable de la triangulación y modelado de las *meshs* de la parte del mapa que representa.
- Almacenar la lista de unidades y edificios del mapa.
- Encargarse de efectos de movimiento de unidades, pathfinding, etc.
- Identificar el espacio tridimensional con el que el usuario esta interactuando con la casilla que representa ese espacio en particular.

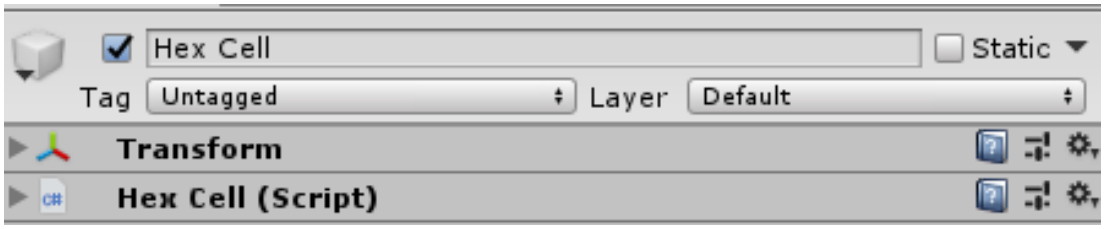


2.1.2 Jerarquía del tablero y meshs que lo componen.

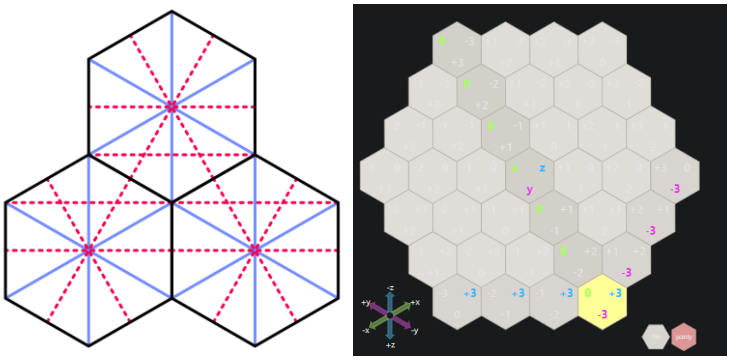
2.2 Casillas

Cada casilla de nuestro mapa debe almacenar una importante cantidad de información.

Para ello, instanciaremos un *Prefab* de un *GameObject 'HexCell'* por cada casilla. **Este *GameObject* solo contedrá información de la casilla que representa en una clase escrita en *C#*, pero no se encarga de su visualización** (de esto se encarga la *mesh* anteriormente explicada). Las propiedades matemáticas de los hexágonos facilitan en gran medida el traslado de información desde coordenadas hexagonales a coordenadas 2D y 3D, lo cual facilita mucho la visualización de mapas hexagonales en espacios tridimensionales y viceversa. Por ende, cada casilla tiene unas coordenadas 'hexagonales' **x**, **y** y **z** que son convertibles a las coordenadas **X** y **Z** del espacio tridimensional de Unity en el centro de dicha casilla. Cada casilla también tendrá una altura que podrá variar de 1 a 6, las cuales se representarán haciendo uso de la coordenada **Y** del espacio tridimensional. Con matemática básica, podemos calcular las posiciones de los vértices de estas casillas a partir de su centro. Toda esta información es recabada por la *mesh* del terreno a la hora de triangular.

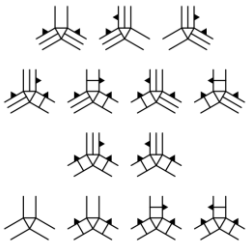


2.2.1 Nuestro Prefab de casillas.



2.2.2 Coordenadas tridimensionales vs hexagonales.

Cada casilla está conectada a un máximo de otras 6 vecinas en las direcciones *NE, E, SE, SW, W y NW*. Es importante que cada casilla se relacione con sus vecinas, ya que contienen información importante a la hora de triangularse: Donde están los bordes de cada una a la hora de construir un puente; si existe algún río, camino o muralla entre ellas; diferencia de altura entre ambas, etc.



2.2.3 Diferentes formas de triangular las conexiones entre casillas según su altura.

Asimismo, las casillas deben memorizar el tipo de terreno que representan (para posteriormente ser texturizadas al triangularse o aplicar algún tipo de efecto al *gameplay*), los adornos que poseen (bosques, ciudades, granjas, templos), si están sumergidas bajo el agua, etc.

