Calificaciones
David Casiano Flores: 8.5
Sol Flora López Antón: 8.5

Fulvinter Maker: Editor de niveles para videojuegos 2.5D
Fulvinter Maker: Level editor for 2.5D video games

Trabajo de Fin de Grado
Curso 2024–2025

Autores
David Casiano Flores
Sol Flora López Antón

Director
Ismael Sagredo Olivenza

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Fulvinter Maker: Editor de niveles para videojuegos 2.5D
Fulvinter Maker: Level editor for 2.5D video games

Trabajo de Fin de Grado en Desarrollo en Videojuegos

Autores
David Casiano Flores
Sol Flora López Antón

Director
Ismael Sagredo Olivenza

Convocatoria: *Septiembre* 2025

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

23 de septiembre de 2025

Agradecimientos

Queremos agradecer a nuestro tutor Ismael por toda la ayuda que nos ha brindado durante el desarrollo del proyecto, ofreciéndonos reuniones incluso durante sus vacaciones.

"Quiero agradecer a mis padres por su apoyo cada día, a mi abuela por confiar siempre en mí, a mi hermana por los paseos de las tardes con el perro para que me diera un poco el aire, y a los xiquets por aguantarme en las partidas nocturnas de Rainbow Six para desconectar."—David Casiano Flores

"A mis padres, por apoyarme siempre pase lo que pase, y por su paciencia. A Morgan, por cuidarme cada día y hacer que siga adelante, pase lo que pase. A Laurel por traerme luz y vida con todo su cariño. A Chloe por siempre hacerme sonreir."—Sol FLora López Antón

Resumen

Fulvinter Maker: Editor de niveles para videojuegos 2.5D

El diseño de niveles es una de las tareas más laboriosas en el desarrollo de un videojuego, especialmente en los géneros de plataformas o metroidvania. En este aspecto, el editor de Unity puede resultar insuficiente, pues no proporciona las herramientas que facilitan este proceso.

Este Trabajo de Fin de Grado tiene como objetivo el desarrollo de una herramienta para la edición de niveles de videojuegos con lógica 2D en Unity. La herramienta proporciona una interfaz gráfica que permite crear y modificar una cuadrícula sobre la que se pueden colocar o eliminar elementos del nivel, como estructuras del escenario, enemigos u objetos interactuables.

Aunque su desarrollo se ha orientado inicialmente a su integración con el videojuego *Fulvinter*, se ha priorizado la independencia y reutilización de la herramienta, permitiendo emplearla en otros proyectos similares en 2D o 2.5D que cumplan ciertos requisitos técnicos. El resultado es un paquete de Unity que puede ser importado en distintos proyectos de juegos en desarrollo, facilitando el trabajo de diseñadores y desarrolladores a la hora de construir niveles de forma más ágil y eficiente. El paquete puede ser descargado desde nuestro repositorio de Github[1].

Para asegurar el correcto diseño y funcionamiento de nuestro editor de niveles, queremos realizar en un futuro pruebas de usuario, de manera que podamos detectar los puntos débiles de la herramienta para poder solucionarlos.

Palabras clave

Editor, Diseño, Nivel, Unity, Videojuego, *scroll* lateral, 2D, 2.5D, *Fulvinter*, Paquete

[1] https://github.com/dcasiano/TFG-EditorNiveles

Abstract

Fulvinter Maker: Level editor for 2.5D video games

Level design is one of the most labor-intensive tasks in video game development, especially in platformer and metroidvania genres. In this regard, Unity's built-in editor can be insufficient, as it does not provide the tools that streamline this process.

This Bachelor's Thesis aims to develop a tool for editing 2D logic-based video game levels in Unity. The tool provides a graphical interface that allows users to create and modify a grid on which level elements—such as environment structures, enemies, or interactive objects—can be placed or removed.

Although its development was initially oriented toward integration with the video game *Fulvinter*, the tool's independence and reusability were prioritized, making it suitable for use in other similar 2D or 2.5D projects that meet certain technical requirements. The result is a Unity package that can be imported into different game development projects, streamlining the work of designers and developers when building levels more quickly and efficiently. You can download this package from our Github repository[2].

To ensure the proper design and functionality of our level editor, we intend to conduct user testing in the future, allowing us to identify the weaknesses of the tool and address them accordingly.

Keywords

Editor, Design, Level, Unity, Video game, side-scrolling, 2D, 2.5D, *Fulvinter*, Package

[2]https://github.com/dcasiano/TFG-EditorNiveles

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El diseño de niveles es una de las tareas más laboriosas en el desarrollo de un videojuego, especialmente en aquellos tipos de juegos en los que el propio nivel es parte de la mecánica jugable, como son los juegos de plataformas, los metroidvanias —nombre popular que se otorga a los juegos cuyas mecánicas se inspiran en *Super Metroid* o *Castlevania: Shymphony of the Night*—. En este tipo de juegos, disponer de un editor específico es crucial para reducir los tiempos de desarrollo y permitir la iteración de los niveles de forma rápida. En este contexto, nuestro trabajo presenta ***Fulvinter Maker***, un editor para juegos 2D/2.5 D de scroll lateral, inspirado en el editor utilizado en el videojuego en desarrollo *Fulvinter*, pero integrado dentro del propio motor de desarrollo Unity. Nuestro editor permite crear juegos con las características mencionadas de forma más rápida que con las herramientas que ofrece el propio motor. A pesar del nombre, el editor está diseñado para que pueda ser utilizado en cualquier tipo de juego y no específicamente para *Fulvinter*, aunque bebe de las ideas que el equipo de desarrollo del juego consideró necesarias para su editor externo.

1.1. Motivación

Durante las dos últimas décadas, en la escena del desarrollo de videojuegos se ha popularizado progresivamente el uso de motores de videojuegos comerciales multiplataforma. En el año 2024, el 90 % de los juegos lanzados en el portal de videojuegos Steam fueron desarrollados mediante motores de dicha categoría (Video Game Insights, 2025). Esto se debe a diferentes motivos. Uno de ellos es el auge de los denominados juegos independientes o *indies* (Obedkov, 2024), los cuales son desarrollados por pequeños grupos de trabajo, normalmente con pocos recursos de capital disponibles. Las empresas dedicadas a este sector no acostumbran a crear su motor de videojuegos propio para desarrollar sus juegos, sino que eligen alternativas como Unity o Unreal Engine, motores comerciales y accesibles al público. Debido a esto último, las compañías propietarias de estos motores han cambiado su modelo de negocio de un modelo de pago por uso a otro de pago por beneficios,

lo cual democratiza su acceso. Asimismo, estos motores ofrecen múltiples ventajas: reducen el tiempo de desarrollo, permiten a los estudios independientes acceder a tecnologías que, de otro modo, les resultaría muy costoso obtener por sus propios medios, y son motores que, tras años de optimizaciones y mejoras, cuentan con un amplio conjunto de herramientas que facilitan su configuración y adaptación a las necesidades específicas de cada estudio.

Otro de los motivos de la popularidad de estos motores es el aumento del costo económico del desarrollo de los videojuegos. Por esto, es una buena estrategia comercial lanzar los juegos en diversas plataformas, como sistemas Windows, consolas de videojuegos o dispositivos móviles. Estos motores facilitan dicha labor, ya que cuentan con acuerdos con distintas plataformas para integrar sus APIs y simplificar el desarrollo multiplataforma.

Pero, aunque estos motores son muy versátiles y disponen de multitud de herramientas para configurarlos, algunos tipos de juegos requieren herramientas muy específicas que faciliten su desarrollo. Este es el caso de los juegos de plataformas o los metroidvania, uno de los géneros más prolíficos de los últimos años. Este tipo de juegos, muy dependientes del diseño de niveles, son propensos a cambios incluso en etapas posteriores al diseño de *whitebox* - fase en la que se prototipa el juego sin incluir elementos gráficos-, por lo que crear los niveles empleando bloques reutilizables supone una ventaja a la hora de modificarlos durante las fases de pulido. Un editor que permita colocar y ajustar los elementos y objetos del nivel en las casillas de una gran cuadrícula facilitaría considerablemente el diseño de niveles. Esto lo respaldan declaraciones como las de Takashi Tezuka, diseñador de múltiples videojuegos de Nintendo, entre los que se encuentran una gran cantidad de juegos de plataformas 2D de la saga *Super Mario Bros.* Tezuka afirmó que desearía haber contado con una herramienta como *Super Mario Maker*, la cual cumple con las características citadas anteriormente, durante el desarrollo de los primeros juegos de la franquicia de "Super Mario Bros" (Webster, 2019). Sin embargo, los editores que incluyen por defecto la mayoría de los motores son insuficientes en este aspecto, por lo que el desarrollo de dichos videojuegos puede resultar una tarea tediosa.

