UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un Asset en Unity para la generación automática de mapas 2D en videojuegos de plataforma

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel-Ionel Bizau

Tutor: Ramón Pascual Mollá Vayá

2020-2021

Resumen

Desarrollo de una herramienta que permita la generación automática de mapas de juegos tipo 2D de plataformas para el motor de videojuegos Unity. La herramienta será configurable para permitir adaptarse a una alta variedad de juegos, permitiendo personalizar tanto la disposición de los elementos interactivos de juego, como de los elementos estéticos. Se utilizarán parámetros de configuración como la longitud media y la dispersión del tamaño de la plataforma, elementos empleados para construirla, distancia mínima y máxima entre plataformas, diferencia de alturas, distribución de recompensas, distribución de elementos de entorno y en general cualquier otro parámetro que permita generar una gran variedad de mapas para juegos de plataformas.

Palabras clave: videojuegos, Unity, juegos de plataformas, generación procedural, Asset Store

Abstract

Development of a tool that allows automatic generation of 2D platform maps for the Unity video game engine. The tool will be configurable to allow adapting to a wide variety of games, allowing the user to customize both the layout of the interactive game elements as well as the aesthetic elements. Configuration parameters will be used such as the average length and dispersion of the platform size, elements used to build it, minimum and maximum distance between platforms, difference in heights, distribution of rewards, distribution of environment elements and in general any other parameter that allows the user to generate a great variety of maps for platform games.

Keywords: video games, Unity, platform games, procedural generation, Asset Store

Tabla de contenidos

Tabla de ilustraciones

1. Introducción

El propósito de la introducción es aportar una visión general de la motivación que ha llevado a la elaboración de este trabajo, así como los objetivos generales del mismo. Luego se listarán los diferentes acrónimos y abreviaturas, así como la estructura del resto del documento.

1.1 Motivación

La industria de los videojuegos está en continuo crecimiento. No ha sido posible acceder a los análisis de mercado directamente debido a su alto coste, sin embargo, los editores de Playtoday[1] recogen información de varias fuentes. Según ellos, en 2020 el mercado global de los videojuegos alcanza unos ciento sesenta y dos mil millones de dólares. Se estima que podría alcanzar unos dos cientos noventa y cinto mil millones en 2026.

Cada vez más son las personas que quieren dedicarse a desarrollar videojuegos de forma independiente (1), sin embargo, el mercado es muy competitivo. En 2015 cada día se publicaban mas de 500 juegos en la plataforma iOS. No solo eso, sino que los desarrolladores *indie* también tienen que competir con las grandes compañías de videojuegos que tienen presupuestos de decenas de millones de dólares y centenares de desarrolladores de varios países trabajando en un solo proyecto.

Teniendo en cuenta lo anterior, los beneficios que se pueden obtener por un desarrollador independiente, sobre todo al principio de su carrera profesional son limitados (1). En muchos casos obligando al desarrollador a buscar una segunda fuente de ingresos y ejercer su desarrollar solo en su tiempo libre, como segundo trabajo. Por estas razones, se ven obligados a publicar sus juegos antes de tiempo, generando un mercado con muchos juegos de baja calidad, haciendo los juegos *indie* menos atractivos para los jugadores.

Realizar este proyecto también tiene una motivación personal. Tanto mi hermana, como yo, tenemos interés en el desarrollo de los videojuegos. Desarrollar este proyecto me permite adquirir los conocimientos necesarios sobre Unity para desarrollar junto con ella un proyecto común. Mi hermana, al ser estudiante de arte se puede encargar de los aspectos gráficos de un videojuego, complementando los conocimientos técnicos que se adquieren mediante este trabajo.

1.2 Objetivos

[1] https://playtoday.co/blog/gaming-industry-statistics/#entire%20global%20gaming%20market

El objetivo de este trabajo es definir los requisitos de un producto que pueda ser útil a los desarrolladores independientes en agilizar el proceso de desarrollo de sus videojuegos, reduciendo así sus costes. Este objetivo se consigue mediante tres puntos principales:

- Crear un *Asset* con el objetivo de publicarlo en la Asset Store de Unity que permita a los desarrolladores crear mapas de juego para sus niveles de forma automática, minimizando el tiempo de desarrollo por nivel.
- Facilitar la expansión de la herramienta por los miembros de la comunidad *indie* mediante la publicación del código en un repositorio de código abierto como GitHub. También se va a prestar especial atención a la calidad del código, para hacer el funcionamiento lo más claro posible, con los comentarios necesarios para facilitar el entendimiento, cuidando la estructura para ser mantenible y redactando los comentarios en inglés.
- Facilitar el aprendizaje y entendimiento a los nuevos desarrolladores de conceptos útiles en el desarrollo de los videojuegos tales como: generación procedural, efecto *parallax*, *pool* de objetos, etc. Para saber más sobre estos términos, en el anexo se encuentran las definiciones de estos y otros términos específicos del área de conocimiento de este trabajo.

Además de definir los requisitos, como parte de este trabajo también se va a implementar un mínimo producto viable que se pueda utilizar para comprobar la validez de los requisitos. De esta forma, en trabajos futuros se pueden modificar los requisitos que sean necesarios, para cumplir con los objetivos del producto.

También, a nivel personal, el objetivo de este trabajo es adquirir los conocimientos necesario de Unity como para participar en el desarrollo de un videojuego 2D.

1.3 Estructura del documento

El resto del documento tiene la siguiente estructura:

En la sección 2 se hará un análisis de los videojuegos de plataformas para conocer las necesidades de los desarrolladores. Después, se podrá ver el estado del arte y cómo este producto se diferencia de otros similares, para luego en la sección 3 entrar en detalle especificando cada uno de los requisitos que debe cumplir el producto final.

Luego, en la sección 4, veremos las diferentes metodologías y herramientas que se van a utilizar en el proceso de desarrollo de la solución.

A raíz de la especificación de los requisitos, en la 5 sección se va a hacer un análisis funcional que dará como resultado el diseño funcional de la solución. Aquí se van a describir los diferentes componentes y como interactúan entre ellos para cumplir con los requisitos especificados.

En la sección 6 se va a hacer un repaso de las tecnologías que se utilizan y por qué han sido elegidas.

La sección 7 servirá para documentar los aspectos relevantes del desarrollo, así como contratiempos y soluciones encontradas. También se va a ver el uso del producto mediante un caso de uso.

En la sección 8 se van a sacar conclusiones y hablar de los posibles trabajos futuros.

En la sección 9 se puede ver un apéndice de vocabulario, mientras que en las secciones 10, 11, 12 y 13 se encuentran anexos que muestran extractos de los ficheros README.md y LICENSE del proyecto.

Para finalizar, en la sección 14 se muestran las referencias utilizadas.

2. Estado del arte

En esta sección se podrán ver el estado del arte, tanto a nivel de tecnologías empleadas en la generación procedural, como un análisis de los videojuegos de plataformas. También se analizarán herramientas similares al trabajo que se desea realizar. La información adquirida mediante al análisis del estado del arte ayudara a encontrar las mejores funcionalidades que el producto final debería tener.

2.1 Generación procedural

La generación procedural de contenido la define Gillian Smith en 2015(2) como el uso de un algoritmo formal para generar contenido que típicamente sería generado por una persona. Veamos a continuación, el estado del arte de la generación procedural:

Según M. Hendrikx et al. (3), cientos de millones de personas juegan juegos de computadora todos los días. Para ellos, el contenido del juego, desde objetos 3D hasta rompecabezas abstractos, juega un papel importante en el entretenimiento. Hasta ahora, el trabajo manual ha asegurado que la calidad y cantidad del contenido del juego coincida con las demandas de la comunidad de jugadores, pero enfrenta nuevos desafíos de escalabilidad debido al crecimiento exponencial durante la última década tanto de la población de jugadores como de los costos de producción. La generación procedural de contenido para juegos (PCG-G) puede abordar estos desafíos automatizando o ayudando en la generación de contenido del juego. PCG-G es difícil, ya que el generador tiene que crear el contenido, satisfacer las limitaciones impuestas por el artista y devolver instancias interesantes para los jugadores.

En su trabajo (3), primero ellos presentan una clasificación completa de seis capas del contenido del juego: *assets*, espacio, sistemas, escenarios, diseño y derivados. En segundo lugar, examinan los métodos utilizados en todo el campo de PCG-G de un gran organismo de investigación. En tercer lugar, asignan los métodos PCG-G a las capas de contenido del juego; resulta que muchos de los métodos utilizados para generar contenido de juegos a partir de una capa se pueden utilizar para generar contenido a

partir de otra. También analizan el uso de métodos en la práctica, es decir, en juegos comerciales o prototipos. En cuarto y último lugar, discuten varias direcciones para la investigación futura en PCG-G.

M. Hendrikx et al. (3) concluyen que han encontrado una relación inversa entre la posición de la capa en la clasificación de contenido y la madurez de las técnicas de generación de procedimientos de esta. También han descubierto que muchos juegos reales usan técnicas PCG-G, pero ese uso a menudo se limita a un tipo particular de contenido del juego. En la Ilustración 1 podemos ver un análisis de diferentes juegos y el uso que emplean de la generación procedural de contenido en las diferentes capas.

Games (with year of release)	Game Bits	Game space	Game Systems	Game Scenarios
Borderlands (2009)	x			
Diablo I (2000)		x		
Diablo II (2008)		x		x
Dwarf Fortress (2006)		x	x	x
Elder Scrolls IV: Oblivion (2007)	x			
Elder Scrolls V: Skyrim (2011)				x
Elite (1984)		x	x	x
EVE Online (2003)	x	x		x
Facade (2005)				x
FreeCiv and Civilization IV (2004)		x		
Fuel (2009)		x		
Gears of War 2 (2008)	x			
Left4Dead (2008)				x
.kkrieger (2004)	x			
Minecraft (2009)		x	x	
Noctis (2002)		x		
RoboBlitz (2006)	x			
Realm of the Mad God (2010)	x			
Rogue (1980)		x		x
Spelunky (2008)	x	x		x
Spore (2008)	x	x		
Torchlight (2009)		x		
X-Com: UFO Defense (1994)		x		

Ilustración 1: Uso de técnicas PCG-G en videojuegos (3)

Otro trabajo que ha llamado la atención es el de A. Lagae et al. (4). Las funciones de ruido procedural se utilizan ampliamente en gráficos por computadora, desde la reproducción fuera de línea en la producción de películas hasta los videojuegos interactivos. La capacidad de agregar detalles complejos e intrincados con poca memoria y coste de creación es uno de sus principales atractivos. El objetivo de este trabajo es proporcionar un valioso punto de entrada en el campo de las funciones de ruido procedural, así como una visión completa del campo para el lector informado. En este informe, se cubren las funciones de ruido procedural en todos sus aspectos. Se describen los avances recientes en la investigación sobre este tema, discutiendo y comparando métodos recientes y bien establecidos.

Una de las funciones para generar ruido procedural de la que hablan A. Lagae et al. es el famoso ruido Perlin (5). En esta, el ruido se crea utilizando primero una función pseudoaleatoria para generar una serie de valores que luego se interpolan en ruido coherente. A continuación, se combinan varias capas de este ruido coherente utilizando diferentes proporciones para crear una textura de aspecto natural con detalles fractales.

Desde su introducción hace más de dos décadas, el ruido Perlin ha encontrado un amplio uso en gráficos. Este es rápido y simple, y ha seguido siendo el caballo de batalla de la industria.

Ginting et al. (6) muestran un caso de uso del ruido Perlin en la industria de los videojuegos. El rápido desarrollo de la tecnología ha convertido al juego en uno de los medios de entretenimiento más populares, desde niños hasta adultos. Los juegos de tipo Endless Runner se pueden interpretar en dos elementos. Personajes del jugador que no pueden detenerse y la pista del juego que continúa infinitamente. El algoritmo Perlin Noise se usa para crear un generador de pistas que continúa ejecutándose aleatoriamente. Los generadores de pistas procedurales pueden producir pistas pseudo-interminables, que se utilizan comúnmente en los juegos de tipo Endless Runner. Cada pista se genera combinando una serie de bloques instalados entre sí, donde cada bloque experimenta un cambio de deformación frente a él. De modo que el siguiente bloque que se hará de acuerdo con el que está detrás de él, repitiendo así el proceso, dará como resultado pistas ilimitadas.

2.2 Videojuegos de plataformas

Para poder definir los requisitos del sistema que se va a crear, es importante conocer las necesidades del usuario final. Debido a que no hay acceso a desarrolladores de videojuegos para poder preguntarlos directamente, hay que encontrar otra manera de descubrir sus necesidades.

Por esta razón a continuación, se va a analizar las principales características de los videojuegos de plataformas. Esta tarea se va a realizar mediante el análisis de dos videojuegos que han ayudado a definir el género en los años ochenta, Donkey Kong y Super Mario Bros y dos juegos más actuales, aclamados por los críticos, Limbo y Celeste.

Donkey Kong

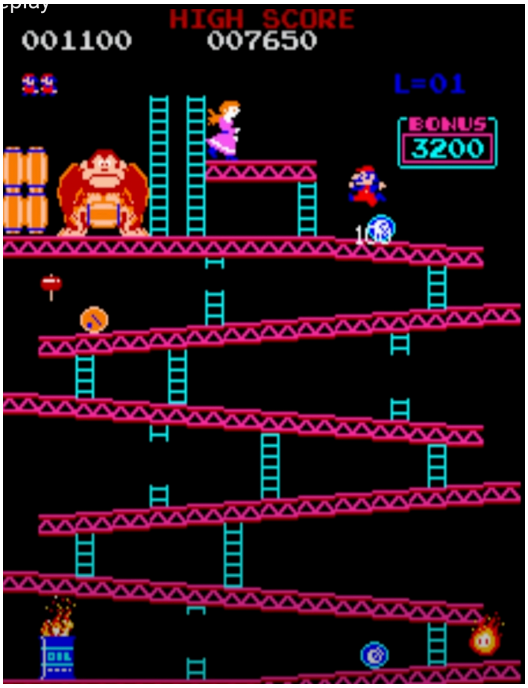

Ilustración 2: Jumpman saltando en Donkey Kong

Donkey Kong es un juego para maquinas de árcade publicado por Nintendo en el año 1981. Está dividido en cuatro niveles, con mapa estático, sin usar técnicas de desplazamiento,

Aunque las plataformas se usaron antes en los videojuegos, se considera el primer juego de plataformas del género, debido a la introducción de una nueva mecánica de juego. Por primera vez, había que hacer uso de saltos para poder ganar. Esto incluye saltos por encima de obstáculos, tal como se puede ver en la Ilustración 2, y saltos calculados a otras plataformas para evitar obstáculos.

Super Mario Bros.

Super Mario Bros. es un juego de plataformas publicado también por Nintendo, en el año 1985 para las consolas Famicom y Nintendo Entertainment System (NES). Es uno de los juegos más famosos del mundo, y uno de los juegos mas influyentes del género.

Este juego consiste en una serie de niveles con una dificultad que sube progresivamente. En cada nivel, el personaje principal, Mario, tiene que pasar por usa serie de obstáculos y enfrentarse a diferentes enemigos para llegar al final del nivel. Mario puede saltar encima de los enemigos para matarlos, o usar un disparo que puede conseguir a través de unos potenciadores que se pueden encontrar en los niveles. En la Ilustración 3 se muestra una imagen del primer nivel, donde el jugador se familiariza con los elementos de juego y los controles de Mario. Al final del nivel se muestran los puntos totales que se consiguen matando enemigos y acabándolo de la forma mas rápida posible.

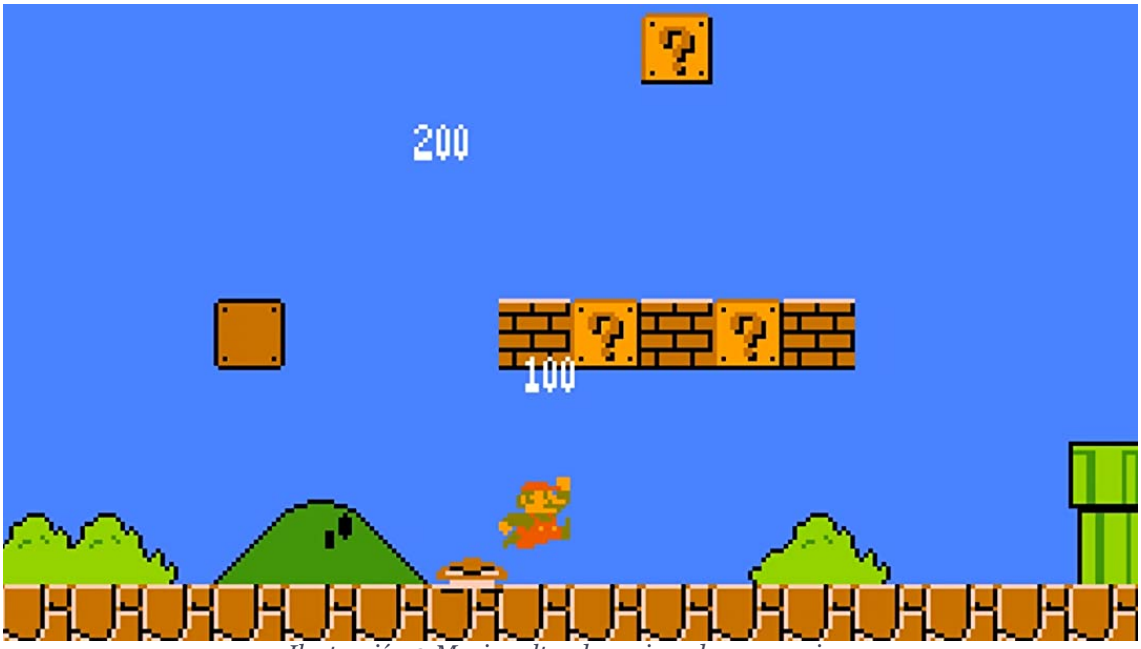

Ilustración 3:Mario saltando encima de un enemigo

El elemento que Super Mario Bros. ha popularizado es el desplazamiento lateral del mapa. A medida que el personaje se mueve hacia la derecha, el mapa se va desplazando, hasta llegar al punto final. Con esta mecánica de juego se completa el género de juegos de plataforma, que se mantiene inalterado en líneas generales hasta el día de hoy.

Celeste

Hoy en día, los juegos de plataforma son muy populares entre los desarrolladores independientes, debido a su relativa facilidad de desarrollo. Con mucha creatividad y pocos recursos se pueden conseguir juegos que no tienen nada que envidiar a los juegos AAA.

Igual que todos los juegos de esta lista, Celeste es un juego de plataformas. Fue desarrollado por un equipo de siete personas en un periodo de dos años, aunque el prototipo inicial donde se marcaron las bases del juego fue creado en tan solo cuatro días por Noel Berry y Matt Thorson en un evento de *Game Jam*. Fue publicado en 2018 para las principales plataformas, Play Station, Xbox, Nintendo, Windows, MacOS, Linux y Stadia. Actualmente se puede adquirir en la tienda online Steam[2].

El juego consiste en una serie de niveles donde el personaje principal, Madeline, se enfrenta a diferentes obstáculos con el objetivo de llegar finalmente a la cima de la montaña Celeste.

Los controles son sencillos, correr, saltar, escalar e impulso en el aire. A diferencia de Mario, no tiene ningún tipo de ataque ya que no se enfrenta a los enemigos de forma

[2] https://store.steampowered.com/app/504230/Celeste/

directa, sino que intenta evadirlos. En la Ilustración 4 se puede ver a Madeline impulsándose desde una pared para avanzar.