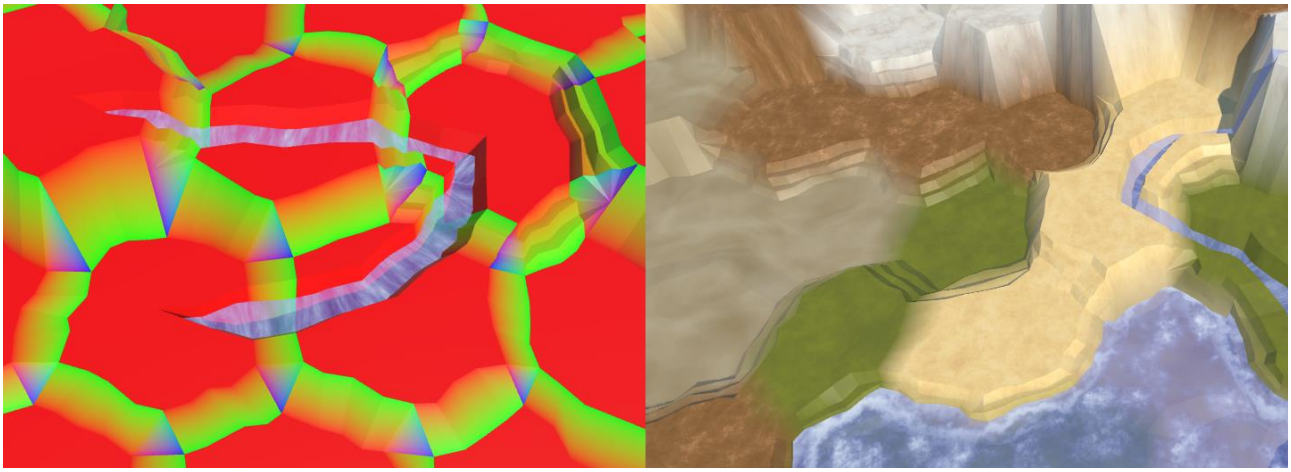
Aunque las casillas se inicializan con los valores establecidos en su prefab, estos son inmediatamente reescritos en caso de que estemos cargando un mapa nuevo, por lo que, llegado el momento de triangular por primera vez, toda la información de cada casilla habrá sido actualizada.

2.3 Texturas

Como los vértices de cada casilla estarán en contacto con vértices vecinos adyacentes únicamente de otras 2 casillas, podemos colorear nuestra *mesh* como un *Splat Map RGB*, sobre el cual imprimir nuestras texturas.

Un *Splat Map* es un método para combinar diferentes texturas, que nos permite mapear la transparencia de una textura. Como solo tendremos un máximo de 3 texturas relacionándose entre sí, podemos mapear el porcentaje de opacidad de cada pixel según la cantidad de rojo, azul o verde que exista en dicho pixel (*RGB – Red, green, blue*).

Para lograr esto, aplicamos un *shader* al material de nuestra *mesh* con el vector de las texturas que deseamos incrustar, y los aplicamos a la hora de triangular la *mesh* de forma que asignamos un color a un tipo de textura y jugamos con la opacidad de cada material respecto a la cantidad de dicho color.



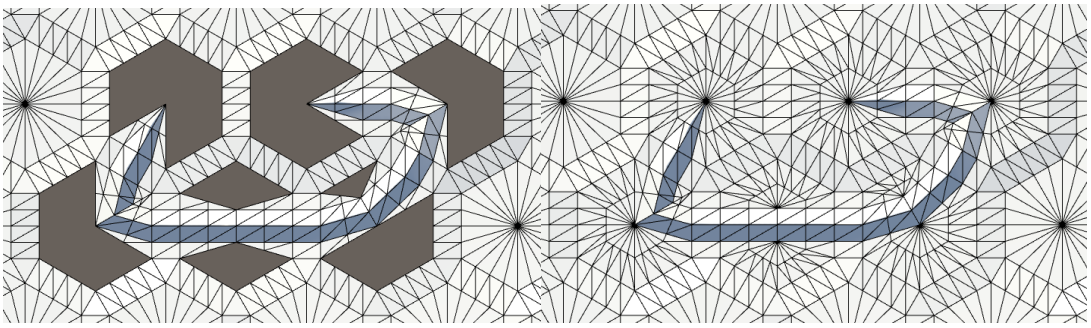
2.3: Al aplicar las texturas de terreno sobre nuestro tablero/Splat Map, logramos efectos muy satisfactorios.

2.4 Decoración

2.4.1 Masas de agua

Obviamente, no podemos tratar de simular masas de agua. En este caso, usaremos el truco mas viejo del libro: Nuestros cuerpos de agua no serán más que una *mesh* plana con un *shader* y material que le haga tener aspecto de agua. No obstante, cada cuerpo de agua es diferente y requerirá un tipo específico de *mesh*.

Para crear **ríos**, primero necesitamos generar un cauce. Esto implica desformar la *mesh* del terreno, y en ocasiones, 'partir' el centro de cada casilla, lo que vuelve la triangulación de cada casilla algo más compleja. Es por esto por lo que nos limitaremos a que cada casilla tan solo tenga un rio entrante y un rio saliente: No nos interesa tener que lidiar con cauces uniéndose o dividiéndose, ya que tendríamos que lidiar con muchísimos casos diferentes.

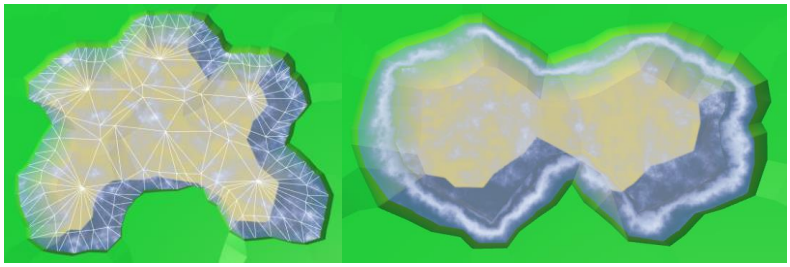


2.4.1.1 Triangular ríos es más complejo de lo que parece.

Sobre este nuevo cauce podemos triangular una *mesh* transparente con un *shader* que aplique ruido y efectos de movimiento que simule un movimiento sincronizado que replique el agua.