Este es el caso de *Fulvinter*, un juego de plataformas cooperativo inspirado en la mitología nórdica desarrollado en Unity. Este motor es muy versátil, pero no dispone de herramientas para ajustar fácilmente piezas o bloques básicos para la construcción de niveles. Estas piezas pueden ser denominadas tiles o voxels si corresponden a recursos puramente 2D o 3D, respectivamente. El uso de estas unidades está muy extendido en juegos 2D, pero no tanto en los 3D. Sin embargo, en juegos de lógica 2D pero con gráficos 3D, comúnmente denominados juegos 2.5D, este tipo de funcionalidad simplifica mucho la edición de los niveles y su posterior modificación. Podemos intuir esta problemática, y la ventaja que otorga un editor personalizado basado en tiles o voxels, si nos fijamos en el videojuego *Super Mario Maker*, un editor de niveles convertido en juego que se publicó en las plataformas de Nintendo Wii U, 3DS y, con su secuela, Nintendo Switch.

Durante el desarrollo de *Fulvinter* se recurrió a herramientas externas de edición de niveles, como Tiled, debido a la ausencia de funcionalidades similares en Unity. Haciendo uso de dicho programa se creaban los niveles, compuestos por múltiples

objetos colocados individualmente en casillas y organizados en diferentes capas. Posteriormente, se generaba un fichero que contenía toda la información del nivel, el cual debía ser importado al proyecto de Unity para ser procesado por varios scripts personalizados y, así, poder generar la escena con todos los elementos del nivel. Sin embargo, este proceso de traducción entre el mapa desarrollado en Tiled y el juego exportado tiene múltiples inconvenientes como, por ejemplo, que las escenas deben ser exportadas de nuevo cuando son editadas en Tiled por un cambio en el diseño del nivel, o que ciertas funcionalidades deben ser creadas a mano para cada juego concreto, ya que no son fácilmente generalizables.

Con este Trabajo de Fin de Grado se pretende resolver estos problemas y facilitar la labor de los diseñadores de niveles, mediante una herramienta integrada en el propio entorno de Unity que permita generar los elementos del nivel directamente en la escena, trasladando así las ventajas de sencillez y comodidad de editores como Tiled a la creación de juegos 2.5D en Unity.

1.2. Objetivos

El objetivo de este Trabajo de Fin de Grado es obtener una herramienta, en forma de paquete de Unity, que pueda ser importada al proyecto de un juego de scroll lateral 2D —ya sea tanto de gráficos 2D como 3D; es decir, juegos cuyo apartado visual se base en sprites o mallas 3D— y ofrezca las utilidades y comodidades indispensables de otros editores de niveles como Tiled. De este modo, la herramienta debe cumplir con los siguientes requisitos:

- Debe proporcionar una cuadrícula 2D, formada por un número de casillas especificado por el usuario, donde se puedan colocar los diferentes elementos del nivel.
- Debe ofrecer las herramientas para colocar o pintar los objetos sobre la cuadrícula y para eliminarlos.
- El usuario debe poder crear diferentes capas sobre las que pintar en la cuadrícula, de manera que pueda organizar los objetos y aplicar ciertos cambios sobre ellos.
- Se debe poder modificar la profundidad de cada una de las capas en el eje perpendicular al plano de la cuadrícula. Esta función está principalmente dirigida a los elementos que componen el arte del nivel, pues el editor está enfocado en juegos con lógica 2D cuya acción transcurra en el plano de la cuadrícula.
- El usuario debe disponer de una paleta con los diferentes objetos que pueda colocar en la cuadrícula, ordenados en categorías.
- El usuario debe poder especificar qué objetos forman parte de la paleta y agruparlos en sus categorías de una manera sencilla.

- El usuario debe poder modificar las propiedades de los objetos que se van a colocar sin tener que alterar su prefab.
- La herramienta debe poder ser utilizada sin necesidad de modificar código.
- La herramienta debe ser configurable para poderse adaptar a diferentes juegos.

1.3. Plan de trabajo

A continuación se detallan las tareas realizadas cada mes para cumplir nuestros objetivos.

Noviembre 2024:

- Investigar el diseño de otros editores de niveles

Diciembre 2024:

- Investigar la API de Unity para el *Editor Scripting*

Enero 2025:

- Comenzar con el desarrollo de la herramienta de editor de niveles
- Implementación de la lógica del grid

Febrero 2025:

- Implementación de los modos de edición de pintar y borrar

Marzo 2025:

- Serialización de objetos instanciados para asegurar su persistencia

Abril 2025:

- Agrupación de los objetos de la paleta en categorías
- Escritura en disco de la información de los objetos al entrar en modo ejecución

Mayo 2025:

- Implementar la posibilidad de crear varias capas para pintar

Junio 2025:

- Implementar la posibilidad de modificar la profundidad de las capas
- Sistema de metadatos de los objetos
- Empezar la redacción de la memoria

Julio 2025:

- Implementar el sistema para añadir objetos a la paleta
- Implementar la posibilidad de ajustar el tamaño de los tiles
- Implementar la carga de la información de los objetos ya instanciados al abrir la escena
- Redacción de la memoria

Agosto 2025:

- Corrección de *bugs*
- Recrear los niveles de *Fulvinter* y *Wizara* usando nuestro editor
- Redacción de la memoria

Capítulo 2

Estado de la Cuestión

Este capítulo lo dividiremos en dos secciones. En la primera, analizaremos la situación actual de los editores de niveles en el mundo del desarrollo de videojuegos. En la segunda, repasaremos las soluciones que se han intentado dar anteriormente al problema que ocupa a este Trabajo de Fin de Grado, analizando algún proyecto similar.

2.1. Editores de niveles similares

Comenzaremos con el estudio del editor 2D Tiled, continuaremos con las alternativas de edición de niveles que nos proporciona Unity de manera nativa, y concluiremos examinando los editores adaptados a videojuego *Super Mario Maker* y *Mega Man Maker*.

2.1.1. Tiled

A continuación procedemos a analizar Tiled, el editor de niveles que ha sido empleado durante el diseño de niveles de *Fulvinter*. Tiled es un editor de niveles 2D gratuito y de código abierto, disponible en Windows, macOS y Linux, cuya principal función es la edición de mapas de baldosas o tiles. Permite crear capas con baldosas rectangulares rectas, isométricas proyectadas, isométricas escalonadas y hexagonales escalonadas. Estos tiles se rellenan de imágenes 2D o sprites, los cuales son agrupados en conjuntos de patrones o tilesets. Estos conjuntos pueden estar formados por una colección de imágenes individuales o por una única imagen que contenga varias baldosas, las cuales son tratadas de manera individual a la hora de dibujar sobre la cuadrícula. También permite crear varias capas de baldosas para una edición más personalizada y cómoda. Dispone de herramientas para pintar sobre las baldosas, borrar el contenido de ellas, seleccionar varias baldosas, rellenar una región de un mismo contenido y seleccionar las baldosas de un mismo patrón, entre otras. Además, permite colocar imágenes libremente, es decir, no ajustadas a una baldosa concreta, para añadir información sobre el nivel del juego. «Introduction —