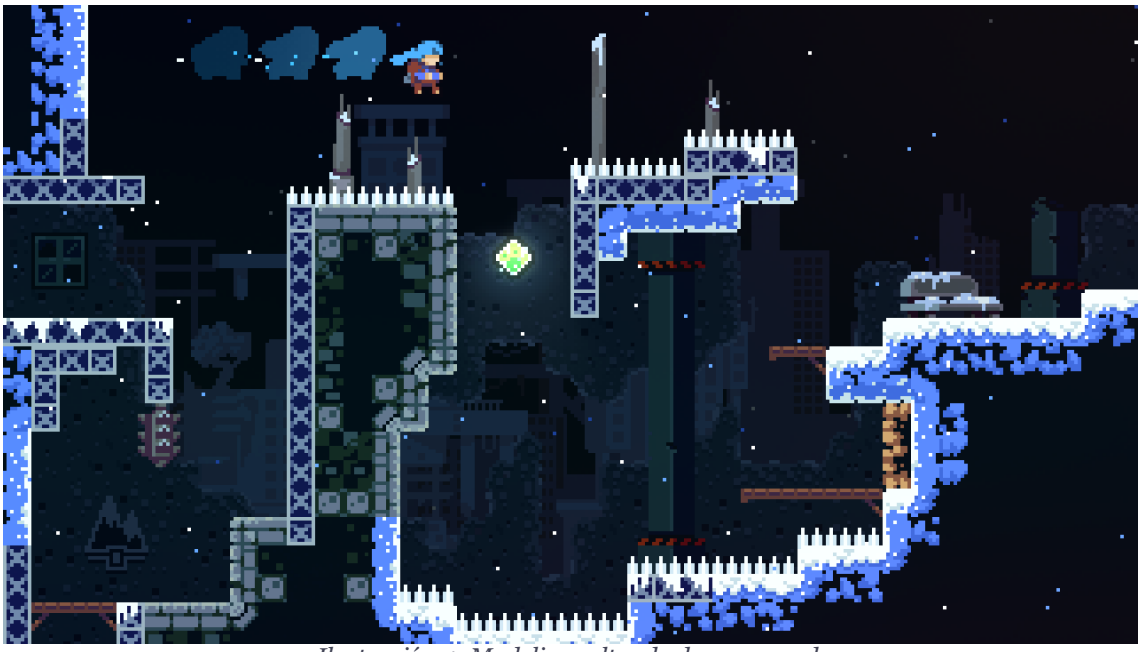

Ilustración 4: Madeline saltando de una pared

En cuanto al control de la cámara, presenta una mezcla de los dos juegos anteriores, Donkey Kong y Super Mario Bros. El mapa está dividido en diferentes segmentos, y la cámara no se desplaza al siguiente hasta no acabar el anterior. Sin embargo, para los segmentos mas grandes, si que hay desplazamiento de cámara para seguir a Madeline.

Limbo

Limbo es un juego de plataformas publicado en 2011 por una compañía independiente danesa, Playdead[3]. Sigue la historia de un niño, cuyo nombre se desconoce y que está en búsqueda de su hermana. A nivel artístico se diferencia claramente, no solo de los otros juegos de esta lista, sino del resto de los juegos de este género. El estilo artístico es monocromo, con una estética sobria que se consigue a través de las sombras, efectos de humo y desenfoques. El efecto *parallax* contribuye a esta estética, incluyendo una capa de vegetación y diferentes elementos desenfocados por delante del plano de juego, que asienta la cámara en el mundo del juego. Esto te hace sentir como si estuvieras viendo al personaje principal por los ojos de alguien quien le persigue.

Las mecánicas de juego son muy sencillas. El niño que se puede controlar puede desplazarse en las dos direcciones, saltar, subir y bajar escaleras e interactuar con algunos elementos del mundo, empujando o arrastrándolos.

[3] https://store.steampowered.com/app/48000/LIMBO/

Ilustración 5:Limbo

A diferencia de Super Mario Bros. Y Celeste, el personaje principal de Limbo se caracteriza por ser mas lento en sus movimientos. Aquí, para llegar no es tan importante la destreza en los movimientos, sino resolver de forma correcta los diferentes puzles con los que el jugador se puede encontrar. Debido a esto, como se puede ver en la ilustración, el mapa de Limbo es mucho mas plano que el de Celeste, siendo mas bien similar a Super Mario Bros.

Características comunes

En la primera parte de esta sección se ha familiarizado con cuatro juegos diferentes que pertenecen al genero de juegos de plataforma. Ahora, se realizará un análisis de las características comunes de estos juegos con el propósito de hacerse una idea de las características que debería tener el producto.

En primer lugar, en cuanto el tamaño del mapa se puede ver que este es muy variado, cambiando incluso en el mismo juego, de un nivel a otro. Por lo tanto, es muy importante que el mapa generado por el proyecto que se va a desarrollar pueda tener un tamaño configurable.

En segundo lugar, un elemento común a todos, como bien indica el propio nombre del género, es la existencia de las plataformas. Estas plataformas son variadas en su tamaño y en muchas ocasiones tienen escaleras para acceder a ellas. El producto final que se pretende desarrollar debe implementar la generación de las plataformas.

Otro elemento común, en este caso solo a los últimos tres juegos presentados, es la presencia de las cuevas. En algunos casos, ya sea durante un nivel entero, o solo para una

sección de un nivel, el personaje se tiene que aventurar en una sección subterránea del mapa. Por lo tanto, es recomendable que el producto de este trabajo sea capaz de generar cuevas.

En cuanto al suelo, en todos los casos, con la excepción de Celeste donde se justifica por la alta movilidad del personaje, es la parte predominante del mapa. Es la zona donde el jugador se siente más cómodo, y donde puede tomar un descanso, por así decirlo, entre diferentes plataformas. Solo se ve interrumpido en ocasiones por algunas grietas, que el jugador tiene que evitar, saltando por encima, o usando alguna plataforma u otro elemento de juego. El producto final debe tener un suelo que sea configurable mediante varios parámetros tales como altura o suavidad.

Por ultimo se hablará de los elementos de juego. Todos los juegos presentan una diversidad muy alta de elementos específicos, Hay elementos de diferentes tipos: enemigos, *power-ups,* objetos interactivos, trampas etc. Debido a esta diversidad tan alta, es imposible hacer una herramienta útil en el proceso creativo de cualquier desarrollador. Por lo tanto, la herramienta tiene que estar preparada para insertar de forma automática y con un alto grado de configuración, elementos específicos del proyecto en el que se va a utilizar. De esta manera, el usuario no se vera limitado por los tipos específicos ofrecidos por el producto, sino que podrá generar los suyos sin limitar su creatividad.

2.3 Trabajos similares

Para seguir con la investigación que permita desarrollar los requisitos, se analizarán las diferentes herramientas similares ya publicadas en la Asset Store, presentando sus principales ventajas e inconvenientes. El análisis de cada una se basa en la descripción publicada y en las opiniones de los usuarios.

Map Generation Framework GenX 2D

Publicado por Anton Vinogradov en 2018, GenX 2D[4] es un *framework* que permite al usuario generar mapas de forma procedural con muy poco código. Permite generar mapas de terreno, cuevas, mazmorras y mas elementos. Está publicado en la Asset Store por un precio de €13.40. En la Ilustración 6 se puede ver un ejemplo de mapa generado con esta herramienta.

4 https://assetstore.unity.com/packages/tools/terrain/map-generation-framework-genx-2d-84131#description

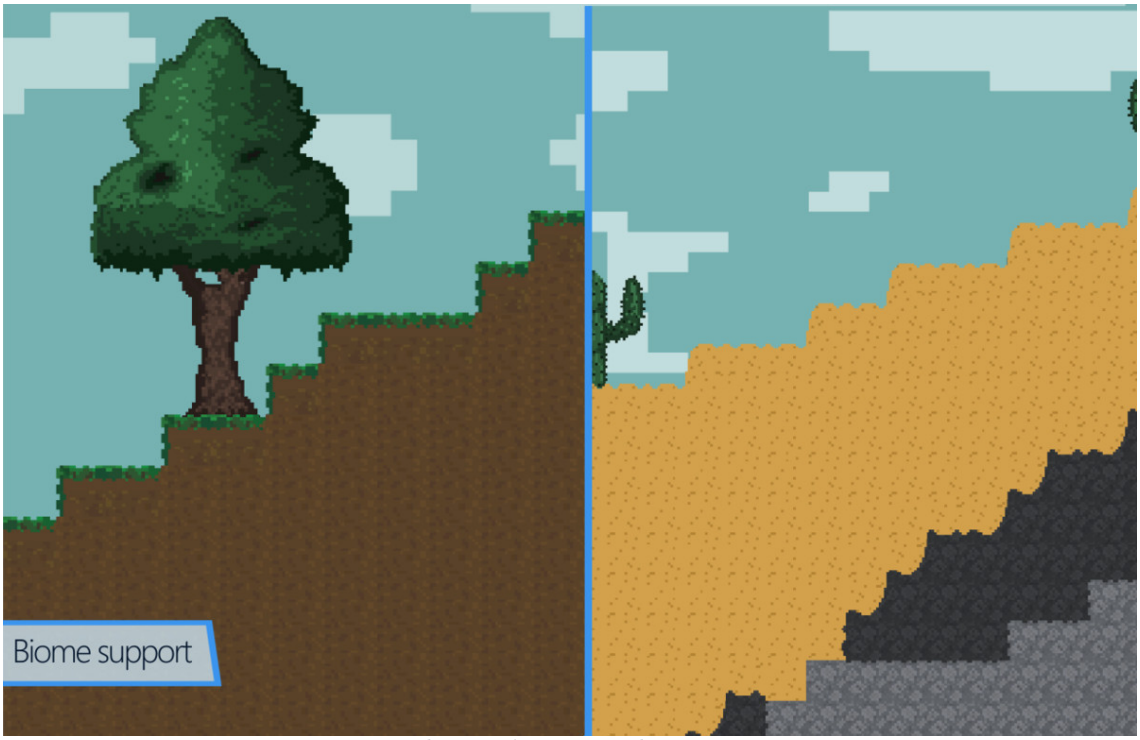

Ilustración 6: Ejemplo GenX

Ventajas:

- Dispone de varias herramientas de configuración
- Permite generación de cuevas
- Permite generación de mazmorras

Inconvenientes:

- No usa Unity Tilemap. Esto lo hace difícil de integrar en proyectos existentes
- Poca documentación. Una parte fundamental de cualquier proyecto. Dificulta mucho su uso.
- El mapa resultante no usa un *collider* compuesto. Esto puede resultar en problemas de rendimiento y errores inesperados en colisiones de objetos.
- Sin soporte. Sin ninguna actualización desde 2018. Esto hace que no sea apta para proyectos grandes que dependan de esta herramienta.

Voxel Terrain 2D

Voxel[5] es una herramienta que permite editar fácilmente de forma visual mapas de tipo cuevas de tierra, formadas por varias capas. Permite usar diferentes materiales, escalas, etc. El precio actual es de €8.93. En la ilustración 7 se observa un ejemplo de lo que se podría conseguir con esta herramienta.

[5] https://assetstore.unity.com/packages/tools/modeling/voxel-terrain-2d-18882

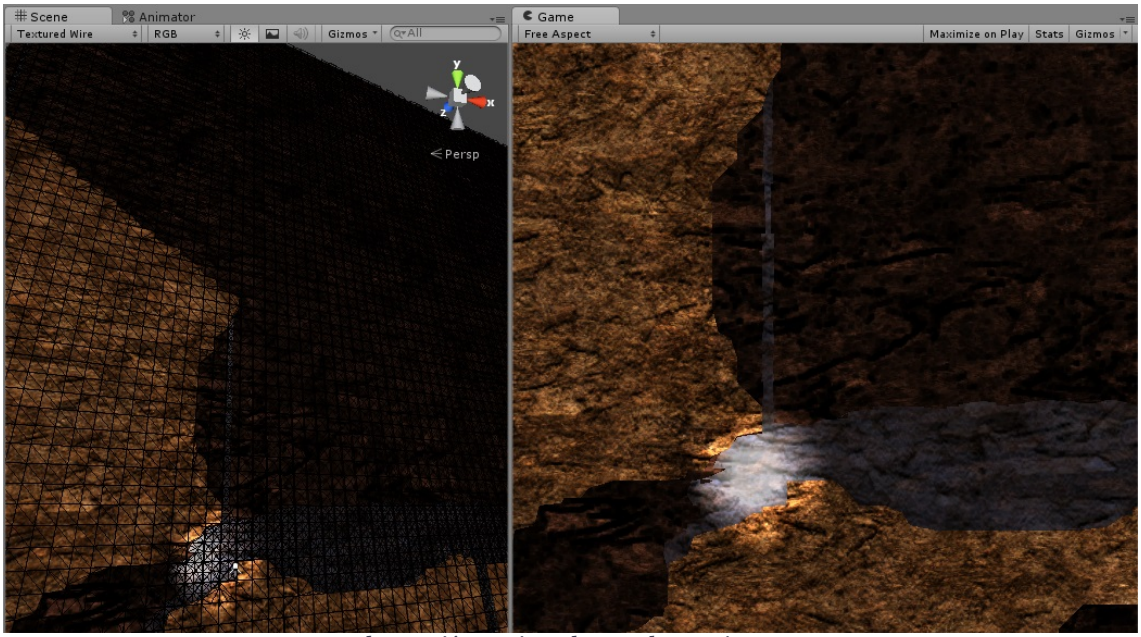

Ilustración 7:Ejemplo Voxel Terrain 2D

Ventajas:

- Fácil de utilizar.
- Permite generar agua que interactúa con el terreno y cualquier otro elemento. También permite crear ríos.
- Permite crear terreno destructible.
- Soporte para efectos físicos en 2D y 3D.

Inconvenientes:

- No permite generación procedural del mapa entero. Esto afecta al tiempo de desarrollo
- Sin actualizaciones desde el 2015, hace más de 6 años. No es apta para proyectos más grandes.

Terrainify 2D

Terrainify[6] es una herramienta que permite generar mapas 2D a base de semillas y *ruido Perlin*. A diferencia de las dos herramientas anteriores, esta es gratuita. En la ilustración 8 se puede ver que se podría conseguir usando esta herramienta.

[6] https://assetstore.unity.com/packages/tools/terrain/terrainify-2d-139257

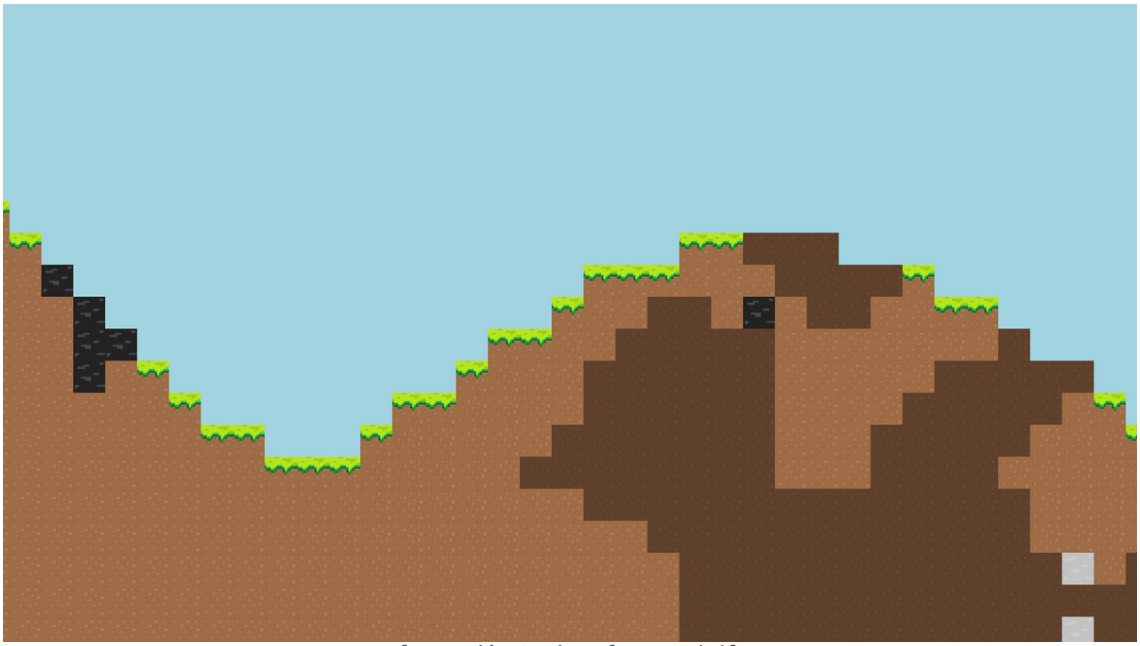

Ilustración 8:Ejemplo Terrainify

Ventajas:

- Utiliza Unity Tilemap.
- Permite cargar y descargar elementos fuera de vista de forma dinámica
- Fácil de usar
- Buena documentación

Inconvenientes:

- No permite generar estructuras o vegetación
- Aunque se realiza una carga y descarga dinámica de los elementos, estos no se guardan y se tienen que generar de nuevo cada vez. Esto tiene un impacto sobre el rendimiento.

Comparativa

A continuación, se podrá ver una comparativa entre los diferentes productos similares, publicados en la Asset Store.

Tabla 1: comparativa de las características de productos similares y este proyecto

	Gen X 2D	**Voxel Terrain 2D**	**Terrainify 2D**	**Este proyecto**
Usa Unity Tilemap	NO	NO	SI	SI
Generación procedural	SI	NO	SI	SI
Edición manual	NO	SI	NO	NO

Simulación de agua	NO	SI	NO	NO
Construcción de cuevas	SI	SI	NO	SI
Construcción de mazmorras	SI	NO	NO	NO
Actualizaciones frecuentes	NO	NO	NO	SI
Carga Dinámica	SI	NO	SI	SI
Carga Dinámica con persistencia	NO	NO	NO	SI

Las características que se han decidido implementar para esta herramienta se han elegido en base a la información presentada en toda esta sección. Se escogieron o las características mas importantes para la generación de mapas para juegos de plataformas en base a la información presentada más atrás. Otras características se han elegido en base al análisis de las opiniones de las herramientas similares ya existentes presentadas más atrás. En la siguiente sección se detallarán todas las características.

3. Especificación de requisitos

Esta sección utiliza como base el estándar IEEE 830 (7) para la especificación de requisitos de software, omitiendo los apartados que no aplican o que ya han sido tratados en otras secciones de este documento.

A continuación, se van a describir todos los requisitos que se contemplan para una herramienta de este tipo, sin embargo, el trabajo de implementarlos todos queda fuera del alcance de un trabajo de fin de grado. Por ello, en una sección posterior se elegirán los requisitos que formarán parte del mínimo producto viable que se va a implementar.

3.1 Introducción

3.1.1 Propósito

El propósito de esta sección es especificar el conjunto de requisitos, tanto funcionales como no funcionales, del producto a desarrollar. Esta sección servirá como base para realizar el análisis funcional que dará como resultado el diseño funcional del sistema.

3.1.2 Alcance

El producto que se desarrolla en el presente TFG se denomina AMG2D, Automatic Map Generator 2D. Está constituido por una serie de *scripts* escritos en el lenguaje de programación C# y una escena de ejemplo. Todo esto, junto con la documentación de su uso será publicado en un repositorio público de control de versiones. En trabajos posteriores se pretende completar el proyecto con más funcionalidad y publicar la herramienta como un Asset en la Asset Store de Unity.

El propósito de la herramienta es permitir la generación automática de mapas, constituidas por diferentes tipos de elementos, como por ejemplo cuevas, plataformas y cualquier tipo de objeto especifico del proyecto en el que se usa. El mapa generado no tendrá terreno destructible y tampoco implementará ningún tipo de lógica especifica de juego, como por ejemplo condición de fin de juego, sistema de puntos etc.

El objetivo es tener una herramienta que sirva para ahorrar tiempo de desarrollo a un usuario de Unity con un proyecto claro en la mente y también que sirva como fuente de ideas a un usuario principiante que quiera aprender sobre Unity, programación o generación procedural.

3.2 Descripción general

3.2.1 Perspectiva

La herramienta AMG2D no constituye una aplicación como tal, sino que se trata de un recurso que cualquier usuario de Unity pueda descargar e integrar en su proyecto, ya sea desde el repositorio de control de versiones, o desde la Asset Store de Unity. La interacción del usuario con la herramienta se realiza mediante unos parámetros de configuración. Estos parámetros se pueden controlar tanto desde el propio interfaz de Unity, mediante la ventana *Inspector,* como mediante el uso de unos ficheros de configuración almacenados localmente.

3.2.1.1 Interfaces de usuario

El interfaz del sistema con el usuario se realiza mediante la ventana *Inspector* de Unity. Los requisitos de este interfaz son los siguientes:

- Todos los parámetros de configuración especificados en los diferentes requisitos funcionales se tienen que poder alterar mediante este interfaz.
- Los parámetros tienen que estar definidos para poder ser modificados de la manera más fácil posible. Por ejemplo:

    - Las propiedades cuyo tipo es un enumerado se tienen que modificar mediante un botón desplegable.
    - Las propiedades de tipo *bool* se tienen que modificar mediante un *CheckBox*.
    - Los parámetros de tipo numérico con un rango finito se tienen que poder modificar mediante una barra de desplazamiento.
- Al poner en puntero encima de cualquiera de los parámetros tiene que aparecer una ventana de tipo burbuja con una breve descripción del parámetro.
- Los parámetros tienen que estar agrupados por los diferentes requisitos funcionales a los que se corresponden.

3.2.1.2 Interfaces software

El producto tiene que interactuar con el propio motor Unity para poder mostrar los diferentes campos tal como se ha especificado anteriormente. Para conseguir esto se recomienda el uso de un objeto de tipo *CustomEditor*[7].