2.4.1.2 Shaders en ríos.

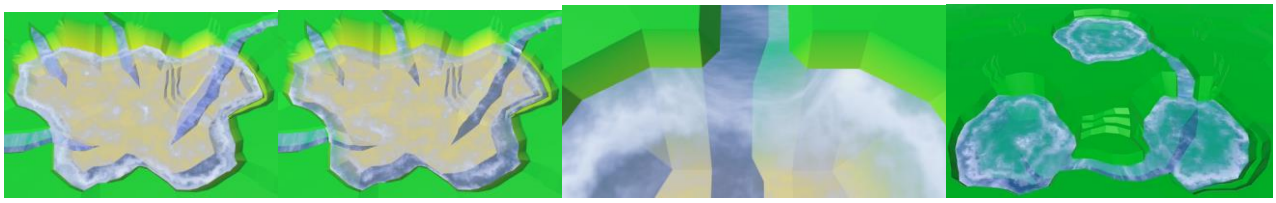
Crear masas de agua inundando una casilla tiene un proceso similar. Tan solo que en este casi triangularemos los bordes de la masa de aguas con *shaders* diferentes a las del cuerpo principal para dar una sensación de oleaje. Es importante triangular bien para que el borde del agua siempre coincida con el borde del terreno y este aspecto no se arruine. Los cuerpos de agua deben estar a una 'altura' superior a la altura de la casilla a la que están inundando, y dicho nivel del mar debe ser similar en las casillas vecinas.



2.4.1.3 Diferencias en los bordes.

Para lograr el efecto de los ríos emergiendo de un lago u desembocando en un cuerpo mayor, es necesario no renderizar los *shaders* de los ríos debajo de cuerpos de agua, y aplicar un nuevo *shader* a una *mesh* intermediaria que sirva para dar la impresión de conectar ambas.

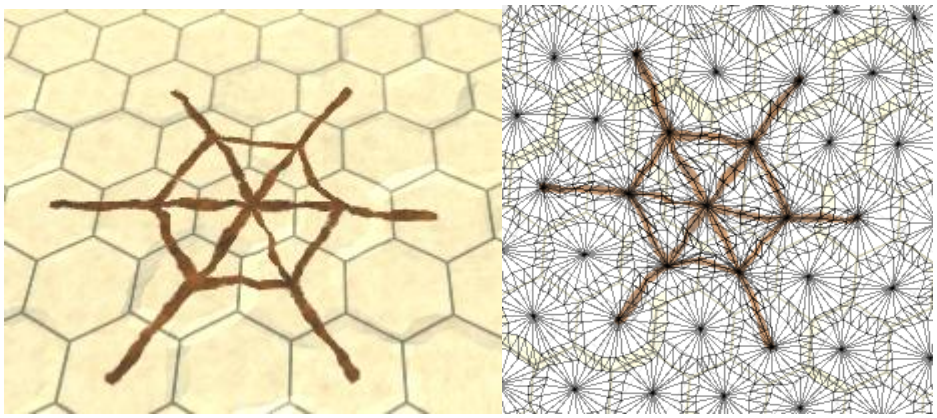
Una pequeña ayuda con la triangulación, para dar la impresión de que los ríos desembocan en cuerpos de agua a través de una cascada, o que emergen naturalmente de ellos, ayuda a mantener la ilusión de un ciclo de agua realista.



2.4.1.4 Flujo del agua.

2.4.2 Caminos

Siendo cada casilla vecina de otras 6 como máximo, podemos conectar dichas casillas a través de caminos. Si dos casillas están conectadas a través de un camino, podemos renderizar en cada una de ellas y en su conexión una *mesh* encima de la casilla que simbolice dicho camino. Los caminos también generan camino entre murallas (separándolas) y ríos (puentes encima de ellos).

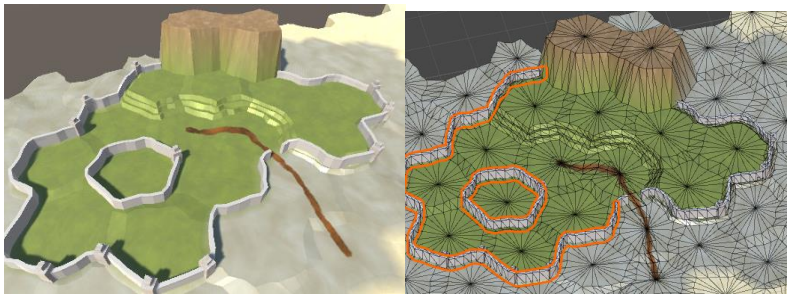


2.4.2.1 Visualización de un camino frente a su mesh.

2.4.3 Murallas

Si una casilla esta ‘amurallada’ y una casilla vecina no lo está, generaremos una pared entre las conexiones de estas dos casillas. Existen casos donde no será necesario que triangulemos murallas, como en el caso de casillas hundidas en el mar, o en el caso de que la casilla adyacente sea una montaña y por ende la ladera natural que forma ya sea una muralla natural en si misma, haciendo que construir otro muro no sea muy útil.

En cualquier caso, las murallas dejarán espacio para que caminos y ríos puedan transcurrir a través de ellas.