Tiled Manual», 2023

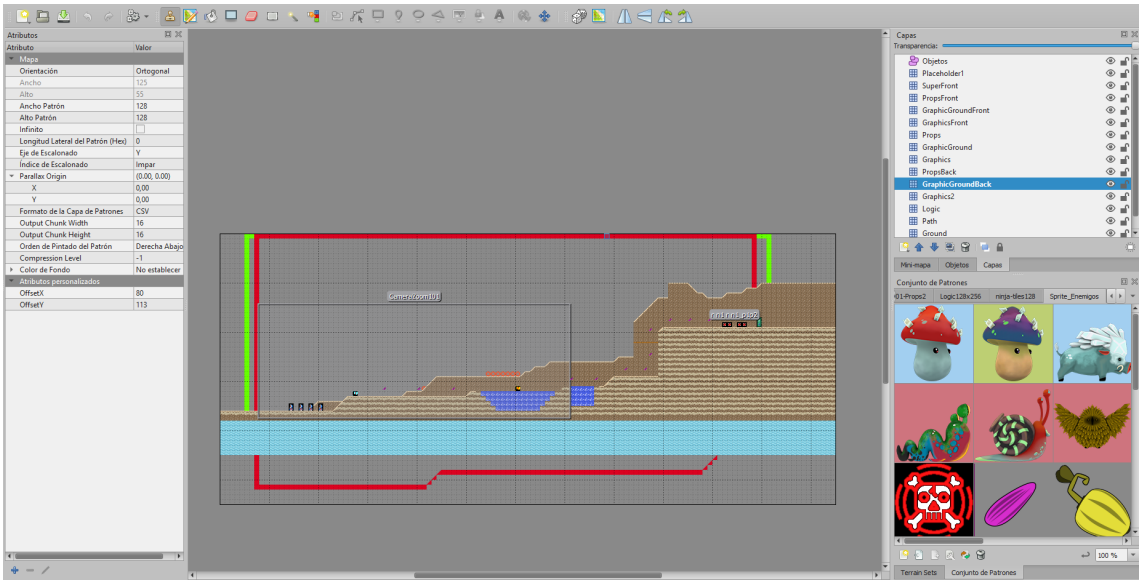

Figura 2.1: Nivel de *Fulvinter* diseñado en Tiled

Tiled ha sido empleado durante el desarrollo de *Fulvinter* para crear los diferentes niveles que componen el juego. En la figura 2.1 se muestra una captura de pantalla de un nivel de *Fulvinter* diseñado en Tiled. Este nivel es, además, el mismo que hemos recreado en Unity haciendo uso exclusivo de nuestro editor de niveles. Para el diseño de niveles en Tiled durante el desarrollo de *Fulvinter* se emplearon sprites 2D que representan a la mayor parte de elementos que componen los niveles: baldosas de terreno, elementos del escenario, enemigos, objetos interactuables, cajas o baldosas de colisión. Con ellos, se creó una primera versión 2D de los niveles, con los elementos básicos que los componen. Pero, dado que el videojuego *Fulvinter* es un juego de gráficos y arte 3D, se debió encontrar una manera para, usando como punto de partida esta versión preliminar, obtener el nivel final de gráficos 3D, aunque manteniendo la lógica 2D.

El siguiente paso es exportar cada nivel diseñado en Tiled a un archivo en formato JSON. Este archivo incluye una matriz o array por cada capa creada para ese nivel en Tiled, las cuales contienen un número que corresponde al identificador de algún sprite del tileset del nivel. El archivo también incluye información adicional sobre el nivel, como sus dimensiones, orientación, ruta de los sprites, etc.

A partir de los datos de los archivos exportados de los niveles se recrearon versiones de gráficos 3D -esto es, con mallas 3D en lugar de sprites 2D, pero manteniendo la lógica 2D del juego donde la acción transcurre en un plano- de los niveles en Unity. De este modo, haciendo uso de varios scripts, se leían los archivos en formato JSON y se instanciaba en la escena de Unity la malla 3D correspondiente al identificador de cada elemento de las matrices de capas. El resultado de esto es una escena de Unity en la que el contenido de cada baldosa del nivel 2D de Tiled es representado con un modelo 3D, tal y como se aprecia en la figura 2.2. Dicho de otro modo, hemos pasado de un nivel 2D completamente plano en Tiled a otro nivel de gráficos 3D,

pero manteniendo la lógica 2D y su diseño original.

Figura 2.2: Nivel de *Fulvinter* exportado a Unity

Este ingenioso proceso puede resultar muy cómodo en ciertos aspectos. Diseñar los niveles en Tiled aporta muchas ventajas, pues el programa aporta las herramientas necesarias para crear niveles de manera intuitiva y rápida. Además, si nuestra intención es desarrollar un videojuego 2D basado en tiles, utilizar esta herramienta puede ser una opción muy efectiva, ya que los mapas creados en Tiled pueden ser fácilmente importados a motores de videojuegos como Godot o GameMaker. Sin embargo, si queremos crear un juego 3D, parece contraintuitivo comenzar su desarrollo con una etapa en 2D para luego convertirlo a 3D. ¿Acaso no es más efectivo crear el juego en 3D desde el inicio?

Pues bien, el motivo por el cual este sistema resulta más eficiente que utilizar el propio editor de escenas 3D de Unity radica en que dicho editor no facilita, de forma nativa, el ajuste preciso de los objetos a una cuadrícula. Esta limitación es comprensible en el contexto de juegos puramente 3D, donde se prioriza la libertad del diseñador para posicionar los elementos con mayor flexibilidad. Sin embargo, en un entorno 2D —especialmente a nivel lógico— esta libertad puede ralentizar significativamente el proceso de diseño, al no contar con una alineación automática que facilite la colocación de objetos.

En un flujo de trabajo convencional, el diseñador debe seleccionar manualmente cada objeto, arrastrarlo hasta la escena, colocarlo cuidadosamente y repetir el proceso para cada instancia, lo que supone una pérdida de tiempo considerable. Por el contrario, herramientas como Tiled o como el editor desarrollado en este proyecto permiten al usuario simplemente "dibujar" los tiles deseados sobre la cuadrícula, los cuales se instancian y posicionan automáticamente en sus ubicaciones correctas, agilizando considerablemente el proceso.

Por otro lado, si se construye un nivel 3D completo —aunque Unity dispone de herramientas como ProBuilder, que permiten crear el *whitebox* del nivel—, este se diseña inicialmente mediante geometría simple (comúnmente con cajas blancas), lo cual representa la lógica y estructura base del entorno. A partir de esta base,

se itera sobre el diseño hasta alcanzar una versión suficientemente sólida, sobre la cual se aplicará el arte definitivo. Este sería el caso ideal, en el que el nivel puede probarse exhaustivamente antes de ser vestido visualmente, asegurando así que los elementos artísticos se integren sobre una estructura jugable ya establecida tanto a nivel lógico como físico. Sin embargo, este proceso rara vez ocurre de forma tan lineal. Cuando el juego se somete a pruebas externas —ya sea mediante *playtesting*, demostraciones en ferias o versiones beta/demos abiertas—, el feedback recibido suele implicar modificaciones importantes en los niveles. Estas alteraciones, aplicadas sobre el *whitebox*, afectan inevitablemente al arte ya implementado, lo cual conlleva un trabajo adicional y un consumo de tiempo considerable.

2.1.2. Tilemap de Unity

Como ya mencionamos anteriormente, Unity es un motor de videojuegos comercial y multiplataforma basado en un modelo de negocio de pago por beneficios. Utilizar Unity aporta muchas ventajas y facilidades, especialmente a los equipos de desarrollo más pequeños y con un menor presupuesto. Proporciona muchas herramientas y funcionalidades de gran utilidad para crear un videojuego, además de contar con una comunidad muy activa que amplía las funcionalidades del motor, por ejemplo, en forma de *plug-ins* o *add-ons*. Estas extensiones pueden ser comercializadas en la tienda online oficial de Unity: la *Asset Store*[1]. Entre todas estas extensiones nos encontramos con Tilemap, un editor de mapas basado en tiles. Dado su similitud con Tiled, vamos a profundizar en sus funcionalidades en este apartado.

El sistema de Tilemap de Unity constituye un conjunto de herramientas y funcionalidades orientadas al diseño y creación de niveles 2D formados por tiles. Se basa en una cuadrícula o grid sobre la que se colocan los diferentes objetos que componen el nivel, haciendo uso de herramientas de pintado específicas. Este sistema es especialmente útil a la hora de crear prototipos de niveles, ya que, al estar directamente integrado en el motor, permite a los diseñadores y artistas probarlos en un escenario y bajo unas condiciones muy similares a las que tendrá la versión final del juego. «Introduction to Tilemaps», 2025

A continuación vamos a describir las principales funciones del sistema, así como su ciclo de trabajo, basándonos en la información detallada en el capítulo 2 del libro *Unity 2022 by Example: A project-based guide to building 2D and 3D games, enhanced for AR, VR, and MR experiences* (Cameron, 2024). Tilemap debe ser instalado en el proyecto de Unity a través del *Package Manager*. Una vez importado, al hacer click derecho sobre la jerarquía de la escena se mostrará la opción de instanciar diferentes tipos de Tilemap. Así, podremos elegir entre un grid rectangular, hexagonal o isométrico, entre otros, según se adapte a las necesidades de nuestro juego. Tilemap ofrece también una paleta o *Tile Palette*, la cual incluye las herramientas de pintado, el selector de Tilemap activo para especificar sobre qué capa pintar, el repertorio de tiles u objetos para pintar disponibles o el selector de brocha, de la cual hablaremos más adelante. En la figura 2.3 podemos apreciar un ejemplo de paleta

[1] https://assetstore.unity.com/

con las herramientas de dibujado y los sprites con los que se podrá pintar en el grid.