3.2.2 Funcionalidad

En cuanto a las funcionalidades de AMG2D tienen que ser las siguientes:

- Generación de suelo. El sistema tiene que ser capaz de generar suelo de una forma aparentemente aleatoria pero controlada por el usuario.
- Generación de plataformas. No se puede hablar de un juego de plataformas sin las plataformas. Por eso es fundamental que el sistema fuera capaz de generar plataformas de una forma controlada.
- Generación de cuevas. Aunque no tan crítico como la generación de plataformas, la generación de cuevas le añade al sistema un punto de versatilidad que le permite diferenciarse de lo sistemas similares disponibles en la Asset Store.
- Desplazamiento infinito. Este es un punto muy importante. Los mapas finitos, hasta un cierto tamaño se pueden crear de forma manual, sin el uso de esta herramienta o similares. Sin embargo, no se puede crear un mapa infinito sin algún tipo de generación procedural.
- Control del fondo del mapa. Aunque no es una parte critica, ya que se podría crear externamente e integrar con el sistema, esta función le aporta algo adicional para diferenciarse de los sistemas similares.

Cada uno de estos requisitos y algunos más se describirán en detalle en la sección de requisitos específicos.

[7] https://docs.unity3d.com/Manual/editor-CustomEditors.html

3.2.3 Características de los usuarios

El producto va a ser utilizado de forma directa por los desarrolladores de videojuegos que incluirán el Asset en sus proyectos.

También se podría considerar como usuarios al jugador final del proyecto creado con esta herramienta. Para el, solo van a tener un uso directo los requisitos no funcionales tales como los de rendimiento y fiabilidad.

Tipo de usuario	Desarrollador avanzado
Experiencia técnica	Media/Alta
Actividades	Usar la herramienta para maximizar su tiempo. Alterar la herramienta para ajustarla a sus necesitadas.

Tipo de usuario	Desarrollador aprendiz
Experiencia técnica	Baja/Media
Actividades	Usar la herramienta para aprender y generar ideas para proyectos.

Tipo de usuario	Jugador del proyecto final
Experiencia técnica	Irrelevante
Actividades	Instalar el juego final en su dispositivo. Consumir el videojuego con fines de entretenimiento.

Existen potencialmente dos tipos de desarrolladores que podrían estar interesados en usar este tipo de herramienta. Pon un lado está el usuario avanzado en la programación y en el uso de Unity. Este usuario busca maneras de maximizar el rendimiento de su tiempo y está interesado en usar esta herramienta como base para luego adaptarla o ampliarla a sus necesidades. Para el, es importante que la arquitectura sea clara y el código bien documentado y fácil de ampliar.

Por otro lado, está el desarrollador inexperto, que se está iniciando en el uso de Unity, la programación o en la generación procedural. Este usuario busca una herramienta fácil de utilizar. Para él es importante que las funcionalidades implementadas estén claras y que variar los diferentes parámetros tenga un efecto claro y controlado sobre el mapa generado. Para este usuario también es importante la documentación del código, ya que le puede servir como material de aprendizaje.

3.2.4 Restricciones

El producto tiene que cumplir con las pautas y directrices[8] de Unity para permitir la publicación en la Asset Store.

El resto de las restricciones vienen dadas por el propio motor de videojuegos Unity, de cual este producto va a ser parte integrada:

- El lenguaje de desarrollo tiene que ser C#.
- El producto no puede disponer de interfaz propio como tal, sino que usara el interfaz que Unity ofrece a los scripts mediante la ventana *Inspector*.
- La implementación de la solución tiene que ser clara y documentada para permitir ampliaciones.
- El producto no puede utilizar librerías y dependencias externas que no son disponibles en el propio Unity.

3.2.5 Suposiciones y dependencias

Todos los requisitos especificados en el presente documento se han redactado teniendo en cuenta las funcionalidades y características de la versión 2020.3 LTS de Unity. Cualquier cambio en la funcionalidad de Unity en las versiones posteriores pueden ocurrir en un cambio de las especificaciones de este producto.

3.2.6 Evolución previsible

Se espera que en el futuro el producto se amplíe con nuevas funcionalidades, aun no conocidas. Es imposible contemplar todas las posibles mejoras para dejar el sistema preparado para futuras ampliaciones. Sin embargo, es importante que el diseño del sistema permita ampliaciones con funciones similares a las descritas en los requisitos específicos de una manera que no requiera refactorización de los requisitos ya implementados.

3.3 Requisitos específicos

A continuación, se detallan los distintos requisitos específicos. Los nombres de los diferentes parámetros y configuraciones de los requisitos funcionales no son finales. Para la versión final es necesario usar nombres en inglés para permitir la publicación en la Asset Store.

[8] https://unity3d.com/asset-store/sell-assets/submission-guidelines

3.3.1 Requisitos funcionales

ID del requisito	*RF01*
Nombre	**Mapas utilizando Unity *Tilemaps***
Descripción	El mapa generado por el sistema tiene que usar como base los *Tilemaps* de Unity. Por lo tanto, el sistema de celdas será cuadricular.

ID del requisito	*RF02*
Nombre	**Guardado de configuración**
Descripción	Todos los parámetros de configuración descritos en los requisitos funcionales que hay a continuación tienen que estar disponibles para su modificación en la ventana del *Inspector*. De mismo modo, también tienen que estar disponibles para cargar mediante un fichero JSON. Este fichero se tiene que poder guardar y cargar mediante unos botones de Carga y Guardado en el interfaz del *Inspector*. Lista de requisitos funcionales que requieren configuración: RF03, RF04, RF05, RF06, RF07, RF08, RF09,

ID del requisito	*RF03*
Nombre	**Limites y posición del mapa**

Descripción	Antes de poder popular el mapa con distintos elementos se tiene que definir su tamaño y su posición. Con este fin se crean los siguientes parámetros:
	• **Altura**. Número entero, expresado en *Tiles*. Representa la altura máxima del mapa.
	• **Anchura**. Número entero, expresado en *Tiles*. Representa la anchura máxima del mapa. Configurar una altura más grande que la anchura resultará en un mapa que se genera de forma vertical, mientras que configurando una altura mas grande que la anchura generará un mapa horizontal.
	• **Dirección**. Dirección en la que se genera el mapa con respecto al punto inicial.
	  ○ **Positiva**. Si el mapa es horizontal, todo el mapa estará colocado a la derecha del punto inicial. Si el mapa es vertical, todo el mapa estará colocado por debajo del punto inicial.
	  ○ **Negativa**. Si el mapa es horizontal, todo el mapa estará colocado a la izquierda del punto inicial. Si el mapa es vertical, todo el mapa estará colocado por debajo del punto inicial.

ID del requisito	*RF04*
Nombre	**Carga dinámica**

Descripción	Para poder limitar el uso de memoria, los mapas generados se tienen que cargar y descargar de forma dinámica, en función de la posición instantánea de la cámara principal.
	
	Se denomina segmento a un grupo de *Tiles* que se generan conjuntamente. Los segmentos que puedan aparecer en la cámara se tienen que generar siempre. Si el mapa es horizontal, es decir mayor ancho que alto, los segmentos serán horizontales. Si el mapa generado es vertical, es decir mayor alto que ancho, los segmentos serán verticales.
	
	Esta función debe tener los siguientes parámetros de configuración:
	• **Habilitar/deshabilitar**. Un parámetro que especifique si esta función está habilitada o no.
	• **Tamaño de segmento**. Medido en *Tiles*.
	• **Numero de segmentos precargados**.
	  ○ Este parámetro determinará cuantos segmentos por delante de la cámara se cargarán en memoria.
	  ○ Empieza a contar a partir del último segmento visible en la cámara principal.
	  ○ Los segmentos más lejanos que este número se descargarán de memoria automáticamente.
	Es importante que la acción de cargar y descargar de memoria a los distintos elementos no tenga un efecto visible para el jugador. Por lo tanto, es necesario utilizar algún tipo de semilla para la generación que fuera siempre la misa, o algún sistema de retención del estado actual.

ID del requisito	*RF05*
Nombre	**Desplazamiento infinito**

Descripción	El mapa generado debe permitir la opción de desplazamiento infinito.
	
	Para activar esta opción se introduce un nuevo parámetro:
	• **Desplazamiento infinito**. Este parámetro permite habilitar el modo de desplazamiento infinito. Tiene las siguientes posibles opciones, pudiéndose activar solo una de ellas en un momento dado:
	  ○ **Deshabilitado**. El desplazamiento infinito está deshabilitado. El mapa tendrá el tamaño máximo especificado por los parámetros de altura y anchura definidos por el requisito RF03
	  ○ **Horizontal**. Se ignorará el parámetro de anchura máxima. En función del valor del parámetro de dirección especificado en el RF03, el mapa se extenderá de forma infinita hacia la izquierda o hacia la derecha.
	  ○ **Vertical**. Se ignorará el parámetro de altura máxima. En función del valor del parámetro de dirección especificado en el RF03, el mapa se extenderá de forma infinita hacia arriba o hacia abajo.

La idea es poder utilizar diferentes combinaciones de modos de desplazamiento infinito y dirección para generar diferentes tipos de mapas. Por ejemplo, con un desplazamiento vertical negativo se generaría una serie de cuevas (RF07) que se extienden hacia abajo infinitamente, mientras que con una dirección positiva se generaría una serie de plataformas (RF08) que suben hacia arriba infinitamente.

Activar esta función necesita la activación de la carga dinámica. En el caso en el que la carga dinámica esté desactivada, se activará de forma automática con el tamaño de segmento igual el número de segmentos que caben en la imagen de la cámara principal y con el número de segmentos precargados a uno.

ID del requisito	*RF06*
Nombre	**Configuración de suelo**
Descripción	Es necesario que el sistema permita configurar diferentes parámetros referentes a la altura del suelo. Estos parámetros tienen que incluir los siguientes: • **Altura media**. Número entero expresado en *Tiles*. Este será el punto medio de altura a partir de cual, mediante variación aparentemente aleatoria, el nivel subirá y bajará. • **Suavidad**. Numero entero que representa la suavidad del terreno generado. Cuanto mayor este número, menor será la inclinación de las rampas de subida y bajada que se generan. • **Modo solo tierra**. Parámetro de tipo *bool*. Habilitando este parámetro todo el mapa estará por debajo del nivel del suelo. Útil para generar mapas de juego usando solo cuevas interconectadas, tal como se especifican en el requisito RF07.

ID del requisito	*RF07*
Nombre	**Generación de cuevas**
Descripción	El sistema debe permitir la generación automática de cuevas. Está generación tiene que estar gobernada por los siguientes parámetros de configuración: • **Tamaño medio de cuevas**. Este parámetro será un numero entero. No se especifica rango, ni unidad, dejando la elección al gusto del programador, sin embargo, el efecto de cambiar este parámetro tiene que ser uno lineal. Es decir, doblando el valor, se tiene que doblar el tamaño aparente de las cuevas. • **Densidad de cuevas**. Con este parámetro se modifica la densidad, o el número de cuevas que aparecen en una determinada zona. Igual al parámetro anterior, este parámetro será un numero entero, con comportamiento lineal. Fijando este parámetro en cero no se generarán cuevas. • Nivel de interconexión de las cuevas. Este parámetro debe tener 3 posibles valores: ○ **Bajo**: Las cuevas estarán interconectadas mínimamente, sin seguir un criterio especifico.

	○ **Medio**: Tiene que haber un grupo de cuevas principal, interconectadas que recorra todo el mapa. Puede haber otros grupos, o cuevas individuales sin conectar con el grupo principal. ○ **Alto**: Todas las cuevas interconectadas. No se permiten cuevas o grupos aislados.

ID del requisito	*RF08*
Nombre	**Generación de plataformas**
Descripción	Otra funcionalidad que interesa tener es la generación de plataformas en el aire. Esta funcionalidad se debe poder configurar mediante los siguientes parámetros: • **Grosor de la plataforma**. Número entero expresado en *Tiles*. Como su nombre indica, representa el grosor fijo de la plataforma. • **Altura mínima**. Número entero expresado en *Tiles*. Altura mínima de una plataforma con respecto al suelo o con respecto a otra plataforma cercana. • **Altura máxima**. Número entero expresado en *Tiles*. Altura máxima de una plataforma con respecto al suelo o con respecto a otra plataforma cercana. • **Anchura mínima**. Número entero expresado en *Tiles*. Como su nombre indica, es la anchura mínima de una plataforma. • **Anchura máxima**. Número entero expresado en *Tiles*. Como su nombre indica, es la anchura máxima de una plataforma. • **Densidad de plataformas**. Número entero. Similar al parámetro de la densidad de las cuevas. • **Probabilidad de escaleras**. Número entero, en intervalo de uno a cien, que representa la probabilidad de que una plataforma tenga una escalera hasta la plataforma inferior. Las escaleras siempre irán de una plataforma hasta el elemento inmediatamente inferior, ya sea otra plataforma, o el suelo. • **Modo de colisión**. Permite configurar el modo en el que otros objetos, por ejemplo, el personaje del jugador, pueden colisionar con las plataformas Tiene dos valores posibles: ○ **Unidireccional**. Los objetos solo colisionan con las plataformas cuando ejecutan un movimiento desde arriba hacia abajo. Esto permite por ejemplo a un jugador saltar a una plataforma que esta justo encima de él, pasando por la plataforma, pero sin caerse una vez está encima de la plataforma. ○ **Completo**. El contacto con una plataforma causa colisión independientemente de la dirección de desplazamiento.

ID del requisito	*RF09*
Nombre	**Colocación de objetos externos**

Descripción	El sistema debe permitir la colocación de objetos de tipo *GameObject*, configurables por el programador, en el mapa generado. Se permite la colocación de un número ilimitado de objetos.
	
	El programador podrá controlar la colocación de los objetos mediante los siguientes parámetros de configuración:
	• **Modo de colocación.** Permite elegir el modo en el que se va a hacer la distribución de los elementos. Tiene 2 valores posibles:
	  ○ **Aleatorio**. La colocación se va a hacer en un modo pseudo-aleatorio, colocando los objetos en el mapa sin considerar las posiciones de los demás elementos del mismo tipo.
	  ○ **Intervalo**. En este modo se coloca un objeto único a un intervalo regulado, con o sin variación del intervalo.
	• **Posición**. Permite elegir en que parte del mapa se colocaran los objetos. Posibles valores:
	  ○ **Aire**. Los objetos se colocarán en el aire, a una altura variable.
	  ○ **Suelo**. Los objetos se colocarán a nivel del suelo.
	  ○ **Tierra**. Los objetos se colocarán dentro de la tierra, pero fuera de cualquier cueva.
	  ○ **Cueva**. Los objetos se colocarán dentro de las cuevas.
	• **Densidad**. Este parámetro será un numero entero. Solo es efectivo si el modo de colocación es Densidad. No se especifica rango, ni unidad, dejando la elección al gusto del programador, sin embargo, el efecto de cambiar este parámetro tiene que ser uno lineal, similar a los parámetros definidos en el requisito RF07. Variar este parámetro tiene que cambiar la densidad aparente de los elementos colocados.
	• **Intervalo de colocación**. Número entero expresado en *Tiles*. Solo es efectivo si el modo de colocación es Intervalo. Con este parámetro se configura el intervalo permitido entre la ocurrencia de cada instancia de este objeto.
	• **Variación del intervalo**. Número entero expresado en *Tiles*. Solo es efectivo si el modo de colocación es Intervalo. Con este parámetro se le añade un elemento de aleatoriedad a la colocación de los elementos.
	  ○ Si este parámetro es un numero positivo, mayor que uno, el parámetro actúa como el máximo de una variación aleatoria al intervalo de colocación.
	  ○ Si es igual a cero, los elementos se colocan a un intervalo fijo.
	  ○ El valor de este parámetro no puede ser mayor que el del intervalo de colocación.
	
	Al permitir usar cualquier tipo de *GameObject* el propósito de este parámetro es funcionar tanto para elementos interactivos de juego como para elementos decorativos.

ID del requisito	*RF10*
Nombre	**Fondo con efecto *parallax***

Descripción	Este requisito funcional, a diferencia de los demás, se centra exclusivamente en una funcionalidad estética. Consiste en una serie de parámetros que permite crear un fondo con efecto *parallax* de varias capas. Debe permitir añadir un número ilimitado de capas de fondo. Cada capa debe tener los siguientes parámetros de configuración: • **Imagen.** Ruta o referencia a una imagen. Esta imagen se reutilizará para crear un efecto continuo. Es responsabilidad del usuario que esta imagen se pueda juntar de forma continua para un efecto correcto de continuidad. • **Orden de capa.** Numero entero. Determina el orden en el que se *renderizan* las diferentes capas del fondo. Una capa que tenga un número mayor que otra, aparecerá detrás. Las capas que tengan número de orden negativo aparecerán por delante de mapa. • **Velocidad de efecto.** Numero de tipo *float*. Representa la velocidad de desplazamiento de la capa con respecto a la cámara principal. Valores inferiores a uno representan un movimiento mas lento que la cámara, el valor uno representa un movimiento igual a la velocidad de la cámara y un valor superior a uno representa un movimiento mas rápido que la propia cámara.

ID del requisito	*RF11*
Nombre	**Configuración estética**
Descripción	El sistema debe permitir la configuración del aspecto de los *Tiles*. Las diferentes imágenes de cada elemento se tienen que especificar mediante una ruta o mediante una referencia directa a una imagen. Se debe permitir la configuración del aspecto de los siguientes elementos: • **Suelo**. Se entiende como suela la línea que separa el aire de la tierra. Se tienen que poder configurar imágenes donde la línea del suelo tiene diferentes ángulos con respecto a la horizontal: 0^{o}, 45^{o}, 90, 135^{o}. • **Tierra**. Se entiende como tierra el espacio que se encuentra debajo del suelo y no es una parte de cueva. • **Aire**. Se entiende como aire el espacio que se encuentra por encima del suelo y que no es plataforma. • **Interior de cueva**. • **Pared de cueva**. Se entiende como pared de cueva el espacio que separa la cueva de la tierra. De forma opcional la configuración del aspecto se podría hacer también mediante una única imagen de tipo paleta, donde cada una de las imágenes estarán en una posición específica, dentro de una imagen de tipo contenedor.

ID del requisito	*RF12*
Nombre	**Colisiones**

Descripción	Para que el mapa fuera usable es necesario que los jugadores puedan interactuar con ella. Esto se consigue a través de colisiones. Los siguientes elementos del mapa deben tener asociado un componente de tipo *Collider*: • **Suelo**. Como es fundamental, el suelo debe tener habilitadas las colisiones para que el jugador no se caiga del mapa. • **Plataforma**. Las plataformas deben permitir las colisiones con los jugadores. Las colisiones de este elemento se tratan más en detalle en el RF08. • **Pared de cueva**. La pared de cueva debe tener colisiones habilitadas para que los jugadores no puedan pasar a la tierra. En resumen, el propósito de las colisiones es hacer lo que se considera como tierra, impenetrable, salvo una excepción que se describe en el requisito RF08.

ID del requisito	*RF13*
Nombre	**Semilla del mapa**
Descripción	Cualquier aspecto de los requisitos anteriores que requieran algún tipo de generación aleatoria se hará en base a una semilla única para todos los elementos. Todas las ejecuciones que se realicen con la misma semilla generaran el mismo mapa, en todos los casos. Esta semilla tiene que ser configurable.

3.4 Requisitos no funcionales

En esta sección se describen los requisitos no funcionales del sistema.

3.4.1 Requisitos de rendimiento

Se considera aceptable el incremento del tiempo inicial de carga al utilizar configuraciones más complejas. Dicho esto, este incremento en ningún momento debe ser de una magnitud que pueda causar limitaciones en el uso de la herramienta.

Se considera uso del producto tanto el desempeñado por el usuario desarrollador a la hora de diseñar un videojuego utilizado este Asset, como el uso del usuario final del videojuego que internamente use este Asset.

3.4.2 Seguridad

Los ficheros XML que componen la configuración tienen que mantenerse ocultos una vez construida la aplicación final que hace uso de esta herramienta, protegiendo a la aplicación frente a ediciones no intencionadas por el desarrollador.

3.4.3 Fiabilidad

El producto debe tener una fiabilidad garantizada, siendo inaceptable cualquier tipo de fallo interno que afecte al funcionamiento normal de la herramienta siempre y cuando el uso es debido y conforme a las especificaciones del producto.

3.4.4 Portabilidad

Al tratarse de un Asset de Unity la portabilidad viene dada por el propio Unity. Unity está disponibles en varios sistemas operativos y permite compilar aplicaciones para varias plataformas. El producto tiene que estar compatible con cualquier combinación de sistema operativo y plataforma destino, igual que el propio Unity.

4. Proceso de desarrollo software

En esta sección se van a describir las diferentes metodologías utilizadas en el desarrollo. Estas metodologías determinan tanto los requisitos en los que se va a trabajar, como la forma en la que se sigue la evolución de las tareas.