2.4.3.1 Murallas y la mesh que las compone.

Podemos añadir elementos decorativos aleatorios a las murallas, como torretas, para dar una impresión de irregularidad.

2.4.4 Monumentos y decoraciones varias

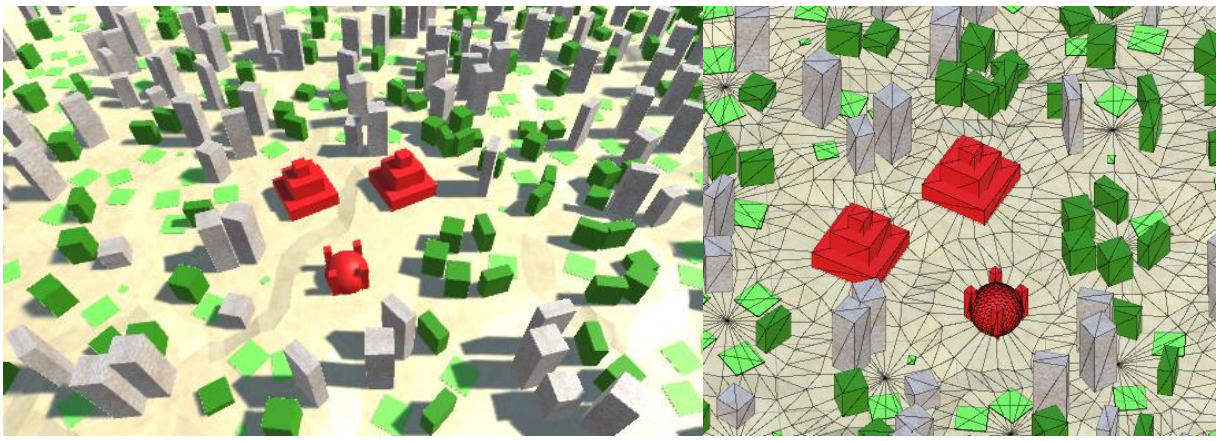
Diferentes elementos decorativos (como árboles, edificios, piedras, césped, granjas, flores, etc.) pueden formar parte de una casilla. Cada hexágono está dividido en 6 triángulos, y cada triángulo puede sostener un elemento decorativo.

No obstante, tener que especificar en que parte de una casilla deseamos colocar un elemento cada vez que deseamos colocar uno puede ser tedioso. Si queremos construir un bosque, ir colocando árbol a árbol puede alargar mucho el proceso. Y guardar toda esa información puede consumir más memoria de la que nos gustaría. En su lugar, cada casilla almacena un nivel de bosques, granjas, edificios… A la hora de triangular, usamos una tabla hash con valores aleatorios usando una *seed* fija para indicar los adornos que se inicializan en cada casilla. A mayor nivel de ese elemento en dicha casilla, mayor probabilidad de que aparezca ese tipo de adorno, con un tamaño más o menos grande.

De esta manera, la tabla hash nos ahorra tener que guardar gran cantidad de información en memoria al almacenar un mapa, y al ser fija, esos elementos siempre aparecerán de la misma forma. Además nos permite diseñar granjas, ciudades o bosques de forma rápida, cómoda y efectiva.

También podemos añadir monumentos en cada casilla.

Los monumentos ‘monopolizan’ la casilla, sin permitir otro tipo de adornos, y su función también es meramente estética (una pirámide, un castillo, etc.).



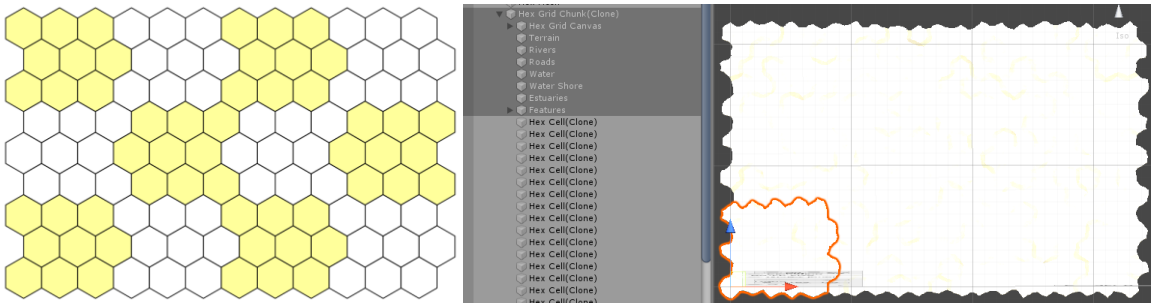
2.4.4.1 Adornos y monumentos, y sus respectivas *meshs*.

2.5 Chunks

Cada vez que modificamos una *mesh*, debemos triangularla de nuevo para reflejar el cambio.

En mapas muy grandes, esto puede causar que cada pequeño cambio ralentice muchísimo el programa por algunos segundos. La solución es no permitir que todo el mapa esté representado por una misma *mesh*, y usar *chunks* en su lugar.

Los *chunks (*trozos*)* son fragmentos del mapa en el que jugamos con la *mesh* que representa a dicha parte del mapa. De esta forma, cuando modificamos una casilla, no debemos triangular de nuevo todo el mapa, sino la *mesh* del *chunk* de dicha casilla (y la de los *chunks* vecinos, si se da el caso).