Figura 2.3: Paleta de Tilemap («Introduction to Tilemaps», 2025)

Para poder utilizar los tiles, se ha de preparar anteriormente los sprites que vamos a utilizar. Existen dos maneras de proporcionar los sprites a Tilemap: mediante sprites individuales o utilizando una hoja de sprites o sprite sheet. Los sprite sheet son más eficientes durante la carga de recursos, pero deben ser troceados usando, por ejemplo, el editor de sprites de Unity. De esta manera, los convertimos en una imagen de tilemap, la cual está compuesta por imágenes más pequeñas ordenadas en una cuadrícula 2D. Estas imágenes, al compartir todas tamaño, constituirán los tiles u objetos que vamos a pintar en el grid para formar el nivel. Como hemos mencionado anteriormente, el sistema de Tilemap admite el dibujado ordenado en capas. Este sistema de capas se basa en crear varios tilemap, todos ellos constituidos como hijos del GameObject del grid. De esta manera se puede especificar en qué orden se dibujarán las capas que hayamos creado, de la misma manera que se determina el orden de renderizado de los sprites colocados en la escena manualmente mediante los llamados *sorting layers*.

Otra funcionalidad que merece la pena destacar son los diferentes tipos de tiles que admite Tilemap. De esta manera, encontramos el tile por defecto, el rule tile y el animated tile. El primero corresponde al tile ya descrito: simplemente lo constituye una imagen previamente ajustada al tamaño de las casillas del grid. Por otra parte, los rule tiles permiten dibujar varios tiles al mismo tiempo. De este modo, podemos

especificar con antelación el contenido de las casillas adyacentes a la casilla que vamos a pintar, tal y como se aprecia en la figura 2.4. Esto es especialmente útil a la hora de pintar, por ejemplo, las casillas que delimitan el borde del nivel, pues es común que estas casillas las rellenen sprites que suavicen este extremo y, así, no termine bruscamente. Por último, nos encontramos con los animated tiles. Como su nombre indica, estos tiles animados se basan en asignar a una casilla diferentes sprites, los cuales se irán alternando a una velocidad determinada para transmitir la sensación de animación.

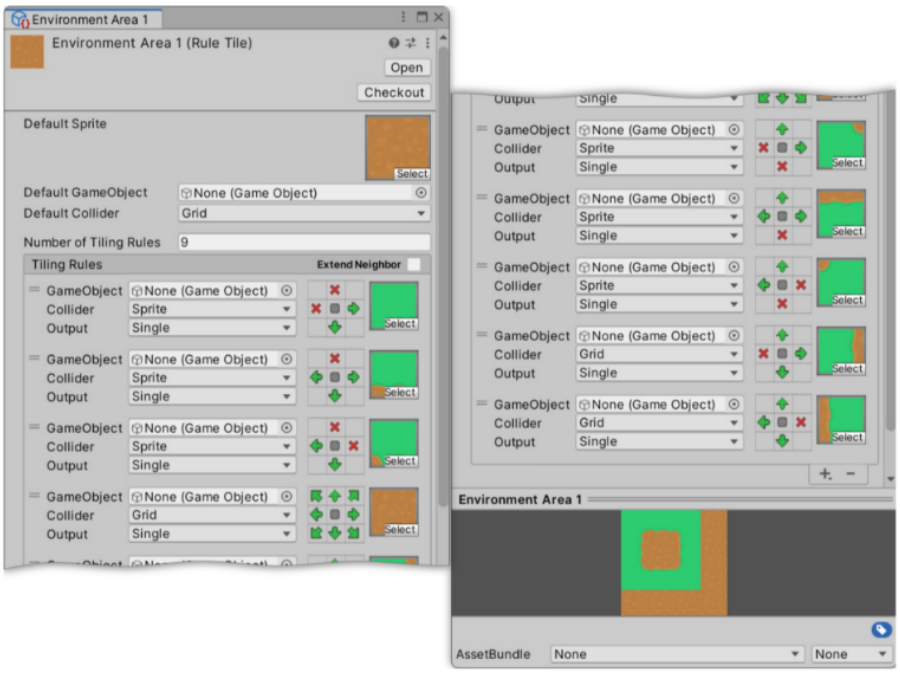

Figura 2.4: Interfaz de configuración del Rule Tile (Cameron, 2024)

Como podemos ver, Tilemap es una herramienta muy completa y nos puede ahorrar mucho tiempo a la hora de diseñar y dar forma a los niveles de un juego 2D organizado en tiles. Además, el hecho de que esté integrada en Unity supone una ventaja sobre otros editores que no lo están, como Tiled, pues no necesitamos importar los niveles creados en el editor externo al proyecto del juego en Unity, con los problemas de compatibilidades y ajustes que ello puede acarrear. Sin embargo, encontramos algunas limitaciones. Tilemap está diseñado para instanciar sprites; esto quiere decir que los objetos que añadamos a la escena con esta herramienta serán imágenes 2D con componentes muy limitados, como un transform —el componente de Unity que determina la posición, rotación y escala del objeto en el mundo de Unity— y una caja de colisión. Esto podría resultar un gran inconveniente a la hora de colocar objetos interactuables o enemigos con IA sobre el nivel, pues requerirán de scripts con comportamientos personalizados. No obstante, existe una manera de instanciar objetos con más componentes: utilizar una brocha de GameObject o GameObject Brush. Este tipo de brocha permite pintar cualquier tipo de GameObject sobre el grid haciendo uso de las herramientas de dibujado de Tilemap.(«GameObject Brush (2D Tilemap Extras Package Manual)», 2023) El principal inconveniente

es la manera en la que se usa: para cada objeto con algún componente que no admita la brocha por defecto de Tilemap se ha de crear una brocha de GameObject personalizada para el objeto en cuestión. Esto puede consumir mucho tiempo en el desarrollo de un juego de un tamaño medio o grande, pues posiblemente contengan muchos objetos o enemigos con comportamientos complejos. Crear una brocha para cada uno de ellos puede resultar muy laborioso. Además, la versión estándar de Tilemap no incluye esta brocha, sino que se debe instalar el paquete de Unity *2D Tilemap Extras* para poder usarla. Esto podría explicar el hecho de que muchos desarrolladores ni siquiera conozcan esta funcionalidad.

Siguiendo la misma línea del inconveniente anterior, encontramos dificultades para instanciar objetos 3D. Tilemap puede ser empleado para pintar tiles con objetos 3D, pero se ha de utilizar la brocha de GameObject, pues estos objetos no son simples sprites. En esta situación, sería necesario crear una brocha personalizada para todos los objetos, incluso para aquellos que conformen el terreno y no tengan más que un transform y una caja de colisión como componentes. Por esto, para desarrollar juegos de gráficos 3D y lógica 2D —un juego 2.5D—, como *Fulvinter*, el caso que nos ocupa a nosotros, utilizar Tilemap resulta muy tedioso.

2.1.3. Super Mario Maker 2

Super Mario Maker 2 es un editor de niveles de la saga de juegos *Super Mario Bros* adaptado a videojuego lanzado para la plataforma Nintendo Switch en 2019. Corresponde a la secuela de *Super Mario Maker* para Nintendo Wii U y 3DS. Este juego fue inicialmente concebido como una herramienta interna de edición de niveles para el equipo de desarrollo. Sin embargo, el equipo encargado en desarrollar la herramienta le presentó al diseñador "senior" de videojuegos Takashi Tezuka la idea de comercializarlo como un juego independiente. Tezuka, que había considerado anteriormente crear una secuela del juego de dibujo *Mario Paint*, vio en *Mario Maker* una oportunidad para crear un juego que incite a los jugadores a fomentar su creatividad de manera similar a *Mario Paint* (Lien, 2014). De esta manera surgió *Super Mario Maker*, un juego que, gracias a su editor de niveles incorporado, provee de las herramientas necesarias para que cualquier jugador pueda crear sus propios niveles, dando lugar así a una comunidad activa que comparte sus creaciones más sorprendentes e innovadoras. Dado que *Super Mario Maker 2* es, a fecha de escritura de esta memoria, el último lanzamiento de la saga y la versión más completa, en esta sección analizaremos dicha edición.

Al iniciar *Super Mario Maker 2* nos encontramos con dos modos: crear, donde está alojado el editor de niveles, y jugar, que permite jugar a niveles creados anteriormente. El proceso de creación de un nivel es muy sencillo e intuitivo. En primer lugar, encontraremos únicamente en el escenario el punto de partida del jugador y la figura de Mario. A partir de ahí podremos colocar diferentes elementos y objetos en el escenario, como el terreno del suelo, plataformas, bloques interactuables, *power-ups* o enemigos. El escenario está inmerso en una cuadrícula dividida en baldosas, sobre la cual se ajustan todos los elementos que conforman el nivel. Por último, necesitamos colocar el banderín de meta para tener un nivel jugable. En la figura

2.5 podemos apreciar la interfaz del editor de niveles. La navegación por la interfaz puede hacerse de dos formas: con el movimiento de los *joysticks* del mando, desplazando así un puntero en forma de mano, o mediante la pantalla táctil de la consola. Esta última opción solo está disponible en el modo portátil de la Nintendo Switch debido a limitaciones físicas de la consola.