4.1 Mínimo producto viable

Tal como se puede ver en la sección 3, donde se especifican los requisitos del sistema, implementar todos los requisitos puede suponer un trabajo muy extenso. Esto no solo quedaría fuera del alcance de un TFG, sino que existe la posibilidad que los propios requisitos no cumplen con las necesidades de los usuarios y que necesiten modificaciones. Aunque se ha realizado un estudio previo de las necesidades de los desarrolladores de videojuegos de plataformas y de las herramientas existentes, aun no se dispone de realimentación directa por parte de los propios interesados.

Por estas razones es muy importante de disponer de un mínimo producto viable, o *MVP (Minimum Viable Product),* que nos permita recibir realimentación de parte de los *early adopters,* antes de invertir tiempo en implementar un producto completo que necesite modificaciones nada mas finalizarlo.

El concepto de *MVP* tiene muchas definiciones posibles, algunas más enfocadas en los aspectos económicos de crear un producto nuevo y otras más enfocadas en recibir *feedback* por parte de los usuarios. Por esta razón, en 2016, Valentina Lenarduzzi y Davide Taib (8) realizaron un estudio sobre las diferentes definiciones de este concepto. Para este trabajo se han considerado las características comunes de las diferentes definiciones que se han encontrado como parte de ese estudio.

Un mínimo producto viable, *MVP*, es una versión inicial del producto final que:

- Tiene la funcionalidad suficiente para poder ser desplegado
- Tiene las características mínimas para ser de utilidad a los usuarios
- Permite probar el producto en el mercado
- Sirve para recibir feedback por parte de los usuarios
- Requiere el esfuerzo mínimo para recibir la recibir la mayor cantidad de información sobre las necesidades de los usuarios, es decir la mejor relación esfuerzo - *feedback*

Considerando esta definición, se ha decidido implementar los siguientes requisitos como parte del *MVP:*

- RF01:
  - Se va a implementar de forma completa.
- RF02:
  - Se va a implementar de forma parcial, permitiendo el modificado de los parámetros de configuración mediante la ventana *Inspector*, pero sin implementar ficheros externos de tipo JSON.
- RF03:
  - Se implementa solo la configuración de altura y anchura.
  - Se implementa solo la generación de mapas horizontales, que tengan la anchura mayor que la altura.
- RF04:
  - Se implementa de forma completa, con una excepción: el parámetro de numero de segmentos no tiene en cuenta el numero de segmentos visibles en pantalla, sino que actúa como numero total de segmentos en todo momento. De esta manera se simplifica la implementación, pero tiene el coste de que el usuario debe determinar el numero mínimo de segmentos que tiene que cargar para no ver los limites del mapa.
- RF06:
  - Se implementa con la excepción del parámetro de modo solo tierra.
- RF08:
  - Se omiten los siguientes parámetros:
    - Probabilidad de escaleras – no se implementan escaleras
    - Modo de colisión: Solo se implementa el modo Completo, por lo que se omite el parámetro.
- RF09:
  - Solo se implementa el modo aleatorio de colocación, por lo que se omite este parámetro de configuración.
  - Parámetro de densidad. Se implementa.

    - Parámetros de intervalo de colocación y variación del intervalo no se implementan.
- RF10:
    - Para el MVP, el parámetro de Imagen se implementa de tipo *GameObject*.
    - Al implementar la Imagen como *GameObject* se emite el parámetro de orden de capa, dejando como responsabilidad del usuario a configurar el orden de capa adecuado en el *GameObject* original.
- RF11:
    - Se implementa el aspecto de los *Tiles* de tipo Plataforma y Suelo.
- RF12:
    - Se implementa con la excepción de las cuevas.
- RF13:
    - Se implementa por completo.

Los requisitos funcionales que no se han mencionado en esta lista no se implementan como parte del mínimo producto viable.

En cuanto a los requisitos no funcionales, se va a dar prioridad a los requisitos de rendimiento y portabilidad.

4.2 Metodología Kanban

dos Santos et al. (9) definen Kanban como un método para visualizar el flujo de trabajo de un sistema de producción. Según los resultados encontrados por Ahmad et al. (10), el tablero Kanban expresa de forma explicita las tareas mas importantes que se necesitan realizar en cada momento, reduciendo el riesgo de dejarlas incompletas.

Para este proyecto se ha decidido utilizar este método para hacer seguimiento de las tareas. Esto es necesario debido a la necesidad de compaginar la realización del proyecto con un trabajo a jornada completa. El hecho de cambiar cada día de contexto de trabajo puede causar la perdida del enfoque en las tareas importantes.

Mas adelante, en usa sección posterior se hablará de la herramienta utilizada en implementar este método de trabajo.

4.3 Control de versiones

Git[9] (11) es un software gratuito y de código abierto que permite tener un control sobre los cambios que se realizan en el código fuente de un proyecto. Git es compatible con las principales plataformas, tales como Linux, Windows y macOS y funciona como un sistema distribuido, en el que cada desarrollador trabaja contra su copia local del

[9] https://git-scm.com

repositorio. Esto lo hace especialmente atractivo para equipos grandes. Fue desarrollado por Linus Torvald en 2005

En el momento de realizar este trabajo, todas las principales aplicaciones o sistemas online de almacenamiento para código fuente de tipo abierto utilizan el sistema Git para el control de versiones, tales como GitHub, Sourceforge, GitLab, Bitbucket etc. Teniendo en cuanta que el objetivo de este trabajo es llegar a ser fácilmente utilizado por el mayor numero de usuarios, se decide usar Git como sistema de control de versiones para ser compatible con el mayor numero de estas aplicaciones online.

En cuanto al flujo de trabajo de las ramas de Git, no se ha encontrado estudios claros que muestren cuales son los mas populares entre los equipos de desarrolladores. Sin embargo, parece que el que mas se menciona en las diferentes publicaciones online es GitFlow.

GitFlow (12) es un flujo de trabajo que consiste en dos ramas principales que nunca se cierran, *main* y *develop*. Develop es la rama en la cual se van incorporando los diferentes cambios que se van realizando en el repositorio. Los cambios no se realizan directamente sobre esta rama, sino se crean ramas especificas para cada cambio, llamadas ramas *feature,* que luego mediante un *merge* incorporan sus cambios en *develop*. Cuando el proyecto esta preparado para lanzar una nueva versión, se crea una rama de tipo *release* a partir de *develop* para esa versión concreta que se pretende lanzar. En esta rama se realizan las ultimas pruebas y solución de bugs. Cuando la versión esta preparada, la rama *release* se incorpora en la rama *main* y *develop* y se elimina. En la Ilustración 9 se puede ver un ejemplo de este flujo que se entiende mejor con un apoyo grafico.

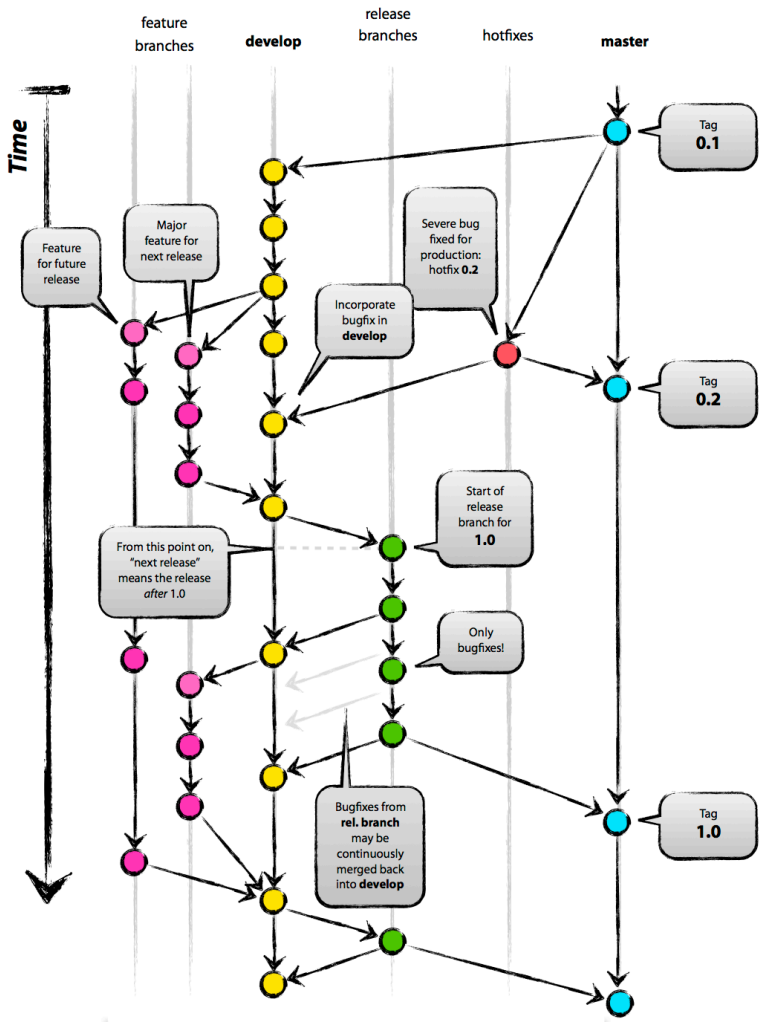

Ilustración 9:Ejemplo Gitflow

5. Análisis funcional

Después de ver en la sección anterior cuáles son los requisitos del sistema, esta sección se va a centrar en especificar el diseño de la solución que permita cumplir con los requisitos.

El análisis funcional es un paso previo al propio desarrollo de software y en su presentación no suele incluir código ya implementado. Sin embargo, debido a la naturaleza de este documento, que incluye secciones que se desarrollan en diferentes etapas del ciclo de vida del software, se ha completado esta sección con fragmentos de código que se han desarrollado después de finalizar el análisis funcional.

En cuanto la arquitectura, se ha seguido un diseño modular, dividiendo el código en varios ensamblados con el propósito de crear una especie de barrera entre los diferentes módulos. Se ha intentado obtener una estructura que no fuera demasiado compleja para

que fuera fácil de entender, pero al mismo tiempo dejar bien definidos los diferentes módulos y sus funciones, para que a largo plazo el riesgo del llamado *código espagueti* fuera mínimo.

5.1 Espacios de nombres

Los espacios de nombres son una manera de definir el alcance de una clase. Utilizar espacios de nombre bien definidos, mas que una herramienta para el programador que los crea, son una forma de presentar el código de una manera lógica. La organización de los mismos suele reflejar la estructura de las carpetas del proyecto, aunque no es algo necesario. Este proyecto tiene definidos los siguientes espacios de nombres:

- AMG2D:
  - Es el espacio de nombres principal del proyecto.
  - Solo contiene una clase, *MapGenerator*, que extiende de la clase *MonoBehaviour* de Unity y es el script principal del proyecto.
- AMG2D.Model:
  - Contiene solo las interfaces de los diferentes servicios de la aplicación.
- AMG2D.Model.Persistence:
  - Este espacio de nombres contiene el modelo de datos del mapa.
  - En este caso persistencia no hace referencia a persistencia en disco, sino a persistencia en memoria del mapa generado, que permite cargar y descargar varias veces el mismo mapa, o partes de ella.
- AMD2D.Implementation:
  - Agrupa todas las implementaciones de todas las diferentes interfaces de servicios del espacio de nombre AMG2D.Model.
  - En proyectos clásicos mas grandes de C# es común encontrar varios espacios de nombres de implementación, para cada una de las implementaciones de una interfaz. Sin embargo, en este caso se ha optado por tener un único espacio de nombres, ya que tener varios no aportaría ninguna ventaja y solo empeoraría la complejidad de la solución.
- AMG2D.Bootstrap:
  - En cuanto al Bootstrap, es un espacio de nombres que tiene una función importante en la implementación del patrón de diseño *Bridge*. Tiene el rol de “atar los cordones” (del inglés *bootstrap*) de nuestra aplicación, indicando que implementación de cada uno de los servicios se va a ejecutar en cada arranque.
- AMG2D.Configuration:
  - Como su nombre indica, este espacio de nombres agrupa todas las clases que contienen los parámetros de configuración del proyecto.

5.2 Arquitectura y ensamblados

Dividir el código en varios ensamblados tiene varias ventajas. En primer lugar, crea una barrera que es mas difícil de cruzar a la hora de referenciar entre si las diferentes partes del código. Aparte de añadir la referencia al espacio de nombres que se quiere utilizar mediante la palabra clave *using,* que en muchas ocasiones el *IDE* la añade sin darse uno cuenta, es necesario añadir también una referencia al ensamblado que contiene esa clase. Esto ayuda a evitar crear de forma accidental referencias no deseadas entre diferentes componentes del proyecto.

En segundo lugar, tener varios ensamblados también ayuda a ahorrar tiempo y frustraciones a la hora de desarrollar nuestro proyecto. A medida que se va modificando y probando nuestro proyecto, el compilador solo vuelve a compilar los ensamblados que hayan sido modificados desde la ultima compilación. Este ahorro de tiempo puede ser muy notable a medida que el proyecto va creciendo, sobre todo si el trabajo se realiza en una maquina con pocos recursos.

Dicho esto, el proyecto queda dividido en cinco ensamblados, encapsulando los espacios de nombre del mismo nombre. Las dependencias entre los diferentes ensamblados quedan reflejadas en la *Ilustración* 10.

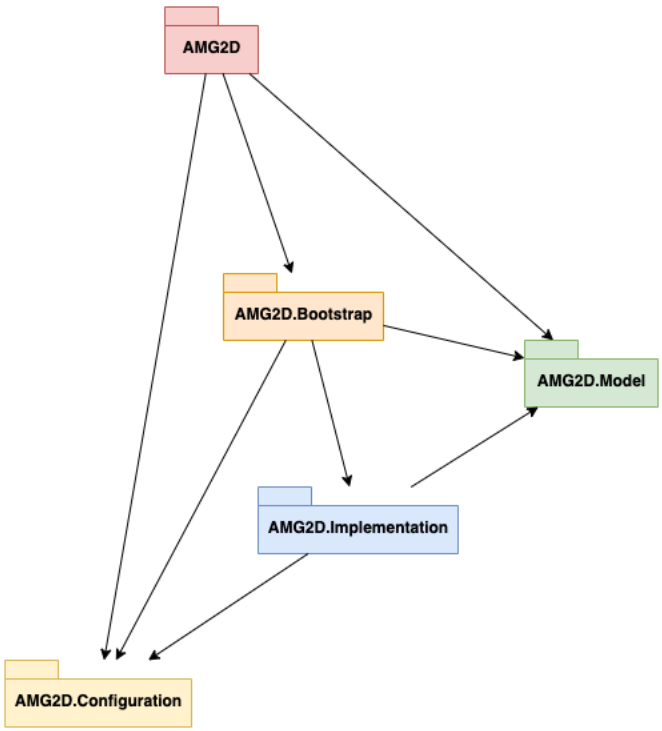

Ilustración 10: Ensamblados del proyecto

Dentro de estos ensamblados, en la Ilustración 11 se puede ver como quedaría la estructura completa del proyecto, con las respectivas clases de cada ensamblado. En el diagrama resultante se han omitido algunas propiedades o métodos privados que no influyen en la estructura para hacer el diagrama mas fácil de leer.

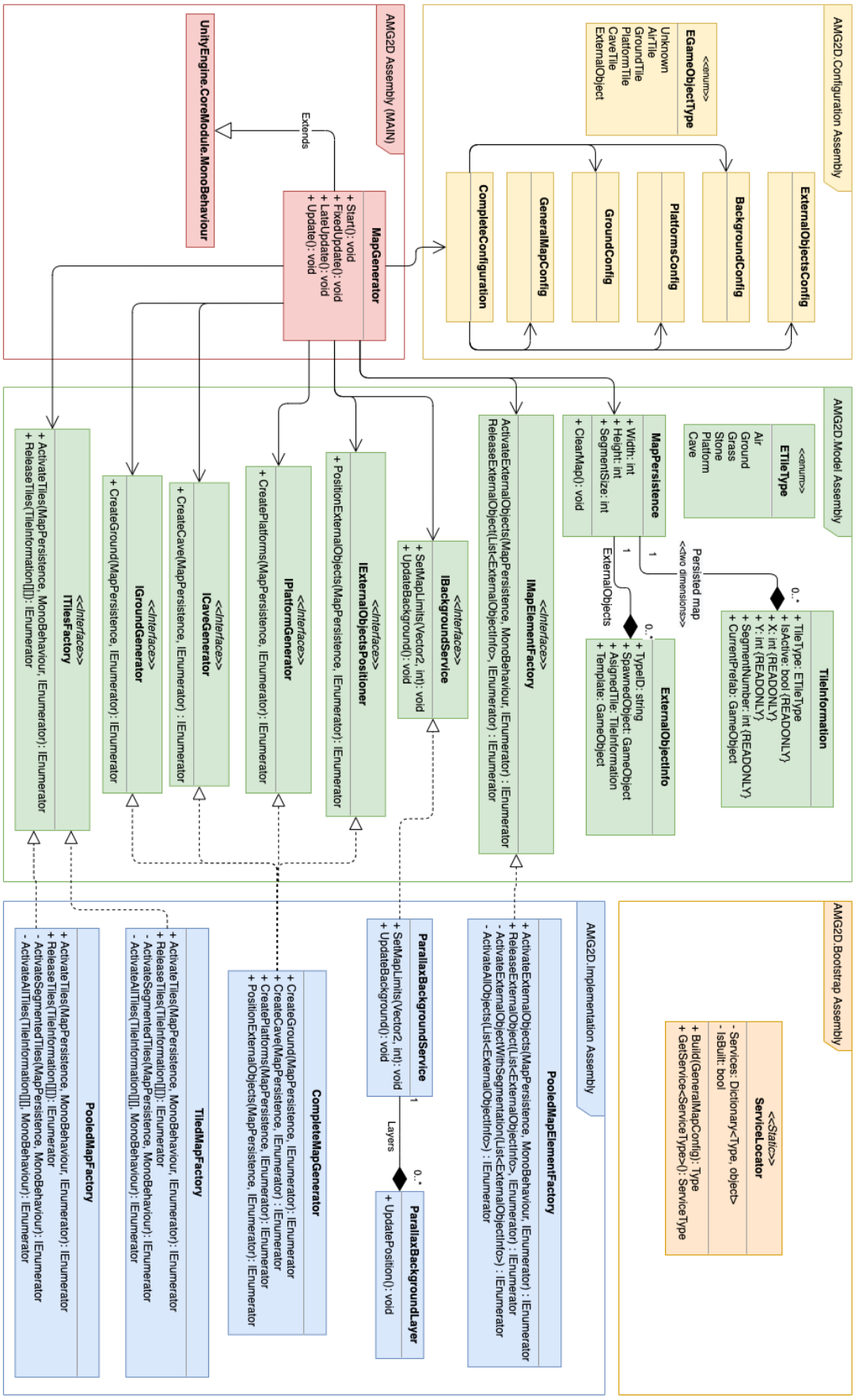

Ilustración 11: Diagrama de clases del proyecto

De todas las clases presentes en la estructura, las mas importantes son las siguientes:

- AMG2D.MapGenerator:
  - Esta es la clase principal del proyecto, y es con la que el usuario interactúa. Al extender de la clase MonoBehaviour, permite utilizarse como script para objetos dentro de las escenas de juego Unity. El usuario interactúa con esta mediante las propiedades que quedaran expuestas en la ventana *Inspector*, tal como se puede ver en la Ilustración 12.

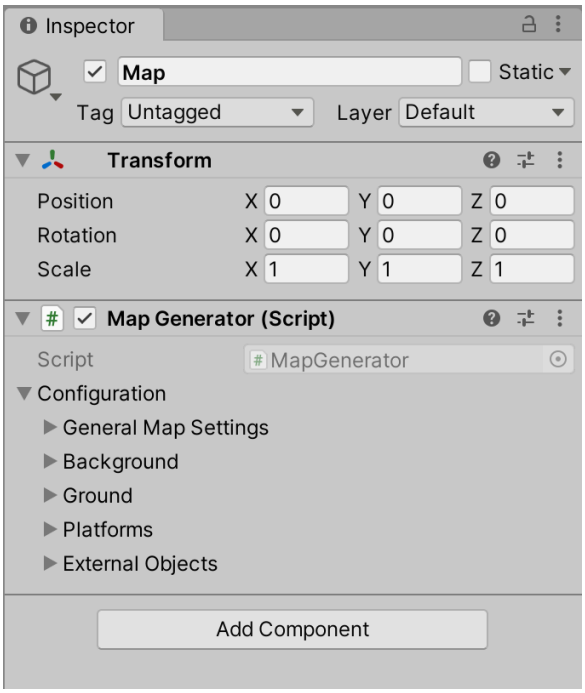

Ilustración 12: Ventana Inspector de MapGenerator

- AMG2D.Model.MapPersistence
  - Esta clase, junto con las demás clases que componen sus propiedades, forman la estructura de base que representa el mapa. Independientemente de si el mapa esta cargado o no en un momento dado, esta estructura se mantiene en memoria y se utiliza como la base para cargar el mapa o partes del mapa cuando es necesario.
- AMG2D.Model.*<interface>
  - Las interfaces presentes en el ensamblado AMG2D.Model componen la totalidad de los contratos de servicio que la clase MapGenerator consume para cumplir con los requisitos del sistema.
  - Las interfaces presentes aquí se dividen en dos tipos:
    - Generadores de estructura: son las interfaces que alteran un objeto de tipo MapPersistence para dotarlo de diferentes elementos, como terreno, plataformas u otro tipo de objetos (IExternalObjectPositioner, ICaveGenerator, IGroundGenerator, IPlatformGenerator)
    - Generadores de elementos de juego: son las interfaces que utilizan un objeto de tipo MapPersistence solo para lectura, generando a partir de los elementos de juego necesarios (IBackgroundService, IMapElementFactory, ITilesFactory)

- AMG2D.Implementation.*
  - Todas las clases presentes en este espacio de nombres son implementaciones de las interfaces del espacio de nombres AMG2D.Model. Estas son las que realmente proporcionan los servicios a la *MapGenerator*. Algunas interfaces tienen mas de una implementación.
- AMG2D.Bootstrap.ServiceLocator
  - Esta clase es la que *MapGenerator* usa para encontrar las referencias a las implementaciones de las diferentes interfaces de servicio que utiliza. Aquí se especifica cual es la implementación que se va a utilizar de cada interfaz.