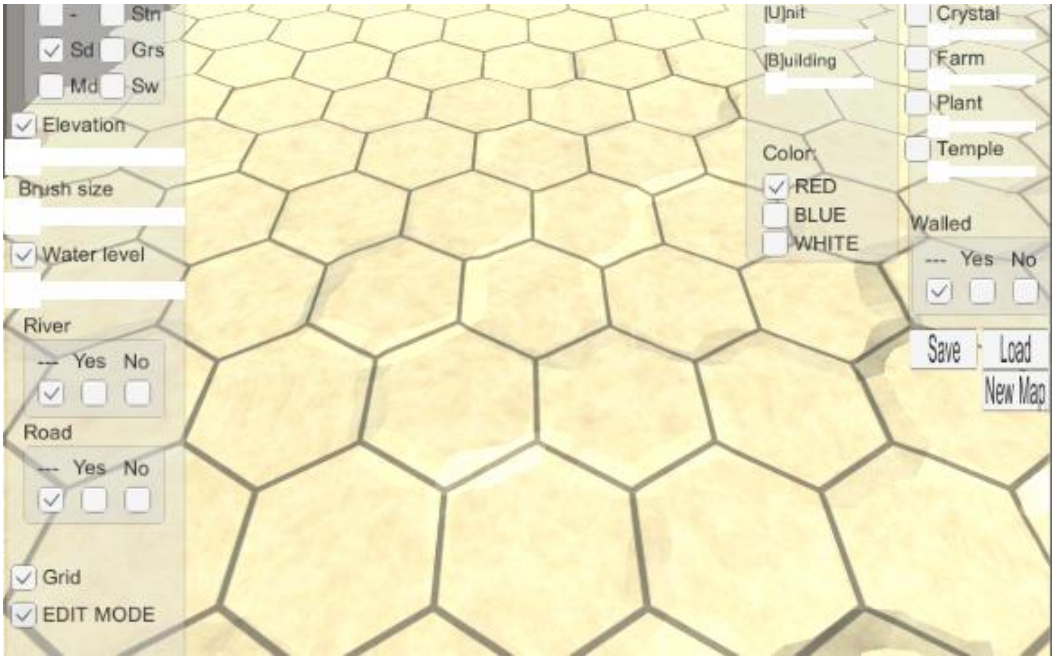


2.5 Los chunks permiten optimizar el rendimiento a la hora de recargar un mapa al editarlo.

2.6 Interfaz

La interfaz es simple y sin ningún tipo de diseño, totalmente funcional y utilitaria.

Podemos asignar un bioma a cada casilla, una altura, ríos, caminos, murallas y niveles de adornos. También se ha añadido una brocha que permite interactuar con mas casillas, facilitando la labor del diseñador. Como extra, se ha añadido una opción ‘grid’ que permite visualizar mejor cada casilla. También tiene botones con accesos directos para guardar cargar y crear nuevos mapas.

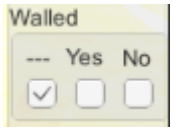


2.6.1 Interfaz del editor de mapas.

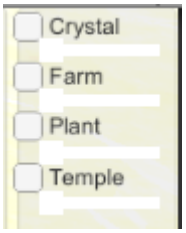
Al hacer click izquierdo sobre una casilla en *EDIT MODE*, podemos realizar las siguientes modificaciones:

Permite seleccionar el tipo de terreno de cada casilla (no realiza modificación en caso de seleccionar '-')

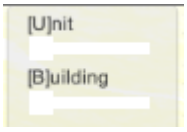
Los indicadores de nivel de agua y elevación de la casilla modificarán estos parámetros de la casilla en caso de estar activos. La casilla tomará valor equivalente al valor que se le especifique a slider (en este caso, un valor entero de 0 a 6).



Permite especificar si una casilla se encuentra o no dentro de un recinto amurallado.

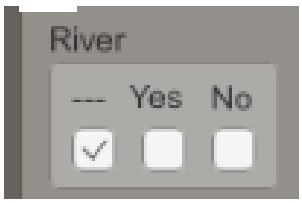


Permite indicar el nivel de elementos decorativos de cada casilla.



Permite decidir qué tipo de unidad deseamos crear al pulsar los botones [U] o [B] (para eliminar una unidad o edificio, shift + U/B).

Al deslizar sobre una casilla a otra con la opción de crear caminos, enlazamos dichas casillas con un camino. Al pulsar en una casilla con la opción de destruirlos, destruimos todas las conexiones de esa casilla.



Los ríos funcionan de forma similar, pero con un detalle importante a tener en cuenta: la casilla desde la que comenzamos a deslizar definirá su *'outcoming river'* o río saliente, mientras que la otra casilla será la que defina su *'incoming river'* o río entrante.

Este *slider* nos permite aumentar el 'rango' de casillas con el que interactuamos, como si aumentásemos el tamaño de brocha en un editor de imágenes.