Figura 2.5: Interfaz del editor de *Super Mario Maker 2* («Game UI Database – Super Mario Maker 2», 2025)

En la parte superior de la interfaz encontramos la paleta para seleccionar el objeto a pintar. En ella aparecen los últimos objetos que hemos utilizado, por lo que su disposición varía. No obstante, es posible fijar uno o varios objetos para tener un acceso rápido a los mismos. Para acceder a la lista completa de objetos debemos usar el botón con el icono de la lupa. Estos objetos están agrupados en diferentes categorías, como terreno, objetos o enemigos. Los elementos de cada categoría están ordenados en un dial para que la navegación con *joystick* resulte más cómoda y rápida, tal y como se muestra en la figura 2.6.

En la zona derecha de la interfaz encontramos las herramientas de borrador, deshacer —elimina la última acción realizada —, borrar todos los elementos del nivel, guardar o cargar escenario —si el usuario desea proseguir con la edición más adelante — y publicar nivel. Esto último permite a los jugadores compartir sus creaciones para que todos puedan probarlas.

En la parte izquierda encontramos las funcionalidades que determinan la configuración del nivel. Aquí podemos elegir la apariencia del nivel, eligiendo entre cinco juegos diferentes de la saga *Super Mario*, incluyendo juegos retro como el *Super Mario Bros* original y otros más contemporáneos como *New Super Mario Bros U*. Con esto cambiaremos los *assets* que podemos utilizar, hasta el punto en que para los juegos retro se usarán sprites mientras que para los modernos se usarán modelos 3D. Esto resulta especialmente interesante para nosotros, pues haciendo uso del mismo

Figura 2.6: Dial de selección de objetos («Game UI Database – Super Mario Maker 2», 2025)

editor podemos crear niveles para juegos estrictamente 2D, como los juegos retro de *Super Mario*, y juegos con lógica 2D pero apartado visual 3D. Nuestro editor también queremos que cumpla con esta condición.

En esta misma parte de la interfaz podemos encontrar un botón de entorno que cambiará la temática del nivel, es decir, el arte. De esta manera, el aspecto visual de los bloques que conforman el nivel cambiará, adaptándose así a la temática escogida. Esta funcionalidad es especialmente interesante, pues permite vestir los niveles de diferentes formas, siendo iguales a nivel lógico. También está localizado el botón para ajustar el movimiento de la cámara —esta puede configurarse para seguir al jugador, moverse a una cierta velocidad en scroll lateral o para seguir una trayectoria específica—, el botón de cronómetro para definir el tiempo límite del nivel, así como un botón de jugar para poder probar inmediatamente el nivel que estamos diseñando. También encontramos un botón de edición cooperativa para que dos jugadores puedan dar forma al nivel simultáneamente. Esto refuerza la idea de que *Mario Maker* es, además de un editor de niveles, un videojuego en el que la creación de niveles debe ser, ante todo, una tarea entretenida y divertida, incluso pensada para ser disfrutada en compañía de amigos.

Por último, en la parte inferior de la interfaz se encuentran los controles para configurar si el nivel avanza horizontal o verticalmente, así como el botón para cambiar entre la estancia principal o secundarias del nivel —en los juegos de *Super Mario* es común que Mario se introduzca en tuberías, las cuales lo llevan a otras zonas ocultas del nivel—. Aquí también encontramos dos iconos que representan el punto de inicio del nivel y la meta, separados por una línea recta. Desplazando estos iconos, de manera que aumente o disminuya la distancia que los separa, podemos modificar el tamaño del nivel. («Play Guide — Super Mario Maker 2 Wiki», 2019)

Como comentamos anteriormente, la navegación por la interfaz y el escenario se realiza haciendo uso de los *joysticks* del mando o mediante la pantalla táctil de la consola. Con ello, se puede realizar una selección de varios elementos ya colocados en el nivel para poder tratarlos como uno solo. De esta manera podemos cambiar su posición en la cuadrícula, duplicarlos o eliminarlos. Todas estas acciones también pueden ser realizadas sobre elementos individuales. Cabe destacar una interacción que permite este editor y que es específica de los juegos de *Super Mario*. Al mantener seleccionado un objeto colocado, un menú es desplegado con distintas opciones, como por ejemplo aplicar un *power-up* al enemigo o a Mario, de manera que disponga de esta ventaja por defecto. También es posible hacer interactuar elementos entre sí para, por ejemplo, introducir *power-ups* o enemigos en cajas sorpresa para que aparezcan cuando Mario las golpee. Estas funcionalidades son un ejemplo de cómo un editor puede ser adaptado a las necesidades específicas de cada juego.

Como conclusión, *Super Mario Maker 2* es un editor de niveles altamente intuitivo que pone la creación de niveles personalizados de los juegos de *Super Mario* al alcance de todos. Esto es debido a la claridad y concisión de su interfaz, además de contar con tutoriales explicativos incluidos en el propio juego que explican cómo utilizar el editor, cubriendo tanto las funciones básicas como las más avanzadas. Es por ello que se ha desarrollado una gran comunidad donde los jugadores comparten sus niveles personalizados para que cualquier persona pueda jugarlos. El acierto en esta fórmula se ve reflejado en su éxito comercial, convirtiéndose en uno de los juegos más vendidos de Nintendo Switch («List of best-selling Nintendo Switch video games», 2025) y recibiendo críticas generalmente muy positivas («Super Mario Maker 2 – Critic Reviews», 2019).

2.1.4. Mega Man Maker

Mega Man Maker es, al igual que *Super Mario Maker*, un editor de niveles 2D basado en tiles de los juegos de la franquicia de *Mega Man* adaptado a videojuego. Es un editor creado por los fans de la franquicia, por lo que no es oficial, y está disponible exclusivamente en Windows. Es, en muchos aspectos, muy similar a *Super Mario Maker* (el cual tomaron como inspiración, de ahí su nombre), por lo que lo analizaremos con menor detenimiento. La información y las capturas de pantalla que se mostrarán a continuación se han obtenido en base a probar el editor nosotros mismos. *Mega Man Maker* puede ser descargado desde su página web oficial[2].

Al iniciar el juego se nos muestra un menú donde podemos seleccionar si queremos crear o editar un nivel o, por el contrario, si deseamos jugar algún nivel creado anteriormente, ya sea nuestro o de otros usuarios. Si es la primera vez que entramos al modo editor, se nos dará la opción de seguir un tutorial sobre cómo usar el editor de niveles. En la figura 2.7 podemos apreciar el uso de textos explicativos y de flechas para conducir la acción del jugador.

En la parte superior de la interfaz encontramos los diferentes objetos que podemos colocar agrupados en categorías. Cada recuadro de la parte superior corresponde

[2]https://megamanmaker.com/

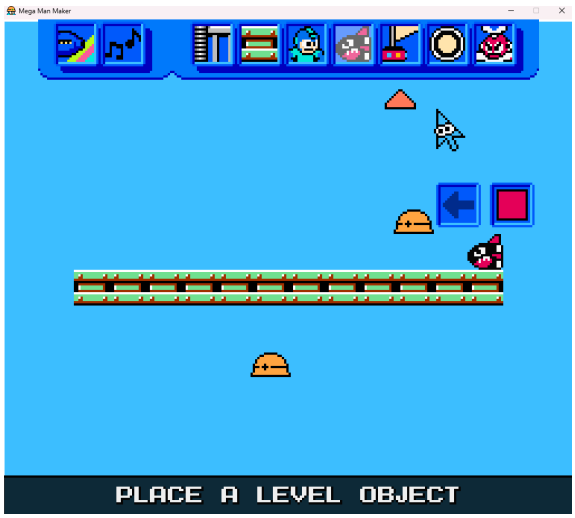

Figura 2.7: Captura de pantalla del tutorial de uso de *Mega Man Maker*.

a un botón para cada categoría. Si hacemos *click* una vez sobre él, seleccionaremos el objeto que aparece en el botón. Este objeto es el último que se ha seleccionado de esa categoría. Si volvemos a hacer *click* sobre el mismo botón se abrirá un menú con la lista completa de los objetos pertenecientes a dicha categoría (figura 2.8).

Figura 2.8: Menú de selección de objetos para una categoría concreta.