5.3 Patrones de diseño

Bridge

El primero primer patrón de diseño del que voy a hablar es *Bridge*. Se trata de un patrón de diseño estructural que consiste en desacoplar la implementación de una funcionalidad de su interfaz. Esta interfaz actúa como una especie de contrato en el código que necesita la funcionalidad ofrecida, sin referenciar directamente a la clase que lo implementa. De esta manera se permite cambiarla fácilmente a posteriori por una implementación diferente. También es muy útil a la hora de probar nuestro código, permitiendo crear implementaciones falsas o *Mock* de las diferentes interfaces con el propósito de probar el comportamiento del código.

Este patrón se esta utilizando para desacoplar toda la funcionalidad ofrecida por esta solución. En la Ilustración 11 se puede ver todas las interfaces del ensamblado AMG2D.Model y sus respectivas implementaciones del ensamblado AMG2D.Implementation. En la *Ilustración* 13 se puede ver como la clase *MapGenerator* referencia a las interfaces y no a las implementaciones para hacer uso de los diferentes servicios.

```
namespace AMG2D
{
    public class MapGenerator : MonoBehaviour
    {
        private IMapElementFactory _elementFactory;
        private IGroundGenerator _groundGenerator;
        private IPlatformGenerator _platformGenerator;
        private ITilesFactory _tilesFactory;
        private IExternalObjectsPositioner _externalObjectsPositioner;
        private ICaveGenerator _caveGenerator;
        private IBackgroundService _background;
```

Ilustración 13: Referencias a servicios

Inversion of control

Inversion of control, conocido como IoC, no es un patrón de diseño sino un principio de programación. Consiste en quitar el control de las clases sobre la creación de los objetos

de los que dependen. Cualquier instancia que se necesite llegara de forma externa. Este se puede implementar mediante varios patrones de diseño, tales como *Service Locator* o *Dependency Injection*.

La inyección de dependencias consiste en pasar por parámetro en el método constructor de una clase cualquier dependencia que ella necesite. Esta manera de realizar la inversión de control es muy cómoda debido a que el código no necesita referenciar ningún tipo de clase o *framework* especifico para resolver las referencias a los servicios. Eso hace que sea muy fácil cambiar de un framework a otro, o incluso inyectar de forma manual las referencias, conocido como *poor man's dependency injection*.

El patrón *Service Locator* consiste en un objeto que se puede referenciar de forma global, al que las clases llaman para obtener los objetos de los que dependen. La principal diferencia entre *Service Locator* y el patrón Factory al que se parece es que los objetos que Service Locator ofrece no son necesariamente instancias nuevas, sino que suelen ser clases de tipo *Singleton* que se comparten en varios sitios de la aplicación. El *Factory* en cambio se dedica a crear siempre objetos nuevos, que suelen ser contenedores de datos sin ofrecer mucha funcionalidad.

En este proyecto se ha optado por aplicar ambos patrones. La inyección de dependencia se utiliza para inicializar las clases del espacio de nombres AMG2D.Implementation que se pueden ver en la Ilustración 14. Estas reciben como parámetro en sus constructores los objetos de configuración necesarios.

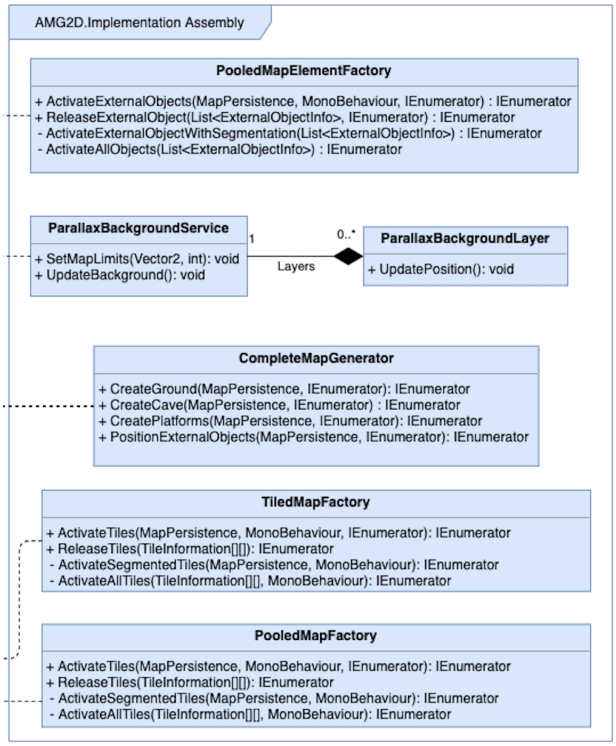

Ilustración 14: Clases de implementación de servicios

Aunque en muchos casos la inyección de dependencias es superior al patrón *Service Locator*, no ha sido posible utilizarla en todos los casos. En el caso de la clase *MapGenerator,* al extender de la clase *MonoBahaviour,* Unity no permite al usuario cambiar su constructor y tampoco instanciarla directamente. Por lo tanto, es imposible

aplicar la inyección de dependencias en este caso. Por eso se ha optado por el *Service Locator*. Al ser una clase publica estática, *ServiceLocator* se puede referenciar desde cualquier parte del código. Mediante esta clase, MapGenerator resuelve sus servicios, tal como se puede ver en la Ilustración 15.

```
private void ResolveServices()
{
    try
    {
        _elementFactory = ServiceLocator.GetService<IMapElementFactory>()
            ?? throw new ArgumentNullException($"Service {nameof(IMapElementFactory)} cannot be null");
        _tilesFactory = ServiceLocator.GetService<ITilesFactory>()
            ?? throw new ArgumentNullException($"Service {nameof(ITilesFactory)} cannot be null");
        _groundGenerator = ServiceLocator.GetService<IGroundGenerator>()
            ?? throw new ArgumentNullException($"Service {nameof(IGroundGenerator)} cannot be null");
        _caveGenerator = ServiceLocator.GetService<ICaveGenerator>()
            ?? throw new ArgumentNullException($"Service {nameof(ICaveGenerator)} cannot be null");
        _externalObjectsPositioner = ServiceLocator.GetService<IExternalObjectsPositioner>()
            ?? throw new ArgumentNullException($"Service {nameof(IExternalObjectsPositioner)} cannot be null");
        _platformGenerator = ServiceLocator.GetService<IPlatformGenerator>()
            ?? throw new ArgumentNullException($"Service {nameof(IPlatformGenerator)} cannot be null");
        _background = ServiceLocator.GetService<IBackgroundService>()
            ?? throw new ArgumentNullException($"Service {nameof(IBackgroundService)} cannot be null");
    }
    catch (Exception ex)
    {
        Debug.LogError($"Failed to resolve services due to: {ex}");
        Application.Quit();
    }
}
```

Ilustración 15: Localización de servicios en la clase MapGenerator

El patrón de diseño *Bridge* mas la inversión de control se complementan muy bien y son la clave para tener un código fácil de modificar y probar permitiendo cambiar fácilmente entre las diferentes implementaciones

Strategy

El tercer patrón de diseño del que se va a hablar es el *Strategy*. Este patrón, al nivel mas básico consiste en hacer que una llamada a un método tenga resultados diferentes en función de algunos factores externos, o estrategia. Tal como se podrá ver a continuación, en este proyecto, la utilidad principal de este patrón es poder variar los diferentes parámetros de configuración en la ventana *Inspector* de Unity y que estos cambios tengan un efecto instantáneo sobre el mapa generado.

A la hora de implementar este patrón, hay varias opciones. Si, por ejemplo, se crean varias implementaciones de una única interfaz como parte del patrón *Bridge* y se hace que estas se intercambien en tiempo de ejecución, en función de una *estrategia*, se estaría hablando del patrón *Strategy*. La otra opción, que es la que se ha implementado en este proyecto, es cambiar el comportamiento de la llamada a un método, haciéndole llegar a la clase que implementa el método un objeto que le indica la estrategia a utilizar.

El objeto que actúa como estrategia es la configuración. A la hora de instanciar las implementaciones de los diferentes servicios se les inyectan un objeto de tipo *CompleteMapConfiguration*. Este objeto es el mismo que la clase *MapGenerator* publica en la ventana *Inspector*. De esta forma, al cambiar cualquier parámetro de configuración, este cambio se propaga a todos los servicios.

Se opto por inyectar la configuración en el constructor en lugar de pasarla en cada llamada de método. Esto es debido a que en este caso la instancia del objeto

configuración es siempre la misma y de esta manera se consigue simplificar las cabeceras de los métodos, omitiendo el objeto de configuración.

```
/// <summary>
/// Prepares internal state of class for resolving services.
/// </summary>
public static void Build(CompleteConfiguration completeConfig)
{
    if(completeConfig == null) throw new ArgumentNullException($"Argument {nameof(completeConfig)} cannot be null");
    _services = new Dictionary<Type, object>
    {
        { typeof(IMapElementFactory), new PooledMapElementFactory(completeConfig) },
        { typeof(ITilesFactory), new TiledMapFactory(completeConfig.GeneralMapSettings) },
        { typeof(ICaveGenerator), new CompleteMapGenerator(completeConfig) },
        { typeof(IPlatformGenerator), new CompleteMapGenerator(completeConfig) },
        { typeof(IGroundGenerator), new CompleteMapGenerator(completeConfig) },
        { typeof(IExternalObjectsPositioner), new CompleteMapGenerator(completeConfig) },
        { typeof(IBackgroundService), new ParallaxBackgroundService(completeConfig) }
    };
    _isBuilt = true;
}
```

Ilustración 16: Creación de los servicios

Prototype

El siguiente patrón de diseño es la clave que permite la implementación del requisito funcional RF09 de una manera sencilla. Se trata del *Prototype*. Este patrón de diseño consiste crear instancias de objetos nuevos mediante la clonación de objetos existentes, en lugar de instanciar directamente objetos nuevos.

La implementación mas común de este método consiste en dotar a la clase que se necesita clonar de un método llamado *Clone()*. Mediante este método, aprovechando también el acceso a las propiedades privadas, un objeto se clona a si mismo.

En nuestro proyecto, la clonación es necesaria en la implementación del requisito RF09, para poder crear objetos nuevos a partir de una plantilla que el usuario establece en la configuración. Tal como se puede ver en la *Ilustración* 17, la clase *ExternalObjectInfo* tiene dos propiedades de tipo *GameObject*. Una de ellas, *Template*, es una referencia común a todas las instancias de este tipo de objeto. La otra, *SpawnedObject*, es un objeto único para cada *ExternalObjectInfo* y se obtiene clonando el objeto de la propiedad *Template*.

```
namespace AMG2D.Model.Persistence
{
    /// <summary>
    /// Object holding the information of a specific external object instance that w
    /// </summary>
    public class ExternalObjectInfo
    {
        /// <summary>
        /// Type of the object.
        /// </summary>
        public string TypeID;

        /// <summary>
        /// <see cref="GameObject"/> instance that represents this external object.
        /// </summary>
        public GameObject SpawnedObject;

        /// <summary>
        /// Asigned tile where the object will be positioned initially.
        /// </summary>
        public TileInformation AsignedTile;

        /// <summary>
        /// <see cref="GameObject"/> template from which the <see cref="SpawnedObjec
        /// </summary>
        public GameObject Template;
    }
}
```

Ilustración 17: Clase ExternalObjectInfo

Sin embargo, gracias a Unity en este caso no es necesaria la implementación de un método de clonación. Unity ya ofrece esta funcionalidad mediante el método *Instantiate(…)* de la clase *MonoBehaviour*. Tal como se puede ver en la *Ilustración* 18, la clase *PooledMapElementFactory* clona un objeto nuevo cuando en su cola de objetos inactivos no encuentra uno del tipo necesario.

```
if (obj.SpawnedObject != null) continue;
if(_pools[obj.TypeID].Count > 0)
{
    obj.SpawnedObject = _pools[obj.TypeID].Dequeue();
    obj.SpawnedObject.transform.position = new Vector2(obj.AsignedTile.X + CENTER_OFFSET, obj.AsignedTile.Y + CENTER_OFFSET);
    obj.SpawnedObject.SetActive(true);
}
else
{
    obj.SpawnedObject = MonoBehaviour.Instantiate(obj.Template,
        new Vector2(obj.AsignedTile.X + CENTER_OFFSET, obj.AsignedTile.Y + CENTER_OFFSET),
        Quaternion.identity);
}
```

Ilustración 18: Clonación de objetos externos

Iterator

El patrón *Iterator* no es uno que se implementa de forma explicita en este proyecto, pero sin embargo es la base de casi todas las llamadas a métodos que se realizan. Esto es debido a las corrutinas de Unity.

Unity, por diseño, no permite modificar objetos de juego desde otro hilo que no fuera el principal. Esto quiere decir que cualquier elemento que se necesite actualizar del mundo de juego, necesita hacerse como parte del código que se ejecuta en uno de los eventos que se lanzan desde la clase *MonoBehaviour,* tales como *Update(), FixedUpdate(), LateUpdate()* etc. Estos eventos se lanzan de forma cíclica antes de generar un

fotograma. Cualquier retraso en la ejecución de estos métodos suponen un retraso en la generación del fotograma y en consecuencia en un deterioro en la experiencia de juego.

Para las tareas frecuentes y rápidas esto no es un problema, sin embargo, hay situaciones donde se necesita realizar una tarea de forma puntual, que actúa sobre objetos de juego y cuya duración es mayor que la de un fotograma. Para estos casos, Unity pone a nuestra disposición las corrutinas. Las corrutinas son una manera de dividir la ejecución de un método en varios trozos pequeños, sobre cuales el motor de Unity itera en cada fotograma. Esta herramienta no esta disponible en .Net y es especifica para la versión de Unity de C#. De esta manera se consigue repartir la ejecución de una tarea mas larga sobre varios fotogramas para no afectar el tiempo de ejecución de un fotograma en concreto.

En la base de esta función se encuentra el patrón *Iterator*. Este patrón de diseño de comportamiento consiste en recorrer una colección de datos, sin conocer la estructura interna de la misma. En este caso, Unity utiliza internamente este patrón para recorrer las diferentes secuencias de nuestras corrutinas. Una corrutina siempre devuelve el tipo *IEnumerator*, tal como se puede ver en la Ilustración 19 y marca las diferentes partes que en la que se divide la ejecución de un método mediante la palabra clave *yield*.

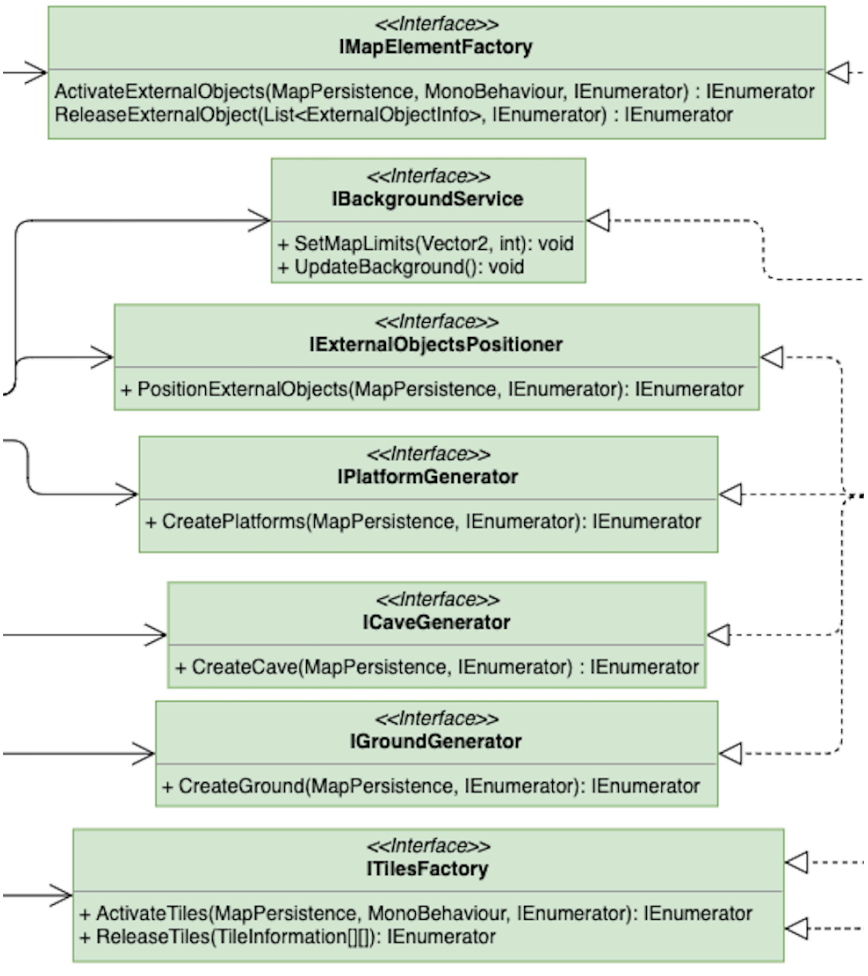

Ilustración 19: interfaces del proyecto

Command

Una dificultad que añade el uso de corrutinas con respecto a los métodos normales es la dificultad de saber en qué fase de la ejecución se encuentran. Esto es muy inconveniente a la hora de saber que la ejecución de una corrutina ha acabado y que se puede empezar la ejecución de la siguiente. En nuestro proyecto, por ejemplo, es necesario llamar al método *ActivateExternalObjects(…)* después de llamar a *ActivateTiles(…)*. Si no se respeta este orden los objetos podrían aparecer antes que el terreno y caerse debajo del mapa.

Para poder encadenar llamadas a diferentes corrutinas de una manera fácil se ha decidido el uso del patrón de diseño *Command*. Este es un patrón de diseño de comportamiento que consiste en encapsular una acción en un objeto. Este objeto se puede usar para ejecutar la acción mas tarde o en un determinado orden.

Como se puede ver en la Ilustración 19, todos los métodos que son corrutinas, es decir devuelven un *IEnumerator*, también aceptan como parámetro un objeto del mismo tipo. Este parámetro representa otra corrutina que se ejecutará como *callback,* al final de la ejecución de la corrutina actual. Es decir, una corrutina se puede encapsular en un objeto referenciando el objeto *IEnumerator* que devuelve. De esta manera se pueden encadenar llamadas a corrutinas sin la necesidad de usar una estructura compleja con booleanos e *if()* anidados para saber cuando una corrutina acaba su ejecución. El resultado final se puede ver en la Ilustración 20. Cada corrutina posterior que se quiere ejecutar se pasa como argumento a la corrutina anterior.

```
IEnumerator startSequence =
    StartCoroutineSequence(
    _groundGenerator.CreateGround(_baseMap,
    _platformGenerator.CreatePlatforms(_baseMap,
    _externalObjectsPositioner.PositionExternalObjects(_baseMap,
    _tilesFactory.ActivateTiles(_baseMap, this,
    _elementFactory.ActivateExternalObjects(_baseMap,
    EnableExternalObjects(
    EndCoroutineSequence()
    )))))));
StartCoroutine(startSequence);
```

Ilustración 20: Encadenado de corrutinas

Facade

Facade es un patrón de diseño que consiste ocultar una estructura mas compleja detrás de un solo interfaz, en ocasiones limitando la funcionalidad a coste de simplificar el uso de una librería.

Tal como se puede ver en la Ilustración 11, las funcionalidades se encuentran repartidas entre las diferentes clases que las implementa. Ofrecer al usuario acceso directo a estas clases podría tener algunas ventajas, como mas libertad a la hora de ejecutar los diferentes métodos. Sin embargo, supondría un aumento en la dificultad de aprendizaje del uso de las diferentes funciones. Por esta razón, se ha decidido ocultar la estructura interna del proyecto a los usuarios finales.