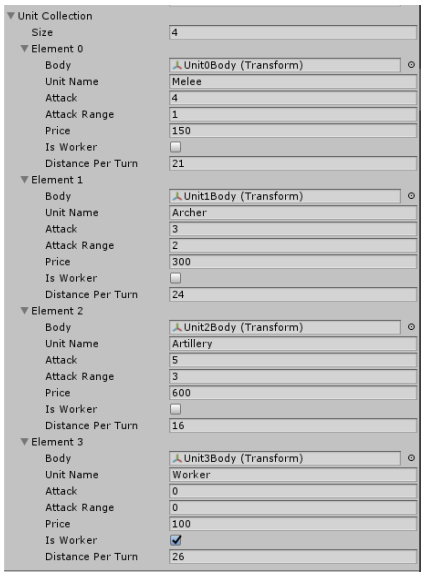
Estos botones nos permiten visualizar los bordes de cada casilla para identificar cada una de ellas más fácilmente. El botón de EDIT MODE permite cambiar de modo de diseño a modo de *gameplay* para probar cosas rápidamente.

Finalmente, los botones de guardar y cargar mapas, así como el botón de crear un nuevo mapa.

2.6.2 Nuestra hud de selección de mapa a guardar/cargar.

3 Editor de Unidades

El editor de unidades guarda en un vector de memoria la información que los diseñadores asignan a cada unidad. Editar unidades es fácil y se puede hacer desde el mismo editor de Unity:



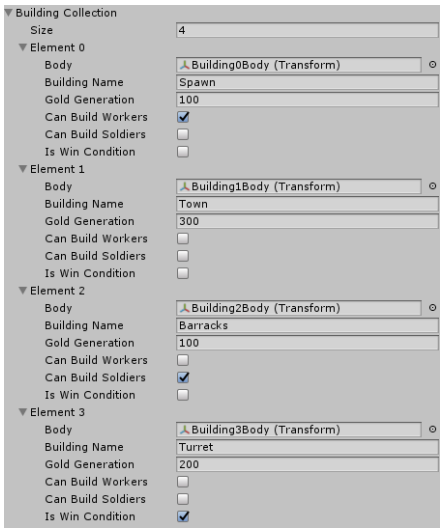
Para añadir nuevas unidades, basta con modificar el *size* del vector. También es posible eliminar elementos.

El diseñador puede especificar el modelo 3D de la unidad (*Transform*), su nombre, sus puntos de ataque, su rango de ataque, su precio, efectos especiales (si es trabajador en lugar de soldado) o su rango de movimiento.

Al ser una serialización de un vector del tipo Unidad, modificar los valores en caso de que un nuevo tipo de variable fuese requerido por el equipo de diseño es fácilmente conseguible.

Y con tan solo cambiar un valor en el editor de Unity, el cambio ya está aplicado *in-game*, por lo que permite agilidad absoluta para los diseñadores.

3.1 Editor de Edificios



De la misma forma podemos aplicar este conocimiento a otros elementos de *gameplay,* como, por ejemplo, edificios, que añaden una nueva capa de profundidad al juego. Desde el editor los diseñadores pueden asignarles un modelado, un nombre, la generación de oro que produce cada uno, el tipo de unidades que puede producir, o si es una condición de victoria.

La versatilidad de editar edificios y unidades desde el propio Inspector de Unity, unido al ponente editor de mapas, cumplen con creces los requisitos básicos para ofrecer unas herramientas de desarrollo decente a los desarrolladores que conformaban el reto de este anteproyecto.

4 Gameplay

El *gameplay* del juego es sencillo. Es un juego *TBS* (*turn based strategy* – estrategia basada por turnos).

Es decir, el juego transcurre con un jugador realizando sus movimientos, y cediendo el control al jugador rival hasta que vuelva a ser su turno.

En este caso, el objetivo del juego es ganar mediante aniquilación del enemigo (destruir todas sus unidades) u conquista de objetivos (conquistar 2 o más edificios que sean considerados Win Condition/ Condiciones de victoria).

Cada turno, el jugador puede realizar una acción con cada una de sus unidades: moverse u realizar una acción especial, siendo esta acción especial atacar en caso de los soldados, y construir en caso de los obreros.

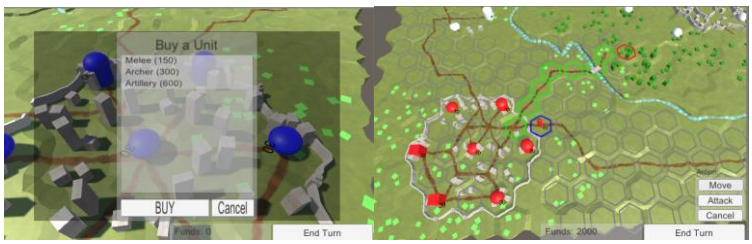
Atacar resta a los puntos de vida de un enemigo los puntos de ataque del soldado atacante. Para poder realizar esta acción, el enemigo debe de estar en el rango de ataque del atacante. Si los puntos de vida de una unidad bajan a 0, muere. Si los puntos de vida de un edificio bajan a 0, se neutraliza, es decir, pasa a ser del bando neutral hasta que sea reconquistada.

Construir permite a una unidad o bien reparar un edificio aliado (recuperándole la misma cantidad de vida como puntos de vida tenga el obrero) u conquistar un edificio neutral (que pasa a tener la misma cantidad de vida que tenía el obrero multiplicada por tres). Para realizar la acción de construcción, es necesario estar al lado del edificio.

Al moverse, las unidades pueden desplazarse dependiendo del movimiento que tengan. Las unidades no pueden atravesar murallas o mar. Además, se ven ralentizadas por bosques, granjas y ciudades, pero son capaces de avanzar más rápidamente a través de caminos.