En la parte inferior derecha de la pantalla encontramos el botón que activará un desplegable con botones para guardar el nivel, cargar un nivel, ajustar los parámetros del nivel o volver al menú principal, entre otros. En la esquina inferior izquierda encontramos el botón de *Play* para poder probar el nivel que hemos construido hasta el momento. En la figura 2.9 se muestra la interfaz completa del editor.

Figura 2.9: Interfaz completa de *Mega Man Maker*.

Una vez hayamos seleccionado un objeto, podremos instanciarlo haciendo *click* izquierdo sobre el escenario. Aunque por defecto está desactivado, podemos mostrar una cuadrícula con los tiles del nivel para guiarnos mejor, en caso de necesitarlo. Sin embargo, consideramos que no es imprescindible ya que, al ocupar cada objeto un tile, resulta sencillo organizar el nivel.

El escenario está formado por distintas zonas o secciones, cada una de ellas formada por un número determinado de tiles. Estas instancias pueden ser unidas para aumentar el espacio del que dispondrá el nivel. Dicha acción se realiza con un botón que aparece en la frontera entre dos secciones, tal y como se aprecia en la figura 2.10.

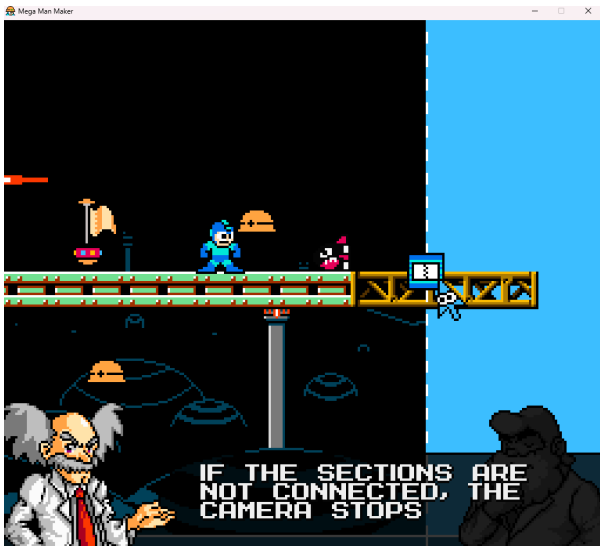

Figura 2.10: Frontera entre dos secciones, con el correspondiente botón que las une.

2.1.5. Conclusiones

Como podemos ver, cada editor tiene sus características, lo cual conlleva ciertas ventajas e inconvenientes:

- **Tiled** es una herramienta muy eficaz para el diseño de niveles 2D basados en tiles, especialmente útil en proyectos que requieren organización por capas y estructuras lógicas complejas. Su mayor fortaleza radica en su simplicidad, personalización y exportación a formatos como JSON, lo que permite su integración en motores como Unity. En el caso de Fulvinter, permitió construir niveles 2D con rapidez y luego transformarlos en niveles 3D, manteniendo la lógica original. No obstante, al no estar integrado directamente en el motor de desarrollo, requiere un paso adicional de conversión o scripting, lo que puede añadir complejidad en ciertos flujos de trabajo.

- **Unity Tilemap** es una herramienta potente y flexible para proyectos 2D dentro de Unity, ideal para desarrolladores con conocimientos técnicos. Su integración nativa evita problemas de compatibilidad, pero sufre limitaciones importantes al intentar integrar lógica compleja, objetos interactivos o gráficos 3D. La necesidad de configurar brochas personalizadas para cada GameObject hace que no sea óptimo para proyectos de gran escala sin automatización adicional. Aun así, para juegos puramente 2D o con lógica simple, resulta muy eficiente.

- **Super Mario Maker 2** destaca por su enfoque en la experiencia de usuario y la creatividad, proporcionando una herramienta accesible y divertida. Está claramente diseñado para jugadores más que para desarrolladores, con una interfaz amigable, soporte para múltiples estilos visuales y edición cooperativa. También permite probar niveles de forma inmediata e integrar mecánicas propias de los juegos de Mario, lo que lo convierte en una excelente herramienta pedagógica. Sin embargo, está limitado a la consola Nintendo Switch y su uso está enfocado únicamente en el ecosistema Mario.

- **Mega Man Maker** es una alternativa no oficial y gratuita con una interfaz similar a Mario Maker, dirigida a los fans de la franquicia Mega Man. Aunque es funcional y fácil de usar, sufre por la falta de soporte oficial y una comunidad más pequeña. Es útil como inspiración o entretenimiento, pero limitado en comparación con herramientas más completas o profesionales. Además, su uso está restringido a PC y no permite exportar los niveles fuera del entorno del propio juego, lo que lo aleja de aplicaciones prácticas en desarrollos comerciales.

En la siguiente tabla podemos ver una comparativa entre las características de cada editor:

Editor	**Tiled**	**Unity Tilemap**	**Super Mario Maker 2**	**Mega Man Maker**
Plataforma	Windows, macOS, Linux	Unity (Windows, macOS)	Nintendo Switch	Windows
Gratuito	Sí (open source)	Sí (requiere Unity)	No	Sí
Soporte 3D	No (solo 2D)	Parcial (con GameObject Brush)	Sí (visual 3D, lógica 2D)	No
Sistema de capas	Sí	Sí	No explícito	No explícito
Tipos de tiles	Estáticos	Estáticos, Rule Tiles, Animated Tiles	Estáticos, con interacciones predefinidas	Estáticos, con comportamientos predefinidos
Interfaz de usuario	Tradicional (estilo IDE)	Integrado en Unity, técnica	Muy intuitiva y accesible	Inspirado en Mario Maker
Exportación / Importación	JSON, XML	Nativo en Unity	No permite exportar	No permite exportar
Objetos interactivos	Limitado (requiere scripting externo)	Limitado (GameObject Brush)	Amplio soporte predefinido	Amplio soporte predefinido
Uso en proyectos reales	Sí (ej. Fulvinter)	Sí (integrado en desarrollo)	No (uso recreativo)	No (uso recreativo)

Tabla 2.1: Comparativa de editores de niveles similares

2.2. Trabajos similares

En esta sección analizaremos tres editores de niveles provenientes de otros proyectos universitarios. Comenzaremos con un editor para juegos exclusivamente 2D, continuaremos con otro orientado a los videojuegos 3D de perspectiva isométrica y concluiremos con un editor para juegos 3D basados en bloques.

2.2.1. TEd2D

TEd2D es un editor de niveles para juegos 2D de Unity basados en tiles, producto de un proyecto universitario. Sus características, funcionamiento y uso se detallan en el documento del proyecto *TEd2D: Um editor para criação de cenários de jogos 2D com Unity* (da Silva Beserra, 2015).

TEd2D consta de los elementos fundamentales de un editor de niveles basado en tiles. Utiliza una cuadrícula o grid sobre la que se ubicarán los objetos, ajustándolos a un tile determinado.

TEd2D hace uso de la librería *UnityEditor* para crear interfaces personalizadas del editor de Unity. Así, modifica la vista del inspector del *GameObject* de la escena que incluye el grid, permitiendo funcionalidades como redimensionar la cuadrícula, cambiar su color, ocultarla, etc.

Dispone de una clase *TileSet*, la cual contiene un vector de objetos de tamaño

personalizable donde se pueden añadir *Prefabs* para su posterior uso con el editor. Esta clase está diseñada para actuar a modo de categoría, agrupando múltiples objetos con alguna propiedad en común, de manera que posteriormente se muestren en la interfaz ordenadamente.

En el inspector personalizado del grid se muestra un botón para elegir el *TileSet* activo. Esto mostrará en la parte inferior del inspector el selector de *Prefabs*. El selector de *Prefabs* consiste en una serie de botones que muestran el *sprite* del *Prefab* que representan. Estos *Prefabs* son los que se han incluido en el vector de objetos del *TileSet*. Por último, se muestra en la interfaz el objeto seleccionado actualmente. En la figura 2.11 se muestra el inspector personalizado que acabamos de describir.