Este patrón de diseño se ha implementado en el proyecto mediante la extensión únicamente de la clase *MapGenerator* de la clase *MonoBehaviour*. Esta clase también referencia toda la configuración del proyecto y los usuarios solo tienen que interactuar

con esta clase mediante la ventana del *Inspector*, tal como se puede ver en la Ilustración 12. De esta manera el usuario no tiene que preocuparse por añadir varios componentes pequeños ni por como interactúan entre ellos. Con esto se consigue un proyecto que es mas fácil de empezar a utilizar por usuarios novatos, pero no se sacrifica mantenibilidad por intentar hacer un proyecto excesivamente sencillo y que fuera difícil de modificar.

6. Tecnologías

A continuación, se verán a ver algunas tecnologías empleadas en el desarrollo del proyecto. Estas tecnologías se han utilizado, en el caso de Unity, de forma directa en el desarrollo, en el caso de GitHub, como herramienta de apoyo para realizar el control de versiones y en el caso de Trello como herramienta de organización.

Unity

Unity[10] es un motor de videojuegos desarrollado por Unity Technologies. Inicialmente solo permitía desarrollar aplicaciones para Mac, pero poco a poco fue expandido con soporte para diferentes plataformas. Actualmente soporta mas de 25 plataformas y es especialmente popular para desarrollar videojuegos para plataformas móviles iOS y Android.

A diferencia de su principal competidor, Unreal Engine, Unity permite desarrollar mediante una API de scripting en el lenguaje C#. Esto lo hace especialmente atractivo para desarrolladores que vienen del mundo de la programación y fue la principal razón por la que se eligió esta herramienta.

En cuanto al coste, Unity es gratis para uso personal y para uso comercial con un volumen de ingresos inferior a 100000 dólares anuales. Esto, junto con la Asset Store, una tienda online donde cualquiera puede publicar elementos comunes de juego tanto gratis como de pago, lo hace especialmente atractivo para pequeños desarrolladores independientes que están empezando en el mundo del desarrollo de videojuegos.

Por lo tanto, teniendo en cuanta el soporte de lenguajes de programación, coste, atractividad para desarrolladores independientes y Asset Store[11] donde se puedan publicar componentes de juegos, convierte a Unity en el motor de videojuegos ideal para desarrollar este proyecto.

Trello

Trello[12] es una herramienta gratuita que permite crear tableros para organizar ideas, ya sean en listas, o tableros de tipo Kanban. Fue lanzado inicialmente en 2011 en formato web. Actualmente, además de web, dispone de aplicaciones para las principales plataformas móviles y de escritorio.

[10] https://unity.com
[11] https://assetstore.unity.com
[12] https://trello.com

La principal ventaja de Trello frente a sus competidores es su facilidad de uso. Tal como se puede ver en la imagen, es muy intuitivo de usar incluso para personas que no han usado nunca este tipo de herramientas, nada mas iniciar sesión. Sus competidores, Jira, por ejemplo, requiere de una extensa configuración previa a empezar a usar sus tableros.

La facilidad de uso de Trello viene con un coste. Lo hace difícil de adaptar para proyectos o empresas mas grandes, que requieren de algo mas complejo para hacer seguimiento de sus tareas. Sin embargo, esto no afecta en el uso de esta herramienta para el desarrollo de este trabajo.

Todo esto, hace que Trello fuera la herramienta ideal para organizar las diferentes tareas que componen este trabajo y hacer seguimiento de estas.

GitHub

GitHub[13] es una web que permite a los desarrolladores alojar y gestionar su código fuente, ya sea de forma pública o privada, mediante el sistema de control de versiones Git. Hoy en día, GitHub es sinónimo de código abierto, pero también permite gestionar código fuente de forma privada, aunque en su versión gratuita esta modalidad tiene algunas limitaciones para las empresas.

GitHub fue lanzado en abril de 2008 y en su primer año acumuló 46000 repositorios.

Para este proyecto, se eligió GitHub por ser la herramienta mas popular para control de versiones para código abiertos. Esto ayudara a hacer llegar el proyecto al mayor numero de usuarios, que es un objetivo de este trabajo. Además, sus funcionalidades cumplen con los requisitos para almacenar proyectos de tipo Unity, ofreciendo almacenamiento para ficheros grande en forma de Git - LFS[14] en el caso en el que fuera necesario.

La dirección del repositorio publico de este proyecto se encuentra en el siguiente enlace: https://github.com/bizz001/AMG2D

7. Desarrollo

En esta sección se van a describir las diferentes tareas relacionadas con este trabajo. Después, se va a hablar de los aspectos relevantes relacionados con la planificación. Luego se presentarán los diferentes problemas encontrados y soluciones aplicadas. Por ultimo, se va a presentar un caso de uso del producto desarrollado. Finalmente, se va a hablar de la experiencia de uso de las diferentes tecnologías y metodologías.

7.1 Planificación

[13] https://docs.github.com/en/github
[14] https://git-lfs.github.com

Al principio del desarrollo del proyecto se ha realizado una estimación del tiempo necesario para su realización. Este paso ha sido muy importante ya que, aunque se disponían de varios meses hasta la fecha de entrega, compaginar el desarrollo del proyecto con un trabajo a jornada completa reduce de forma considerable las horas disponibles cada semana para dedicar al proyecto.

La estimación inicial se ha dividido en varias tareas principales. Estas son las principales tareas y su estimación, que suman en total 340 horas:

- T1: Aprender el uso de Unity
    - Unas 100 horas totales repartidas de la siguiente manera:
        - 50 horas dedicadas para la realización de un curso online en la plataforma Udemy[15]
        - 50 horas repartidas a lo largo del proyecto para aprender el uso de algunas funcionalidades especificas de Unity, necesarias para su realización.
- T2: Redactar la primera parte de la memoria:
    - introducción y estado del arte
        - 15 horas
    - Especificación de requisitos
        - 30 horas
- T3: Redactar proceso de desarrollo:
    - 10 horas
- T4: Realizar el análisis funcional:
    - 30 horas
- T5: Implementar MVP
    - 100 horas
- T6: Implementar pruebas unitarias
    - 10 horas
- T7: Preparar casos de uso
    - 30 horas
- T8: Redactar parte final de la memoria
    - 15 horas

En cuanto a la carga de trabajo, la dedicación semanal consiste en unas 8 horas totales dedicadas al proyecto de lunes a viernes y otras 8 horas el sábado, dejando el domingo para descansar. Es decir, 16 horas semanales. A lo largo de estos meses también se han tomado 3 semanas de vacaciones del trabajo que se han dedicado al proyecto y en las que he estado más de 10 horas diarias trabajando en dedicación exclusiva al proyecto.

El proyecto se ha desarrollado a lo largo de las semanas comprendidas entre el mes de abril y el mes de agosto.

7.2 Aspectos relevantes

[15] https://www.udemy.com

Precisión de la estimación

La mayor desviación en cuanto a la estimación inicial se ha producido en cuanto a la especificación de requisitos. Concretamente se subestimo el tiempo necesario para estudiar los diferentes videojuegos y herramientas similares. Esta tarea llego a costar aproximadamente el doble de su estimación inicial, unas 60 horas.

Para compensar esta desviación, se ha tenido que reducir el alcance de los casos de uso, preparando solo un ejemplo, en lugar de los dos que se planifico inicialmente.

El resto de las tareas no han sufrido grandes desviaciones.

Orden de las tareas

Generalmente, el orden de las tareas ha sido el mismo que el orden en el que se han presentado en la planificación. Sin embargo, ha habido tareas que se han solapado con otras, tales como la T1 con todas las demás. Durante todo el desarrollo del proyecto no se han dejado de adquirir nuevos conocimientos sobre Unity. También se solaparon la T7 con la T5, debido a la necesidad de usar un caso de uso para probar las diferentes funcionalidades de MVP a medida que se han ido desarrollando.

Pruebas unitarias

Las pruebas unitarias son un tipo de prueba que consiste en probar el resultado de ejecutar la mas pequeña unidad de código, es decir un método. Para una prueba unitaria se especifica unos datos de entrada específicos y se compara el resultado obtenido contra el resultado esperado. Si coinciden, la prueba se considera superada.

Inicialmente se ha planteado implementar ensamblados específicos para las pruebas unitarias. Unity tiene soporte para pruebas unitarias, tanto para lo que llaman *PlayMode* y *EditMode*. Sin embargo, debido a que casi la totalidad del código consiste en corrutinas en lugar de métodos normales, el uso de los proyectos de prueba típicos de Unity no ha sido posible de forma directa, en el tiempo estimado inicialmente. Para poder realizar pruebas unitarias de métodos de tipo corrutinas es necesario invertir tiempo adicional tanto en aprender sobre las técnicas que permitan las pruebas unitarias de estos métodos y también en su implementación. Estas tareas se estimaron a un numero de horas que dejaría el numero total de horas fuera del alcance de un TFG y se decidió realizar las pruebas de forma manual, utilizando un caso de uso, sin automatizar.

Documentación

Una parte muy importante del producto es la documentación de uso. De nada sirve tener un producto mínimo, si no se puede utilizar por los usuarios. Por eso, como parte de la tarea de desarrollar del MVP se ha incluido también el desarrollo de una documentación básica de uso. Esta documentación se entrega junto con esta versión inicial, como parte del fichero README.md del repositorio. De esta manera, al acceder a la pagina web del repositorio lo primero que ven los usuarios es la documentación, tal como se puede ver en la *Ilustración* 21.

También se incluye un fichero LICENSE que estipula el tipo de licencia utilizado, que en este caso es una licencia MIT. Los detalles de esta licencia se incluyen en el Anexo 4 – Acuerdo de licencia.

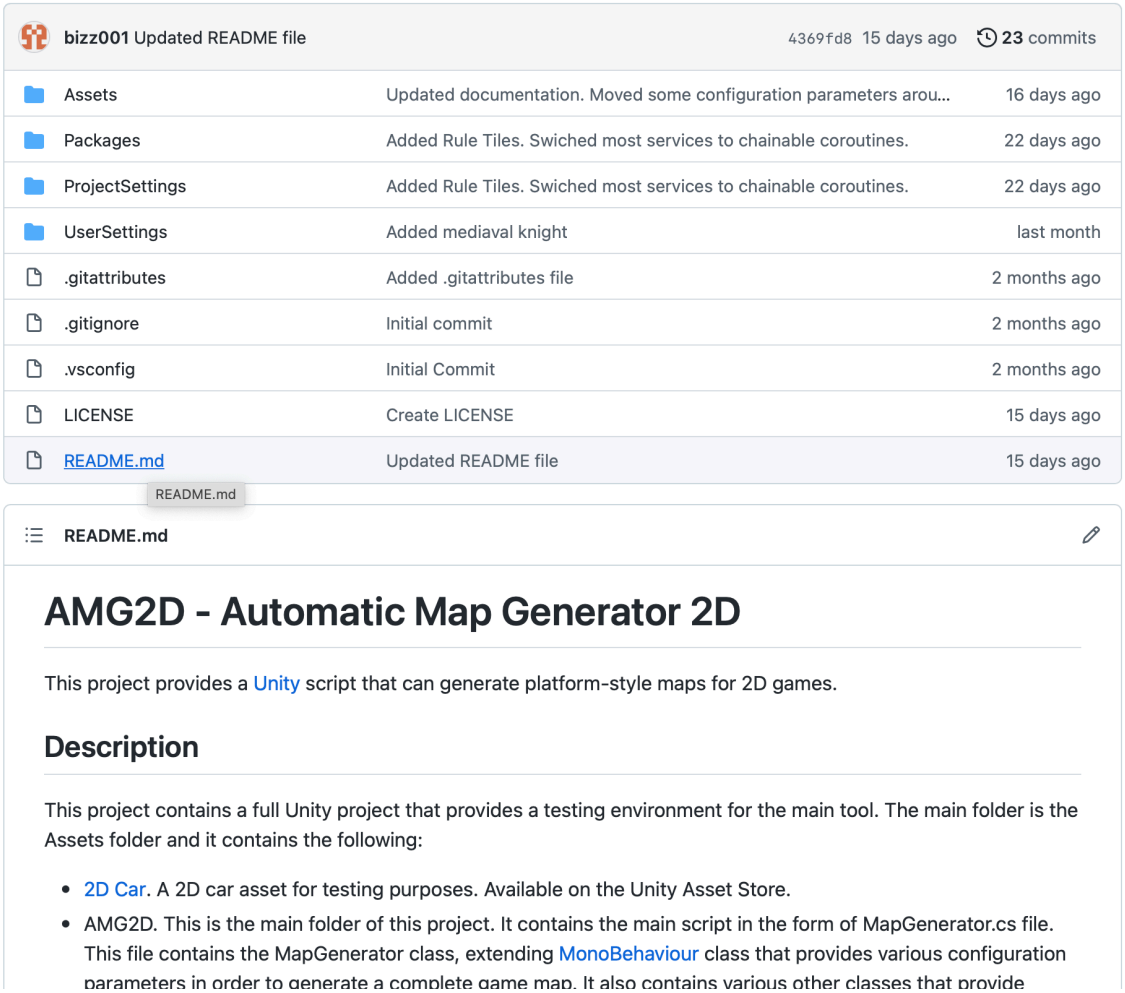

Ilustración 21: Pagina inicial del proyecto en la web de Github

Al final de este documento, en el las secciones 10, 11, 12 se puede ver extrctos de contenido del fichero README.md.

7.3 Contratiempos y soluciones

Efecto *parallax*

El efecto *parallax* se obtiene a partir del desplazamiento de la cámara principal. Cada vez que se mueve la cámara principal, se aplica el mismo movimiento a las diferentes capas, pero alterando dicho movimiento con un valor multiplicativo denominado intensidad del efecto. De esta manera, si la intensidad es mayor que uno, las capas se

desplazarán mas rápido que la cámara y si es menor que uno, se desplazarán mas lento que la cámara. El valor cero indica una capa que no se mueve con respecto al mapa.

En cuanto a la implementación de la funcionalidad del efecto *parallax* se ha tenido una dificultad relacionada con la fluidez del desplazamiento de las capas. Concretamente, al actualizar la posición de las diferentes capas, estas no lo hacían con fluidez. Es decir, había un retraso de un fotograma o dos en la actualización de la posición de las capas que hacía que parezcan temblar.

Para solucionar este problema se ha estudiado de forma profunda la documentación de Unity en cuanto a la actualización de la posición de los objetos. Se encontraron dos problemas.

Por un lado, se descubrió que la actualización del fondo se hacía en el momento inadecuado. Concretamente al actualizar el fondo en el evento *Update()* podría producirse antes de actualizar la posición de la propia cámara para el próximo fotograma, causando que se usara la posición del fotograma anterior en la lógica de actualizar el fondo. Esto contribuyo al efecto de temblor del fondo. Para solucionar esto, se cambio la actualización del fondo al evento *LateUpdate()*. Este evento se invoca después de invocar el *Update()* de todos los elementos de juego, asegurando que la posición de la cámara en el momento del evento es la del próximo fotograma y no del fotograma anterior.

Por otro lado, se descubrió alterar la posición de un elemento de juego no siempre se realiza de forma instantánea, por razones que parecen estar relacionadas con los cálculos internos de efectos físicos. Se ha observado que cuanto mayor el desplazamiento que un objeto realiza con respeto al objeto padre, mayor el efecto no deseado. Esto es debido a que alterar la posición de un objeto modificando de forma directa las coordenadas espaciales esta pensado mas bien para cambios puntuales de posición y no movimientos continuos. Sin embargo, para nuestro caso esta técnica es necesaria para poder seguir de forma instantánea los cambios en la dirección y velocidad de la cámara.

Como solución a este caso se encontró la posibilidad de alterar solo la posición de la textura del objeto, aplicándole un desplazamiento con respecto a la posición del objeto. De esta manera conseguimos alterar la posición visual de un elemento, pero sin alterar la posición del *GameObject* asociado. Este tipo de movimiento es mucho mas ligero y debería eliminar por completo el efecto de temblor.

Sin embargo, debido a que usar la técnica de mover la textura supondrían cambios importantes en la implementación, se decidió dejar esta solución para mejoras futuras. De forma temporal, para limitar el efecto se ha decidido asignar el elemento padre de los diferentes elementos de juego de forma selectiva. Tal como se puede ver en la Ilustración 22, si el efecto *parallax* es superior a 0,5, es decir si la capa del fondo tendrá un movimiento relativo a la cámara inferior que el movimiento relativo al mapa de juego, se le asignará como padre la propia cámara. Si no, el padre del elemento se quedara el por defecto, la propia escena. De esta manera se consigue minimizar el movimiento relativo de los fondos, haciendo que el efecto de temblor del fondo fuera inapreciable.

```
//Create parent to hold all background instances.
var finalPrefab = new GameObject();
if (config.ParallaxIntensity >= 0.5f)
{
    finalPrefab.transform.SetParent(_generalConfig.GeneralMapSettings.Camera.transform);
}
finalPrefab.transform.position = config.BaseImage.transform.position;
config.BaseImage.transform.SetParent(finalPrefab.transform);
```

Ilustración 22: Asignación del elemento padre del fondo

Implementación de *ITilesFactory*

ITilesFactory es el interfaz del servicio que instancia elementos de juego para representar el mapa que se ha generado internamente y se encuentra guardado en un objeto de tipo *MapPersistence*.

El proceso de implementar esta interfaz es el que mejor muestra el proceso de aprendizaje en el uso de Unity. Inicialmente la implementación se realizó utilizando objetos individuales de tipo *GameObject* para cada *Tile*. El manejo de esto objetos es como el de cualquier otro en un lenguaje orientado a objetos. Esto lo hace relativamente fácil de utilizar para alguien quien proviene del mundo de la programación y sin experiencia en programación de videojuegos o Unity.

Esta implementación, que se encuentra en la clase *PooledMapFactory* consiste en un objeto inicial con la función de plantilla. Estos objetos se fijan en la configuración, tal como se puede ver en la Ilustración 23, habiendo uno para cada tipo de *Tile*, tales como suelo, plataforma, cueva etc. A partir de estas plantillas, cada vez que sea necesario, se generan clonas que se fijan en las distintas posiciones del mapa. Cuando tenemos la segmentación habilitada, para mejorar el rendimiento, se utiliza una colección donde se van almacenando los objetos inutilizados para reutilizarlos en lugar de crear objetos nuevos.

```
namespace AMG2D.Configuration
{
    /// <summary>
    /// Class that holds general map configuration parameters.
    /// </summary>
    [Serializable]
    public class GeneralMapConfig
    {
        /// <summary>
        /// Aspects collection for each type of tile.
        /// </summary>
        [SerializeReference]
        public Dictionary<EGameObjectType, GameObject> Aspects;
```

Ilustración 23:Configuración del aspecto de cada Tile

Esta implementación, además de no cumplir con el requisito funcional RF01, no cumple con los Requisitos de rendimiento, causando ralentizaciones en el juego al habilitar la segmentación. Esta ralentización, aunque se aprecia una mejora al implementar la

reutilización de los objetos, sigue siendo sustancial. Por ello, se decidió investigar el uso de los objetos de tipo *TileMap* como alternativa.

El uso de *Tilemaps* tiene varias ventajas. Por empezar, están diseñados justo para este caso de uso. Debido a esto, el rendimiento es superior al de usar *GameObjects*. En segundo lugar, hace que nuestro mapa fuera compatible con *RuleTiles*. Esta funcionalidad de Unity permite simplificar la configuración del proyecto y además facilitar la integración de la herramienta en proyectos existentes.

RuleTiles es una funcionalidad de Unity que consiste en un tipo especial de *asset* que al utilizarse dentro de un *TileMap* permite configurar reglas especiales de aspecto en función de las *Tiles* cercanos, tal como se puede ver en la Ilustración 24: Configuración de RuleTiles. De esta manera se consigue un aspecto que cambia de forma dinámica, simplificando la implementación interna de *ITilesFactory* al no tener que implementar esa funcionalidad.