Los edificios generan una pequeña cantidad de dinero variable al comienzo de cada ronda. Además, algunos de esos edificios permiten reclutar diferentes tipos de unidades.

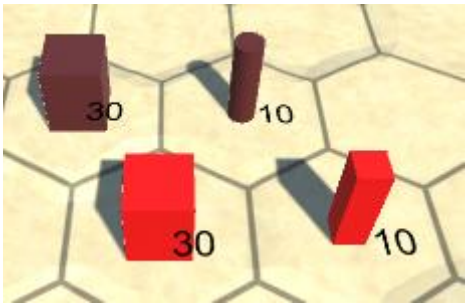
Estrategizar para construir un ejercito mas poderoso que tu rival y generar una mayor ventaja económica, o ir lo más rápido posible para hacerse con los puntos de victoria rápidamente, son las principales formas de ganar en este juego.



4.1.1 Ejemplos de HUD Ingame

El control del *gameplay* está en un *GameObject* intermedio *HexGameUI,* que se encarga de interpretar las acciones recibidas por el jugador desde la HUD, y ejecutarlas realizando los cambios correspondientes a estas acciones en *HexGrid*. *HexGrid* almacena la información de todas las unidades, edificios y casillas, así que por ende también es el encargado de producir efectos de movimiento o ataque, y de que las unidades y edificios interactúen entre sí. Las unidades y edificios tienen sus efectos de interacción, junto a sus atributos específicos, definidos en un script *HexUnit* y *HexBuilding* respectivamente. *HexGrid* también es capaz de comprobar cuando se ha cumplido una condición de victoria cuando *HexGameUI* se lo solicita.

Asimismo, *HexGameUI* es la encargada de indicarle a *HexGameGrid* como colorear las unidades para indicar si dicha unidad ha realizado o no una acción en el actual turno.



4.1.2 Unidades y edificios activos frente unidades y edificios cansados.

HexGameUI no es solo el encargado de ordenar a *HexGrid* que tiene que hacer. También almacena información como el turno del jugador actual, o el número de fondos (*funds*) de los que dispone cada uno de ellos.

También se encarga de mostrar los elementos de la interfaz que el jugador requiere para jugar: por ejemplo, desplegar un menú de acciones cuando el jugador seleccione una unidad, cerrarlo en caso de que el jugador deseleccione dicha unidad u seleccione a otra nueva, indicar de quien es el turno actual o cuantos fondos tiene un jugador, desplegar un menú de compra de unidades a la hora de seleccionar un edificio si este puede producir unidades, impidiendo la compra de unidades que superen el presupuesto del jugador actual, y similares-

Si fuese necesario, sería capaz de gestionar otros tipos de datos como el numero de turnos transcurridos hasta el momento o el tipo de casilla/unidad/edificio seleccionado por el jugador actualmente y mostrarlo por la interface del usuario.

Dicho objeto también se encarga del display de la hud de victoria.

5 Sistema de guardado

El sistema de guardado de este anteproyecto fue diseñado con la continuidad en mente.

Es decir, en caso de que el juego saliese a la luz y requiriese una futura update que afectase a la función de cargado y guardado de mapas, el nuevo programa sería capaz de leer, interpretar y cargar mapas antiguos.

Esto se logra implementando una cabecera o *header* al comienzo de cada fichero de guardado, indicando en que versión se guardo dicho mapa. Si en un futuro se realiza un cambio en el sistema de carga de mapas, tan solo será necesario detectar que el mapa a leer es antiguo, y actuar en consecuencia ante tal situación.

El formato de fichero del sistema de guardado es inventado (.map), y es tan solo un fichero que almacena la información del estado del juego y de cada casilla, unidad y edificio del mapa en binario.

De esta forma, cada fichero pesa una pequeña cantidad de bytes en lugar de un buen puñado de KB. Si es menester realizar una gran cantidad de guardado de mapas, este formato ahorrará mucho espacio. Además, el *BinaryReader* y *BinaryWritter* de C# son increíblemente sencillos y rápidos de usar.

Unido a que la información de las estadísticas de cada unidad/edificio, etc. se almacenan *ingame* en lugar de en memoria, derivamos en un sistema de guardado increíblemente eficiente.

Un sistema de guardado basado en texto, como podría ser JSON o XML, podría consumir mucho espacio o ser problemático a la hora de guardar mapas grandes, de más de 1000 casillas con gran cantidad de información. Al usar un sistema de guardado binario, nos arrebatamos de esos problemas.

Unity nos permite usar “Application.persistentDataPath” para guardar los ficheros en el archivo correspondiente a la memoria de nuestro juego en cualquier sistema operativo en el que nos encontremos.

No obstante, aunque en este trabajo nos limitemos a guardar todos los mapas en este fichero, no es una buena práctica, ya que guardamos mapas que podrían ser parte de un modo campaña del juego o que podrían estar sobrescribiendo sin querer a otro mapa. Usar una carpeta para guardar los mapas creados por diseñadores, otra para los archivos de guardados de los jugadores, y otra para los mapas creados por los jugadores, sería la mejor estrategia.

No obstante, esto requeriría de un tiempo que no poseo en este momento.