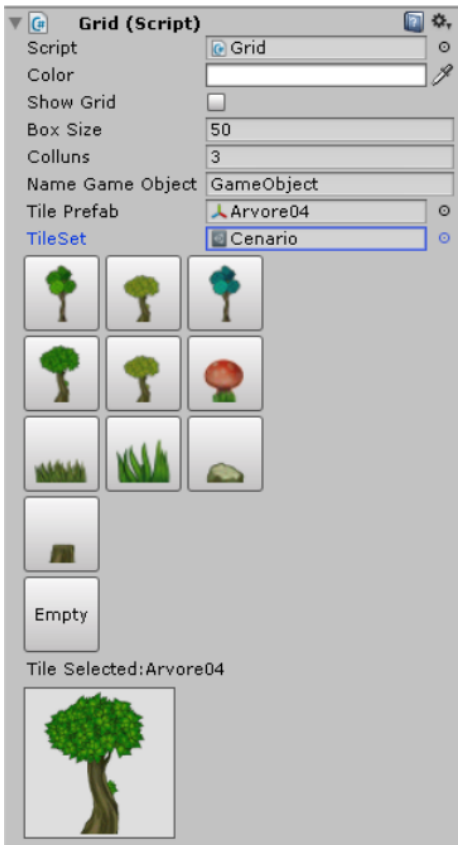

Figura 2.11: Inspector personalizado de TEd2D (da Silva Beserra, 2015)

El editor TEd2D se distribuye en forma de paquete de Unity, de manera que su importación al proyecto de un juego en desarrollo es muy sencilla. La manera en la que se crean las diferentes categorías de objetos, agrupados en *TileSets*, y se añaden los objetos resultan muy inmediatas e intuitivas. Además, la interfaz personalizada, aunque es bastante simple, incluye los elementos y funcionalidades indispensables para la edición de niveles 2D por tiles, por lo que resulta clara y concisa.

Sin embargo, el proyecto tiene algunas limitaciones. El editor no proporciona soporte para juegos 3D, tan solo admite *sprites* como recurso artístico. Esto choca claramente con nuestros intereses, pues nuestro editor ha de ser compatible con modelos 3D para poder crear niveles para juegos 2.5D. En la sección final de su documento, proponen como posible ampliación el soporte para juegos 3D.

También hemos encontrado algunas limitaciones menores respecto a la manera en la que se usa. Por ejemplo, para pintar objetos —es decir, para instanciarlos en la escena — se hace uso del *click* izquierdo del ratón, mientras que el derecho sirve para eliminarlos. Creemos que utilizar herramientas separadas para pintar y borrar podría ser una mejor opción, pues quedarían así distinguidas dos funcionalidades completamente opuestas. Esto evitaría borrar accidentalmente objetos mientras se dibuja.

2.2.2. IsoUnity

IsoUnity es un conjunto de herramientas para Unity orientadas a la creación de videojuegos 3D de perspectiva isométrica. Es el resultado de un Trabajo de Fin de Grado de la Facultad de Informática de la UCM con el nombre *Un conjunto de herramientas para Unity orientado al desarrollo de videojuegos de acción-aventura y estilo retro con gráficos isométricos 3D* (Pérez Colado & Pérez Colado, 2014). IsoUnity está enfocado en el desarrollo de juegos del género aventura gráfica, con gráficos retro y perspectiva isométrica. Un ejemplo de videojuego que encaja con esta descripción es *La abadía del crimen* («La abadía del crimen», 1987), el cual les sirvió a los desarrolladores de IsoUnity como punto de partida. En la figura 2.12 se muestra una captura de uno de los *remakes* de *La abadía del crimen*. En ella se puede apreciar la perspectiva isométrica de este tipo de juegos.

Figura 2.12: Captura del videojuego *The Abbey of Crime Extensum*, *remake* de *La abadía del crimen* («The Abbey of Crime Extensum», 2016)

IsoUnity contiene múltiples herramientas para facilitar el desarrollo de estos juegos, como interfaces para diálogos o gestión de objetos en inventarios, pero nosotros nos ocuparemos de analizar su editor de mapas, pues es la funcionalidad que más se aproxima a nuestro proyecto.

La perspectiva isométrica de este tipo de juegos implica, inevitablemente, que la manera en la que se crean los niveles sea distinta a los juegos 2D y 2.5D de scroll lateral y basados en bloques. Sin embargo, creemos que estudiar IsoUnity puede resultarnos de gran ayuda, pues podremos averiguar cómo se enfrentan a los retos que surgen a la hora de crear niveles con elementos 3D.

Los niveles construidos con IsoUnity se asientan sobre una cuadrícula de casillas. Cada casilla albergará, como máximo, una celda. La celda constituye la unidad básica con la que se construyen los niveles. Una celda es, en esencia, una malla 3D de forma cuadrangular, cuya base es de tamaño fijo y viene determinada por las dimensiones de las casillas. Sin embargo, la altura de las celdas puede ser modificada. Además, su acabado superior puede ser personalizado, variando en cierta medida el ángulo que forma la cara superior respecto a la base. Esto permite una gran variedad de celdas posibles (figura 2.13), dando así al diseñador la libertad necesaria para poder crear la base de las estructuras que conforman el nivel.

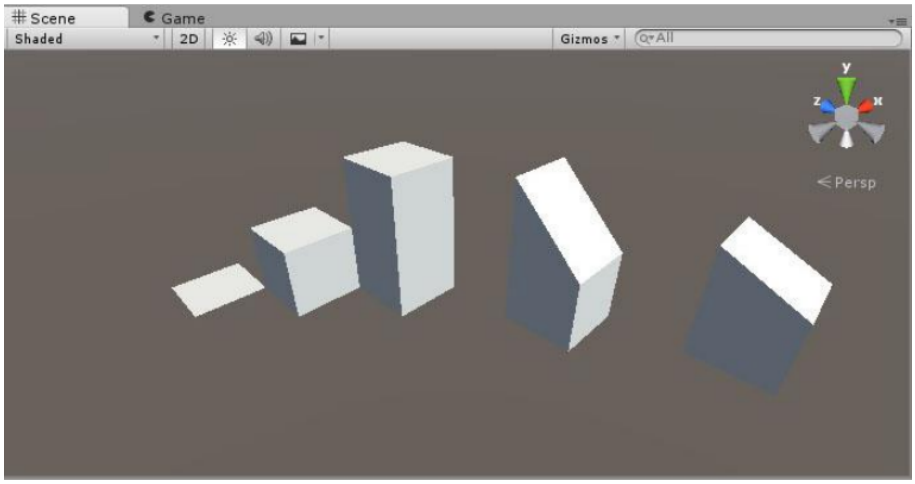

Figura 2.13: Celdas de diferentes alturas y acabados superiores creadas con IsoUnity (Santamaría Barcina & Alexiades Estarriol, 2015)

La cuadrícula de casillas está contendida en un GameObject que se incluye por defecto en la escena. Al hacer *click* sobre él, aparecen una serie de botones sobre su inspector personalizado (figura 2.14).

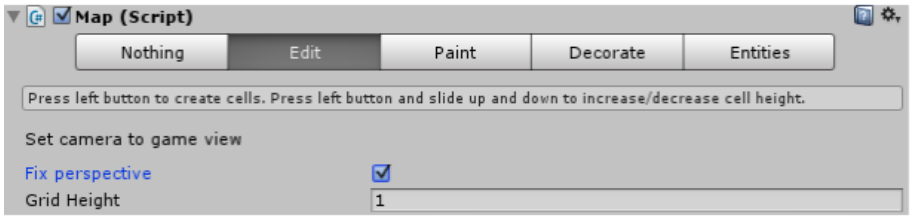

Figura 2.14: Herramientas de IsoUnity, mostradas en el inspector personalizado (Pérez Colado & Pérez Colado, 2014)

El botón *Edit* será el que nos permita colocar las celdas sobre las casillas. Tras seleccionar esta herramienta, al hacer *click* sobre una casilla se instanciará una

celda sobre ella. Sobre cada casilla puede haber una única celda, de manera que si pintamos donde ya había una celda, esta se borrará para dejar espacio para la nueva. Las celdas instanciadas serán de la altura especificada en el inspector personalizado.

Para ayudar al diseñador a saber cómo será la celda que instanciará, IsoUnity emplea la denominada celda fantasma (figura 2.15). Esta es una celda semitransparente que simula la celda que se creará al hacer *click* sobre la casilla en la que se encuentra el ratón. Esta funcionalidad es especialmente útil cuando estamos colocando celdas de diferentes alturas.

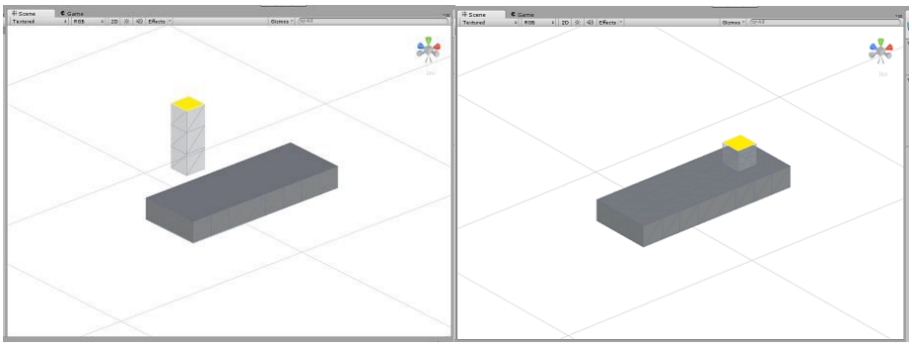

Figura 2.15: Celda fantasma de IsoUnity (Pérez Colado & Pérez Colado, 2014)

De este modo, podemos construir la base del nivel a modo de *white-box*. Cada celda puede ser marcada como transitable o no transitable por el jugador. De esta manera, podemos definir los límites del nivel, además de las celdas que constituirán obstáculos u objetos sobre los que el jugador no podrá navegar.