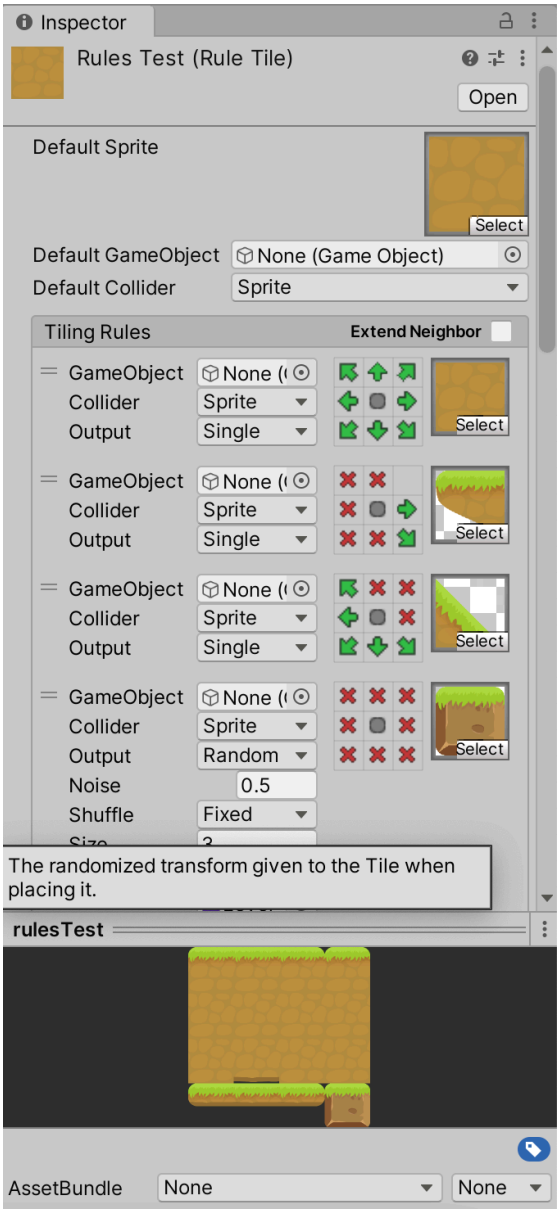

Ilustración 24: Configuración de RuleTiles

Esta nueva implementación de *ITilesFactory* se ha realizado en la clase *TiledMapFactory*. Se crea un *TileMap* para los elementos de mapa de tipo plataforma y otro para los elementos de tipo suelo. Por la manera en la que internamente la clase *TileMap* gestiona los recursos, no es necesario utilizar una colección para almacenar objetos con el propósito de reutilizarlos. Cada *Tile* de *TileMap* que es del mismo tipo referencia siempre el mismo objeto *TileBase* sin crear objetos nuevos.

Optimización de rendimiento

En cuanto al rendimiento, se ha aprendido sobre todo el uso de las herramientas de análisis de rendimiento, o *profiling*. Una herramienta de *profiling* es una aplicación independiente, o componente de un IDE que mide en detalle el rendimiento de un programa en desarrollo. Generalmente se miden aspectos como el uso detallado de CPU que permite ver en cada momento que parte del código necesita mas tiempo para ejecutarse, el uso detallado de memoria con detalles como numero de objetos activos y recursos asignados y otros detalles como uso detallado de red, disco etc.

En la primera fase del desarrollo, se encontraron algunos problemas de rendimiento relacionados con la activación de los segmentos de mapa. Para solucionar estos problemas iniciales se decidió medir el tiempo de forma manual, utilizando la clase *Stopwatch* tal como se puede ver en la *Ilustración* 25 donde el objeto llamado *totalWatch* es de tipo *Stopwatch*. Esta clase permite medir el tiempo que transcurre entre dos puntos de la traza del código.

```
totalWatch.Stop();
var foreachTime = totalWatch.ElapsedMilliseconds;
totalWatch.Restart();

foreach (var tilesLine in tiles)
{
    if (tilesLine.First().IsActive) _segmentParents[tilesLine.First().SegmentNumber].SetActive(true);
}

totalWatch.Stop();
var parentActivation = totalWatch.ElapsedMilliseconds;
totalWatch.Restart();

var activeSegmentColliders = _segmentParents.Where(pair => activeSegments.Contains(pair.Key))
    .Select(pair => pair.Value.GetComponent<CompositeCollider2D>());

totalWatch.Stop();
var colliderSelection = totalWatch.ElapsedMilliseconds;
totalWatch.Restart();

foreach (var collider in activeSegmentColliders)
{
    collider.GenerateGeometry();
}
totalWatch.Stop();
var colliderGeneration = totalWatch.ElapsedMilliseconds;

Debug.Log($"{nameof(ActivateAllTiles)} time: Foreach: {foreachTime}; " +
    $"parentActivation: {parentActivation}; colliderSelection: {colliderSelection}; colliderGeneration: {colliderGeneration}");
return true;
```

Ilustración 25: Uso de la clase Stopwatch

El resultado de ejecutar este código se puede ver en la Ilustración 26. Utilizando esta técnica se consiguió localizar problema en la generación del *collider*.

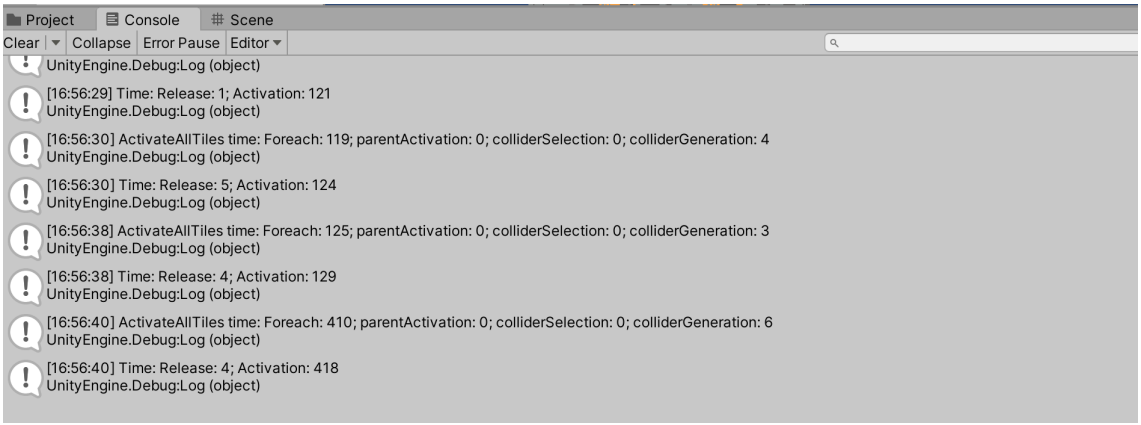

Ilustración 26: mediciones de tiempo con Stopwatch

Aunque esta técnica consiguió dar los resultados esperados, tiene muchos inconvenientes. Por empezar, no se tiene visibilidad del tiempo que no se mide. Por lo tanto, tiene que existir una suposición previa de donde podría estar el retraso para poder localizarlo, pudiéndose tardar horas en localizar un retraso. Esto no ocurre con el *profiler* que mide todos los tiempos y muestra de forma visual cual es la parte que mas tarda.

En segundo lugar, tener que modificar el código supone un riesgo de hacer alguna modificación no planteada y alterar la ejecución de los métodos. Este riesgo es aun mayor cuando el proyecto no dispone de pruebas unitarias para detectar fácilmente los cambios, como es el caso de este proyecto.

En tercer lugar, el simple hecho de añadir nuevos mensajes en consola puede alterar de forma inesperada los tiempos de ejecución, haciendo aún mas difícil localizar el problema.

Por todas estas razones, antes de seguir con el desarrollo se decidió invertir tiempo en el estudio de las herramientas de *profiling* de Unity. Este tiempo resulto bien invertido, ya que, más adelante, la aplicación presentó problemas de tiempos al implementar el uso de *TileMaps*. Esta vez se utilizó el *profiler* de Unity. En cuestión de minutos se localizó el retraso en la manera en la que se realiza la descarga de los *Tiles* cuando un segmento de mapa queda inactivo. En la *Ilustración* 27 se puede ver el pico de mas de 100 milisegundos durante esta fase.

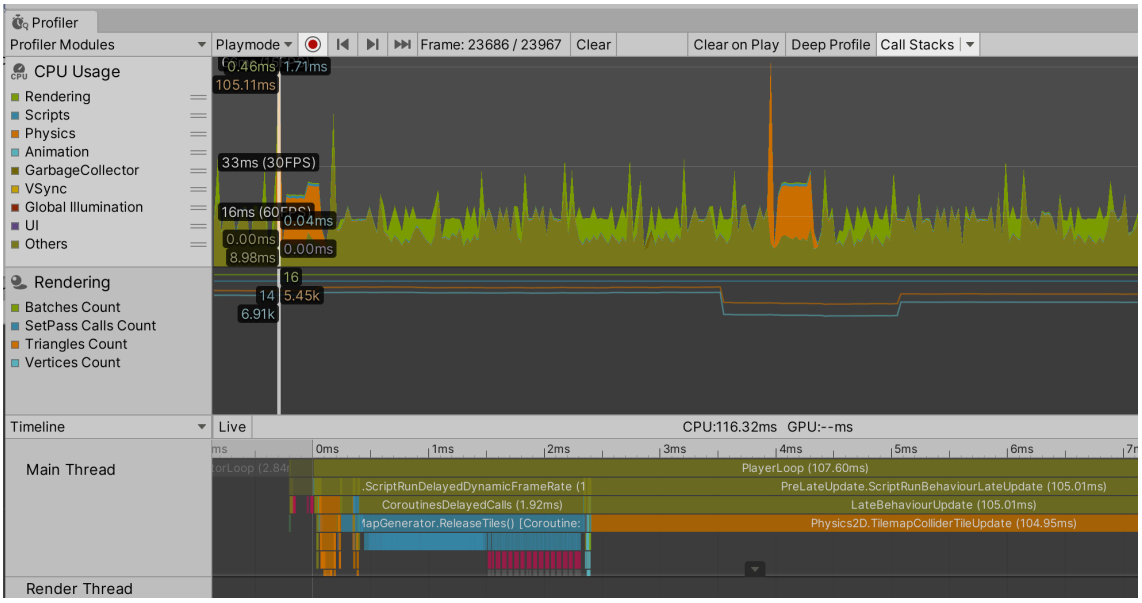

Ilustración 27: Pico de tiempo al modificar segmentos activos

Concretamente el método responsable de realizar esa acción no hacía un uso correcto de las corrutinas, realizando demasiadas acciones entre dos fotogramas. Se modifico el método para repartir su carga entre varios fotogramas, solucionando el problema. Después de implementar estos cambios, el análisis de tiempos queda tal como se muestra en la *Ilustración* 28.

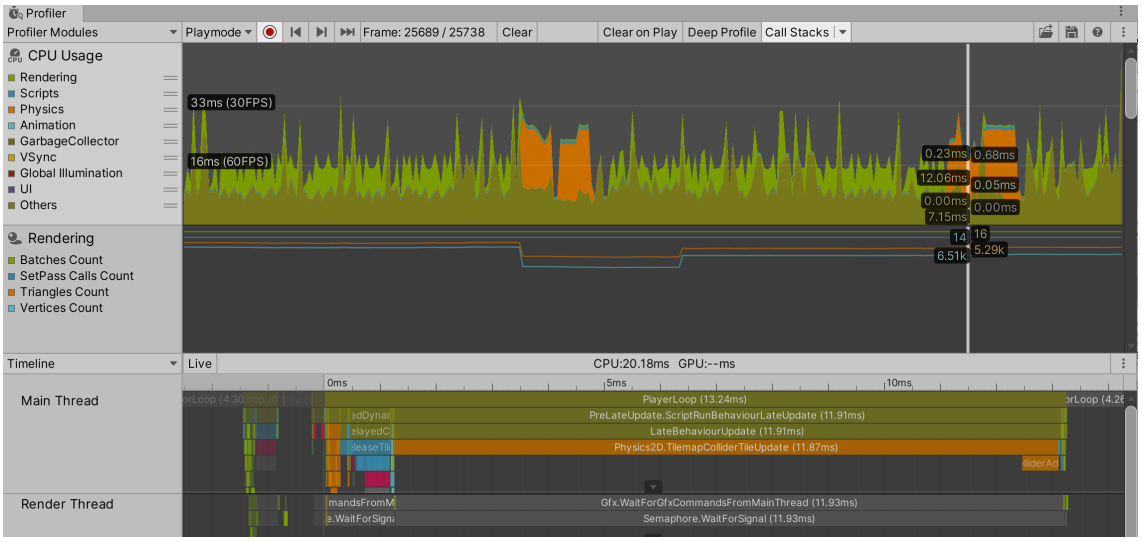

Ilustración 28: Optimización de la descarga de segmentos

El *profiler* también se utilizó para optimizar la configuración de la segmentación. Tal como se puede ver en la *Ilustración* 29 se redujo de forma considerable el tiempo que se tarda en cargar y descargar los segmentos, empleando una configuración con segmentos mas pequeños y reduciendo la velocidad de carga y descarga de segmentos individuales.

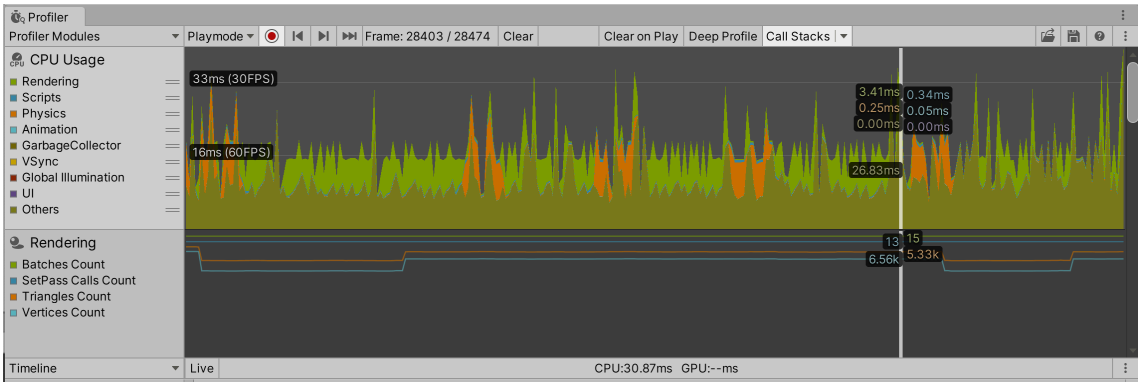

Ilustración 29: Optimización en la configuración

Al realizar todas estas pruebas con el *profiler* se ha identificado un posible comportamiento no deseado. Cuando el personaje se mueve distancias muy cortas hacia adelante y hacia atrás puede causar que el mismo segmento se tenga que cargar y descargar repetidas veces en muy poco tiempo. Aunque es un caso muy concreto, esta acción puede tener un impacto negativo sobre el rendimiento. Para corregir este comportamiento se ha implementado una función de histéresis muy simple y sin ningún tipo de configuración. Al final de cada actualización con éxito de los segmentos activos se guarda el ultimo segmento actual del jugador. Cuando se vuelven a actualizar los segmentos si se intenta activar el segmento que estaba activo anteriormente se omite la actualización. Esto se realiza mediante una simple línea de código que se puede ver en la *Ilustración* 30.

```
//Hysteresis to prevent erratic loading/unloading
if (currentPlayerSegment == _lastTransitionedSegment) yield break;
```

Ilustración 30: Histéresis en la carga de segmentos

7.4 Caso de uso

Para poder probar las diferentes funcionalidades implementadas fue necesario crear un entorno de pruebas. Este entorno consiste en una escena con un mapa de juego libre, o *sandbox* con elementos de juego básicos. Los distintos elementos de juego son lo que aparecen la Ilustración 31.

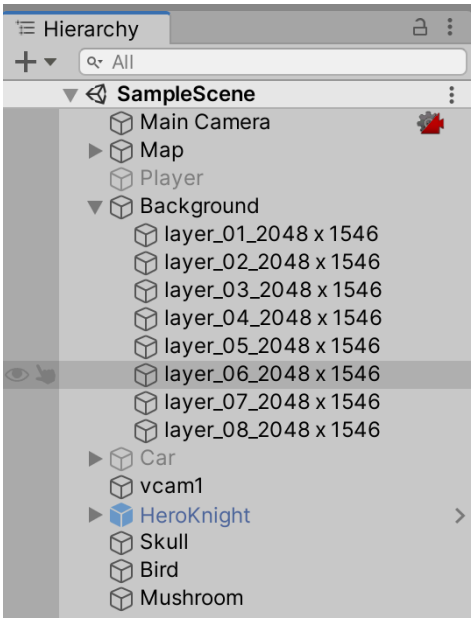

Ilustración 31: Escena de prueba

Los distintos elementos son los siguientes:

Main Camera

La cámara principal de la escena. Se controla mediante una cámara virtual de Cinemachine. Cinemachine es un sistema de control de cámara. Permite controlar la cámara principal con facilidad a nivel básico y al mismo tiempo permite una configuración avanzada de efectos y movimientos de cámara.

Map

Este es el objeto generador de mapa. Se trata de un objeto de tipo *GameObject* vacío al que se le añade como componente el script *MapGenerator*. A partir de este script, mediante la configuración que tiene disponible en sus propiedades publicas, se permite controlar todos los parámetros del mapa generado.

Player

Se trata de un objeto sencillo, con forma de rombo rojo que permite simular un personaje. Tiene un script básico para controlar su movimiento en los dos ejes y también la velocidad. La ventaja que tiene ese elemento es que no interactúa con la gravedad, pudiendo volar. Esto junto con el control de velocidad lo hace muy útil para las pruebas de carga y descarga rápida de segmentos de juego.

Background

Se trata de un objeto simple, con rol de contenedor. Dentro de este objeto se almacenan las diferentes imágenes que se utilizan para el efecto *parallax*. Estas imágenes son las que el script *MapGenerator* del objeto Map tiene que referenciar.

Car

Objeto de tipo coche 2d que se puede mover por el mapa. Sirve como objeto jugador para realizar pruebas. Es parte del *asset* 2D Car[16] descargado de la Asset Store.

Vcam1

Camera virtual de tipo Cinemachine. Esta cámara sigue a uno de los posibles objetos de tipo jugador que esté activo en ese momento. Esta cámara contrala virtual controla los movimientos de la cámara principal.

HeroKnight

Objeto de tipo 2D con aspecto de caballero medieval. Incluye script que le permite moverse por el mapa, saltar, atacar etc. Además, tiene animaciones especificas para cada una de estas acciones. Descargado como parte del *asset* Hero Knight – Pixel Art[17] de la Asset Store.

Skull

Objeto simple con aspecto de calavera. Se utiliza para probar la funcionalidad de colocar objetos externos dentro de la tierra, Es parte del paquete Pixel Adventure 2[18] descargado de la Asset store.

Bird

Objeto simple con aspecto de pájaro. Se utiliza para probar la funcionalidad de colocar objetos externos en el aire, Es parte del paquete Pixel Adventure 2 descargado de la Asset store.

Mushroom

Objeto simple con aspecto de seta. Se utiliza para probar la funcionalidad de colocar objetos externos en el en suelo, Es parte del paquete Pixel Adventure 2 descargado de la Asset store. De todos los objetos utilizados de este paquete, es el único que tiene *collider* e interactúa con el motor físico del juego, incluyendo la gravedad.

Otro *asset* importante de esta escena, pero que no se incluye como objeto de forma directa en la misma, es el *asset* de tipo Rule Tile que determina el aspecto del mapa entero. Este incluye reglas para mostrar dieciocho aspectos diferentes, en función de la presencia de sus vecinos. En la Ilustración 32 se muestran algunos ejemplos.

[16] https://assetstore.unity.com/packages/tools/physics/2d-car-73763
[17] https://assetstore.unity.com/packages/2d/characters/hero-knight-pixel-art-165188
[18] https://assetstore.unity.com/packages/2d/characters/pixel-adventure-2-155418

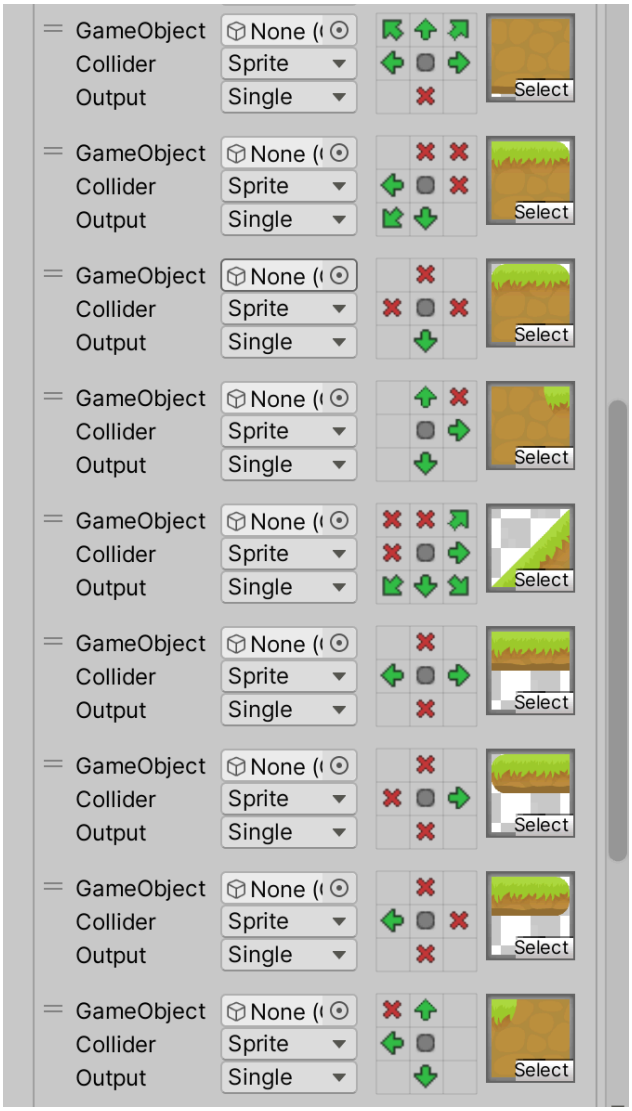

Ilustración 32: Ejemplo Rule Tiles

La escena quedaría tal como se ve en la Ilustración 33. En la parte superior de la imagen se puede ver la vista final de juego, mientras que en la parte de abajo tenemos una visión mas amplia de la escena. Ahí se puede ver la ilusión del fondo con sus diferentes capas, que finaliza a poco después del limite de la cámara. También se puede ver coincide aproximadamente con la región donde acaba el ultimo segmento activo del mapa. A medida que el jugador avanza hacia la derecha, el fondo se irá desplazando y aparecerán nuevos segmentos del mapa, hasta el limite establecido en la configuración.