6 Problemas y dificultades

Generalmente, desarrollar un juego siempre da problemas inesperados y grandes dolores de cabeza, ya que incluso desarrolladores con experiencia dan por sentado o asumen que funcionalidades básicas serán más fáciles de implementar de lo que realmente son. Por ello se decidió asignar un proyecto que, a priori, pudiese ser lo suficientemente pequeño como para finalizarse en 3 o 4 meses, y tener tiempo para pulir y presentar el proyecto final.

Como era de esperar, estos cálculos fueron erróneos, y sobreestimamos tanto el tiempo que se poseía, así como el tamaño del proyecto.

Si bien es cierto que se han superado los requisitos iniciales de este anteproyecto, falta pulimiento y funcionalidades que habrían supuesto un toque final al trabajo realizado, como una mejor gestión del sistema de guardado de ficheros, una interfaz más lograda, modelado de unidades, edificios y adornos, o mecánicas más interesantes, como generales que aportasen bufos y poderes a cada jugador.

La falta de tiempo por diversos factores durante el segundo cuatrimestre, el caos y ruptura de la rutina causada por la COVID-19, el alocado desarrollo de las prácticas externas y la cuestionable gestión del tiempo por parte del alumno provocó retrasos que sumados a errores y problemas inesperados ralentizaron el proyecto final.

A esto se sumó el que diversos contratiempos producidos en características del juego que a priori se estimaban más rápidas o sencillas de implementar (como el sistema de unidades o la *HUD* de selección de acción de cada unidad), sumados a la cantidad de tiempo drenado en el desarrollo de una herramienta de diseño de mapas aceptable, y la necesidad de testear una a una las funcionalidades añadidas y comprobar que estas no rompiesen ninguna otra.

Todo esto desembocó en un "*mini-crunch*" de varios días antes de entregar el proyecto. En este estado, las últimas implementaciones del proyecto están añadidas con prisas, poco testeo, y con una calidad inferior a las del resto del proyecto, lo cual daña el resultado final, pero permite llegar a la *'deadline'* con un prototipo jugable y cumpliendo todas las funciones requeridas.

Conclusiones y líneas futuras

Hacer un videojuego es mucho más difícil de lo que parece, e incluso sabiéndolo por adelantado, es fácil que un imprevisto arruine una fecha de entrega. Sin embargo, también es increíblemente gratificante y divertido.

Este proyecto ha permitido aprender mucho de Unity3D, ganar algo de experiencia como '*gamedev*,' e indagar en la mejor forma de desarrollar herramientas de diseño y juegos.

El resultado final tiene mucho que mejorar. Especialmente:

- El sistema de guardado de mapas podría dividirse en archivos según se guarde un mapa como mapa jugable para un modo historia/campaña; una partida guardada en medio de una pelea; o sea un mapa creado por un jugador para ser jugado en un futuro.

- La HUD de tanto el editor de mapas como de el apartado jugable requieren de más pulimiento y deberían ofrecer más información de la que dan.

- El sistema de combate es extremadamente simple y podría complicarse y añadir profundidad a este con muy poco esfuerzo (daño dependiente de la vida actual, casillas con defensa, unidades con armadura, etc.).

- Las unidades podrían tener algún tipo de animación de pelea por defecto.

- Incluir alguna *gimmic* de más, como generales que ofreciesen *buffos* a los jugadores, podría dar mucho jugo al juego.

No obstante, el código está bien estructurado y es fácil de leer y editar, por lo que, con el tiempo necesario, realizar las tareas realizadas anteriormente no debería suponer una mayor dificultad.

Summary and Conclusions

Making a video game is much more difficult than it seems, and even knowing that in advance, it is easy for an unforeseen event to ruin a delivery date. However, it is also incredibly rewarding and fun.

We have learned a lot about Unity3D in this project, gained a little experience as game developers, and studied about the best ways on the development of the design tools for a game.

The result has a lot to improve. Especially:

- The map saving system could be divided into files according to whether a map is saved as a playable map for a story/campaign mode; a game saved in the middle of a fight; or a map created by a player to be played in the future.

- The HUD of both the map editor and the playable section require more polishing and should offer more information than what they give.

- The combat system is extremely simple and could be complicated and add depth to it with very little effort (current life dependent damage, defended squares, armored units, etc.).

- The units could have some kind of fight animation by default.

- Including some extra gimmicky, such as generals offering stat buffs to players, could give a lot of depth to the game.

However, the code is well structured and easy to read and edit, so, with the necessary time, performing the tasks done previously should not be a major difficulty.

Presupuesto

El siguiente presupuesto simula el coste del anterior proyecto de haberse realizado con un equipo profesional en un plazo de tiempo de 6 meses:

Tipos	Descripción
Programador senior	1800€/m
QA	1300€/m
Artista 3D	1500€/m
Artista VFX	1650€/m
Música (Encargo)	1500€
Otros *assets*	1500€
Presupuesto final	**2.680.500 €**

Bibliografía

Unity Scripting and Shaders:
Jasper Flick: https://catlikecoding.com/unity/tutorials/

Hexagonal grid reference and implementation guide:
Amit Patel: https://www.redblobgames.com/grids/hexagons/
https://www.redblobgames.com/grids/hexagons/implementation.html

General purpose support:
https://answers.unity.com/index.html
https://stackoverflow.com

Unity scripting documentation:
https://docs.unity3d.com/ScriptReference/

C# .NET documentation:
https://docs.microsoft.com/es-es/dotnet/

Music:
Kevin MacLeod: https://incompetech.com