Ilustración 33: Escena de prueba

7.5 Kanban y GitFlow en la practica

El tablero Kanban ha sido muy útil en la gestión de las tareas de este proyecto. Aunque no hay un equipo involucrado para necesitar coordinación, es importante tener controladas las diferentes tareas para no perder de vista alguna tarea importante que pueda causar un retraso en acabar el proyecto. En la Ilustración 34 se puede ver una imagen de como quedaban organizadas las tareas en una fase intermedia del desarrollo del proyecto.

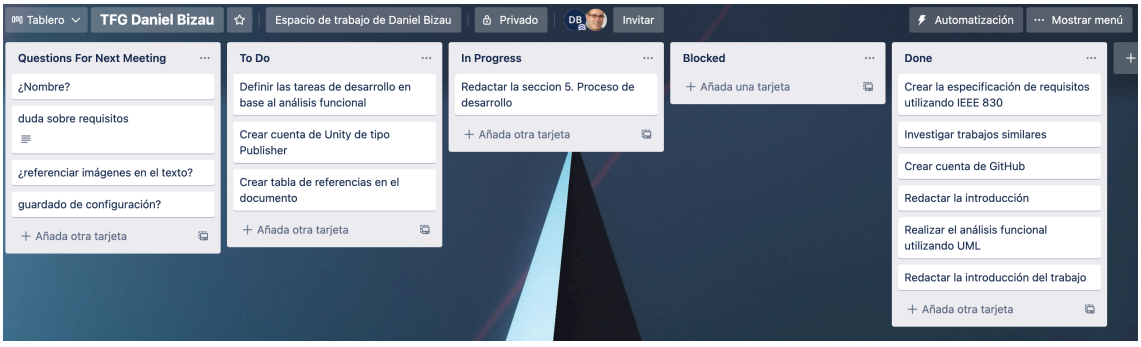

Ilustración 34: Uso de Trello en el desarrollo

Tal como se puede ver, se han utilizado tanto columnas para el ciclo de vida de las tareas, *To Do, In Progress, Blocked, Done*, como una columna adicional. Esta columna, *Questions For Next Meeting,* no contiene tareas, sino apuntes de las diferentes dudas que han surgido, para aclararlas en la próxima reunión con el tutor del proyecto.

En cuanto a Gitflow, se ha conseguido el objetivo de practicar el uso de este flujo de trabajo. Todos los *commits* durante el desarrollo se han realizado en ramas tales como *feature/ImplementParallaxBackground, feature/MigrateToTileMaps* etc. Una vez finalizado el desarrollo de una característica, la rama correspondiente se ha fusionado en la rama *develop* cuando el proyecto tenia implementadas todas las características necesarias para el *MVP*, se ha creado la rama *release/v1.0.0*. En esta rama se han ido aplicando los cambios necesarios para preparar el lanzamiento de la primera versión. Aquí se han realizado pequeñas modificaciones, tales como añadir documentación de uso, cuyos extractos se pueden ver más adelante en los anexos, documentación interna del código etc. Cuando la rama estaba preparada, se ha fusionado en la rama *master*.

Como resultado de la experiencia de practicar el uso de Gitflow se han aprendido varias cosas. En primer lugar, es importante que al fusionar desde las ramas *feature* hacia la rama *develop* se realice utilizando un *commit* de fusión. Esto se consigue añadiendo la opción *--no-ff* al ejecutar el comando *merge*. De esta manera se preserva la topología de las ramas, dejando reflejadas todas las fusiones en el historial. Normalmente este paso se realiza de forma automática por la herramienta con la que se realiza el *pull request*. En la Ilustración 35 se puede ver de forma visual cual es el efecto de utilizar, o no, *--no-ff*.

Gitflow está diseñado para que un equipo de varios desarrolladores trabaje sobre un mismo proyecto al mismo tiempo. Mas que por su propia funcionalidad, que en un equipo de solo un miembro no aporta mas que otros flujos mas simples, inicialmente fue elegido solo por su popularidad en las empresas de desarrollo software. Sin embargo, desarrollar cada característica en una rama propia ha sido muy útil en ayudar a centrar el esfuerzo en una sola tarea, para poder así alcanzar los distintos hitos del proyecto, Sin distraerse por cambios menores en características ya desarrolladas.

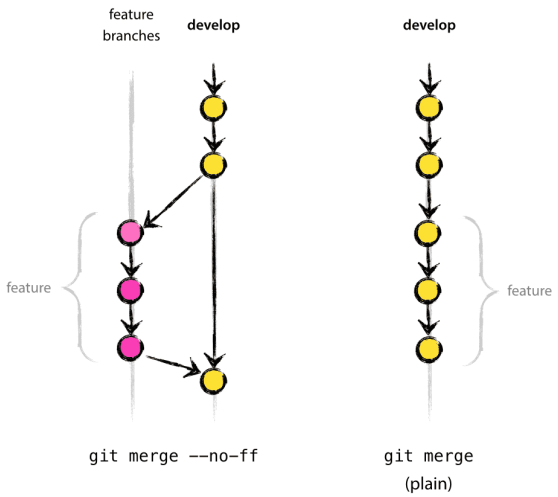

Ilustración 35: comando git merge –no-ff

8. Conclusiones y trabajos futuros

Los objetivos del proyecto han sido alcanzados en su totalidad. En la sección 3 se han definido los requisitos para una herramienta que permite a los desarrolladores de videojuegos agilizar el proceso de desarrollo. Esta herramienta permite generar de forma automática mapas 2D. Permite la integración con componentes ya desarrollados, siendo capaz de generar objetos a partir de cualquier plantilla tipo de *GameObject* y además permite crear con facilidad fondos con efecto *parallax*.

Cada componente del mapa generado se coloca de forma determinista para una configuración especifica y permite variar el mapa generado para una misma configuración, alterando el numero de semilla. La totalidad de los parámetros se puede ver en el Anexo 2 – Configuración. Para un usuario novato, alterar los diferentes parámetros de configuración puede ser muy útil en el aprendizaje de conceptos tales como *parallax, ruido Perlin,* o *semilla de generación pseudoaleatoria*.

Además, en el desarrollo se han cuidado la estructura del código y la documentación interna de las diferentes partes para facilitar el mantenimiento y expansión de la herramienta por personas que no han participado en el desarrollo inicial. La explicación detallada de la arquitectura y los patrones de diseño se encuentran en las secciones

Todo esto, junto con una documentación básica de uso se encuentra alojado en un repositorio en GitHub bajo una licencia MIT, para ser fácil accesible a los desarrolladores de videojuegos.

A nivel personal, también se han alcanzado los objetivos propuestos. Se ha aprendido el uso de Unity a nivel básico, adquiriendo conocimientos sobre el tipo *GameObject* componentes, *assets, scripts* etc. También se han aprendido algunos conceptos mas avanzados tales como corrutinas, Cinemachine, *Tilemaps, RuleTile* etc. Además de aprender sobre Unity, se han adquirido conocimientos sobre el uso de Gitflow y la importancia de depurar aplicaciones utilizando herramientas de tipo *profiler,* entre otras.

Para el futuro, hay varias líneas de mejoras que se pueden realizar. La primera seria actualizar las especificaciones en base al *feedback* de los *early adopters* al utilizar el *MVP* implementado. Para realizar eso es necesario hacer llegar este *MVP* a diferentes desarrolladores interesados en probarlo. Solo con dejar el repositorio publico en GitHub no es suficiente, ya que nadie sabrá que existe. Esto se podría hacer publicando el repositorio en foros de especialidad, o incluso contactando de forma directa mediante correo electrónico u otras vías a pequeñas empresas *indie*.

Publicar el *MVP* en la Asset Store también puede ser una vía de ganar visibilidad y recibir *feedback*. Sin embargo, para este paso se necesita revisar de forma exhaustiva los requisitos de Unity en cuanto a publicaciones nuevas en este portal.

De forma paralela a la actualización de los requisitos se debería dotar al proyecto de pruebas unitarias. Para ellos se necesitan adquirir los conocimientos necesarios sobre

como integrar corrutinas en proyectos de pruebas unitarias. Este paso es muy importante para poder cumplir con los requisitos de fiabilidad marcados en la sección 3.4.3.

Una vez cumplidos los puntos de arriba, implementado mas características el producto puede salir del estado de MVP, llegando a un producto final, publicado en la Asset Store.

A nivel personal este proyecto ha sido un reto. Compaginar el trabajo a jornada completa y realizar el proyecto por la tarde y fin de semana ha sido un ejercicio de compromiso y un reto en organización de tiempo. Sin embargo, fue una experiencia única al poder disfrutar por primera vez de mi pasión por los videojuegos como algo mas que un simple pasatiempos. En el futuro espero poder realizar proyectos similares, ya sean personales o a nivel profesional.

9. Apéndice de vocabulario

A continuación, se van a definir varios términos específicos del área de conocimiento de este trabajo.

- Desarrollador indie:
  - Desarrollador de videojuegos que trabaja de forma independiente, o en una compañía pequeña.
- Asset
  - Cualquier archivo que Unity soporte y que pueda servir para la creación de un videojuego.
- Game Jam
  - Concurso en el que los participantes tienen que crear un juego desde cero en un tiempo muy limitado, generalmente de 24 a 72 horas, siguiendo las reglas de los organizadores en cuanto a temática u otras restricciones.
- Framework
  - Cualquier componente software que se integra en el producto final, ya sea como API, plantilla de código, referencia de ensamblado etc., que ayuda en agilizar el proceso de desarrollo
- Collider
  - Es un elemento invisible del mundo de juego que permite a un objeto interaccionar con otros objetos y con el motor físico del juego, determinando efectos de colisión.
- Refactorización
  - Es la acción de cambiar la estructura interna de una aplicación o de un fragmento de código, sin alterar su funcionalidad.
- Tilemap
  - Es un componente de Unity que consiste en un sistema que permite el manejo de elementos de tipo Tile con el propósito de crear mapas 2D.
- Tiles
  - Son Assets de Unity que se utilizan para crear mapas 2D al posicionarlos en objetos de tipo Tilemap de forma ordenada.
- RF – Requisito funcional

    - Una descripción cuantificable del comportamiento del software para unos datos de entrada especificados
- GameObject
    - Son los objetos fundamentales de una escena de juego, que actúan como contenedores para los diferentes tipos de componentes de juego.
- Parallax
    - Parallax, o efecto paralaje, es el efecto óptico que hace que un objeto mas lejano parezca que se mueve mas lentamente que un objeto cercano.
- Renderización
    - Es el proceso, realizado por un software, mediante cual se generan imágenes digitales a partir de modelos informáticos.
- Checkbox
    - Es un elemento de interfaz que permite al usuario realizar una elección binaria, indicada normalmente mediante la presencia o ausencia de un símbolo dentro de una forma cuadrada.
- Juego AAA
    - Termino que se utiliza para clasificar juegos que tienen presupuestos de desarrollo y marketing muy por encima de un juego típico.
- Power-up
    - Up objeto que el jugador puede recoger que le otorga alguna ventaja para conseguir el objetivo del juego.
- Tipo bool/booleano
    - Un tipo de dato que representa dos posibles valores, normalmente representados por *verdadero* y *falso*.
- IDE
    - IDE, o *Integrated Development Enviornment,* es una aplicación que proporciona al programador varias funcionalidades con el propósito de maximizar su productividad.
- Feedback
    - Es la información ofrecida sobre un proyecto por entidades no involucradas de forma directa en el desarrollo, que se utiliza para mejorar el resultado del proyecto.
- Stakeholder
    - Entidades que tienen un interés en el resultado final de un proyecto.
- Pull request
    - Es el proceso mediante cual el desarrollo realizado en una rama Git se revisa, normalmente por otro desarrollador del equipo, antes de fusionar en otra rama.

10. Anexo 1 – Estructura del proyecto

En este anexo se muestra un fragmento del fichero README.md que habla sobre la estructura del proyecto. Igual que todo el contenido del proyecto, está escrito en el idioma ingles con el propósito de alcanzar el mayor publico internacional.

La dirección del repositorio publico de este proyecto se encuentra en el siguiente enlace: https://github.com/bizz001/AMG2D

AMG2D – Automatic Map Generator 2D

This project provides a Unity script that can generate platform-style maps for 2D games.

Description

This project contains a full Unity project that provides a testing environment for the main tool. The main folder is the Assets folder, and it contains the following:

- 2D Car. A 2D car asset for testing purposes. Available on the Unity Asset Store.
- AMG2D. This is the main folder of this project. It contains the main script in the form of MapGenerator.cs file. This file contains the MapGenerator class, extending MonoBehaviour class that provides various configuration parameters in order to generate a complete game map. It also contains various other classes that provide different functionalities to the main class.
- Hero Knight – Pixel Art. A 2D character for testing purposes. Available on the Unity Asset Store.
- Pixel Adventure 2 An asset package for testing purposes. Available on the Unity Asset Store.
- Platformer Tileset. A Tileset package for testing purposes. Available on the Unity Asset Store.

11. Anexo 2 – Configuración

En este anexo se muestra un fragmento del fichero README.md que habla sobre los diferentes parámetros de configuración. Igual que todo el contenido del proyecto, esta en el idioma ingles con el propósito de alcanzar el mayor publico internacional.

La dirección del repositorio publico de este proyecto se encuentra en el siguiente enlace: https://github.com/bizz001/AMG2D

The configuration parameters include the following:

General configuration

Name	Type	Description

Height	int	Height of the generated map.
Width	int	Width of the generated map.
Map Border Thickness	int	Thickness of the map border. If set to 0 the map will have no border.
Ground Tile	TileBase	TileBase object reference that will be used to complete ground tiles.
Platform Tile	TileBase	TileBase object reference that will be used to complete ground tiles.
Generation Seed	int	Generation seed of the entire map. Each seed will generate a deterministic random map.
Camera	GameObject	Camera object for movement tracking. Used to determine active segments when segmentation is enabled.
Enable segmentation	bool	Indicates whether map segmentation should be enabled.
Segment Size	int	Size in tiles of each individual segment.
Number of segments	int	Number of segments that will compose the map.
Segment loading speed	int	Speed at which each individual segment is loaded. Setting this value to a number greater than the value of SegmentSize parameter will cause segments to load instantly. Might affect performance.
Objects to enable	List {GameObject}	List of external objects that will be set to active once the map is finished loading. Useful for enabling player character for example so that it appears when map is fully loaded.

Background Configuration

Name	Type	Description
Vertical Parallax Modifier	float	Value by which the parallax intensity of each layer is multiplied with in order to apply vertical parallax. Recommended values between 1.0 and 0.1

Horizon height	float	Height of the horizon as a proportion of the total height. 0.5 will set the horizon in the middle of the map height.
Map padding	int	Vertical background padding relative to the map vertical size.
Background layers	BackgroundLayerConfig[]	Array containing the background configuration of each parallax layer.

Background layer configuration

Name	Type	Description
Base image	GameObject	GameObject reference representing the image to be set as layer. The user is responsible of setting the Sorting Order of base image of each layer so that the layers with lower parallax intensity appear behind layers with higher parallax intensity.
Parallax intensity	float	The intensity of the parallax effect. Must be a value between -1 and 1. A value of 0 means the layer will be static relative to the map. A value of 1 means the layer will be static relative to the camera. A value less than 0 means the layer will be in front of the map.
Repetition	int	Number of times to repeat the layer in order to cover all camera view. Increase this value if you see the edge of the background when moving.

Ground configuration

Name	Type	Description
Initial height	int	The starting and the average height of the generated ground.
Smoothness	float	The smoothness of the generated ground.

Platforms configuration

Name	Type	Description
Enable platforms generation	bool	Indicates whether platform generation should be enabled.
Maximum height	int	Indicates the maximum height, in tiles, of the platforms.
Minimum height	int	Indicates the minimum height, in tiles, of the platform.
Thickness	int	Indicates the thickness of the platforms. Tiles.
Density	int	Indicates the density of the generated platforms.
Min width	int	Indicates the minimum width of the generated platforms.
Max width	int	Indicates the maximum width of the generated platforms.

External objects configuration

External objects are configured using an array of External objects configuration:

Name	Type	Description
Unique ID	string	Unique ID of this External object configuration
Object Template	GameObject	Object template that will be cloned across the map.
Density	int	Density of the objects spawned.

12. Anexo 3 – Guía rápida

En este anexo se muestra un fragmento del fichero README.md que describe una guía rápida de uso. La carpeta AMG2D y la configuración a la que hace referencia se incluyen en las secciones 10 y 11.

La dirección del repositorio publico de este proyecto se encuentra en el siguiente enlace: https://github.com/bizz001/AMG2D

Getting Started

For now, in order to use this project, simply download or clone the repository and then copy the AMD2D folder from this project into your project. Then add the MapGenerator script as a Component to a blank GameObject and configure the generator acording to your needs based on the parameters documentation listed above.

13. Anexo 4 – Acuerdo de licencia

Este anexo contiene la información de licencia del repositorio de alojado en Github y localizada en el fichero LICENSE. Se muestra en su texto original, en ingles.

La dirección del repositorio publico de este proyecto se encuentra en el siguiente enlace: https://github.com/bizz001/AMG2D

14. Referencias

1. *Case study the indie game industry and help indie developers achieve their success in digital marketing*. **Wu, Mengling.** Boston, Massachusetts : Northeastern University, 2017.

2. *An Analog History of Procedural Content Generation*. **Smith, Gillian.** Pacific Grove, Ca, USA : 10th International Conference on the Foundations of Digital Games, 2015. 978-0-9913982-4-9.

3. *Procedural content generation for games: A survey*. **Hendrikx, Mark, et al.** 1, Netherlands : ACM Transactions on Multimedia Computing, Communications, and Applications, 2013, Vol. 9. DOI = 10.1145/0000000.0000000.

4. *A Survey of Procedural Noise Functions*. **Lagae, Ares, et al.** 8, s.l. : Computer Graphics Forum, 2010, Vol. 29. DOI: 10.1111/j.1467-8659.2010.01827.x.

5. *An image synthesizer*. **Perlin, Ken.** 3, New York, NY, USA : Association for Computing Machinery, 1985, Vol. 19. doi 10.1145/325165.325247.

6. *Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game*. **Kandida Ginting, Anirma, et al.** s.l. : The International Conference on Computer Science and Applied Mathematic, 2019. doi:10.1088/1742-6596/1255/1/012064.

7. *830-1998 - IEEE Recommended Practice for Software Requirements Specifications*. **s.l. : IEEE , 1998. doi: 10.1109/IEEESTD.1998.88286.**

8. *MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product.* Lenarduzzi, Valentina and Taibi, Davide. 39100 Bolzano/Bozen - Italy : Free University of Bolzano/Bozen, 2016. DOI 10.1109/SEAA.2016.56.

9. *On the benefits and challenges of using kanban in software engineering: a structured synthesis study.* Medeiros dos Santos, Paulo Sergio, Caetano Beltrão, Alessandro and Pedraça de Souza, Bruno. 13, Rio de Janeiro, Brazil : Journal of Software Engineering Research and Development, 2018, Vol. 6. doi 10.1186/s40411-018-0057-1.

10. *Kanban in software development: A systematic literature review.* Ovais Ahmad, Muhammad, Markkula, Jouni and Ovio, Markku. Finland : Department of Information Processing Science, University of Oulu, 2013. DOI 10.1109/SEAA.2013.28.

11. *Git.* Spinellis, Diomidis. 3, s.l. : IEEE Software, 2012, Vol. 29. doi 10.1109/MS.2012.61.

12. *GitFlow: flow revision management for software-defined networks.* Dwaraki, Abhishek, et al. New York, NY, USA : Association for Computing Machinery, 2015. doi: 10.1145/2774993.2775064.