Con el botón *Paint* podemos hacer uso de la herramienta de pintado (figura 2.16), la cual sirve para dotar a las celdas de texturas. Para ello, IsoUnity utiliza un tipo de texturas propio llamado *IsoTextures*. Este tipo de texturas se adaptan a las diferentes caras de las celdas. Recordemos que, por la perspectiva isométrica de este tipo de juegos, tan solo son visibles tres caras, a lo sumo, de cada celda.

La última herramienta que nos encontramos para dar forma al nivel es la del botón *Decorate*. Esta sirve para añadir elementos decorativos, como elementos del mobiliario. Su funcionamiento es bastante similar a la herramienta de pintar, por lo que no entraremos en mucho detalle sobre ella. Entre sus funcionalidades podemos destacar la posibilidad de elegir en qué posición de entre las tres disponibles se colocará: sobre el suelo, pegada a la cara izquierda de la celda o a la cara derecha.

Para saber sobre la casilla sobre la que se encuentra el ratón, las tres herramientas descritas anteriormente hacen uso del sistema de físicas de Unity. Siendo más específicos, se proyecta un rayo desde la ubicación del ratón. De esta manera, se detecta sobre qué elemento ha colisionado. Con esto es posible saber la casilla sobre la que se encuentra el ratón para poder instanciar una celda o conocer la cara sobre la que vamos a aplicar una textura. Este sistema nos puede ser de gran utilidad en nuestro proyecto, pues ambos comparten la necesidad de proyectar objetos 3D sobre una cuadrícula bidimensional. En nuestro caso particular, podemos aplicar esta técnica para saber el tile sobre el que se encuentra el ratón, de manera que los objetos se coloquen en la casilla correcta.

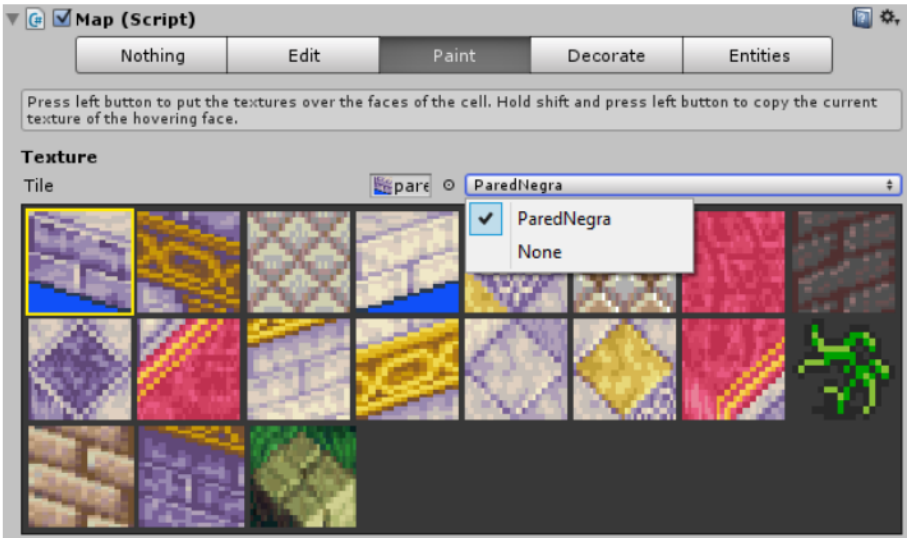

Figura 2.16: Interfaz de la herramienta de pintar de IsoUnity (Pérez Colado & Pérez Colado, 2014)

2.2.3. Plug-in de Unity para juegos basados en tiles

A continuación describiremos los aspectos más relevantes de un editor de niveles para juegos basados en tiles, producto de un proyecto universitario. La información sobre esta herramienta ha sido extraída del artículo *Tile-based game plugin for unity engine* (Nasution et al., 2020). En él, se detalla el diseño e implementación de un editor de niveles, en forma de *plug-in* para Unity, para juegos basados en tiles.

La herramienta es exclusivamente compatible con niveles de juegos 3D, por lo que la unidad básica para la creación de niveles son los bloques cúbicos. Es por ello que la manera correcta de describir este editor es como editor para juegos basados en vóxeles —la unidad cúbica básica que compone un objeto tridimensional—. No obstante, y dado que el propio autor del artículo lo describe así, de ahora en adelante usaremos el término de tile para nombrar también a los vóxeles. Además, tal y como se describirá más adelante, los bloques cúbicos se ubican sobre una cuadrícula formada por baldosas o tiles, por lo que no consideramos inapropiada esta nomenclatura.

Para poder crear niveles, el editor emplea un GameObject llamado *MyLevel*, el cual está presente en la escena. Al hacer *click* sobre él, descubriremos la interfaz de creación del editor, contenida en un inspector personalizado. Para ello, hace uso de la API *UnityEditor*. La interfaz la podemos apreciar en la figura 2.17.

Esta interfaz es muy sencilla, pero a la vez concisa. Incluye dos vectores donde se almacenarán los *prefabs* de los bloques y de los elementos decorativos. También podemos especificar el *Cursor Type* a través de un menú desplegable. Esto sirve para elegir la herramienta de dibujado. Podemos escoger entre añadir tile, eliminar tile, añadir decoración y eliminar decoración (figura 2.18).

Para elegir el objeto que queremos instanciar, debemos especificar en la variable *Index To Spawn* el índice que ocupa el objeto deseado en el vector mencionado

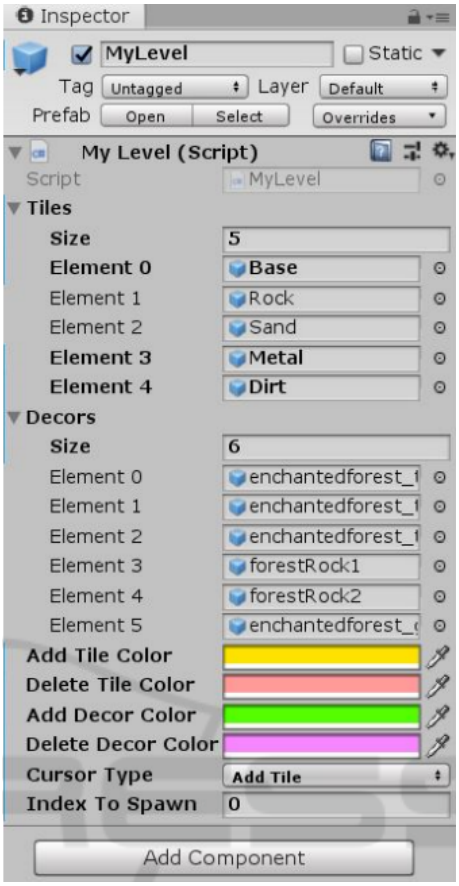

Figura 2.17: Interfaz de creación del editor (Nasution et al., 2020)

anteriormente.

Una vez hayamos añadido los *prefabs* al vector, seleccionado la herramienta de dibujado y especificado el índice del objeto, podremos empezar a crear el nivel. Para ello, dispondremos de una cuadrícula en la parte inferior de la escena, a modo de suelo. Al tratarse de un editor para juegos 3D, el usuario tendrá libertad total para mover la cámara según sus necesidades.

Sobre la cuadrícula se ubicarán los bloques que conformarán el nivel. Para que esta tarea sea más sencilla, el editor proporciona un cursor 3D. Esto es, básicamente, un objeto presente en la escena consistente en las aristas de un cubo del tamaño de un tile, tal y como se aprecia en la figura 2.19. Este cursor 3D se ubica en la posición del ratón en la escena, y representa la ubicación donde se colocará el bloque o, por el contrario, la unidad de bloque que el usuario tiene seleccionada. El cursor se desplaza por la escena a medida que el usuario mueve el ratón.

Haciendo uso de estas herramientas, el usuario podrá dar forma a la estructura del nivel, colocando los bloques que lo conforman y las decoraciones contenidas en él (figura 2.20).

A vista de esta información, podemos concluir que el editor es bastante sencillo, pero está dotado de las herramientas y funcionalidades imprescindibles para la creación de niveles 3D basados en tiles o bloques.

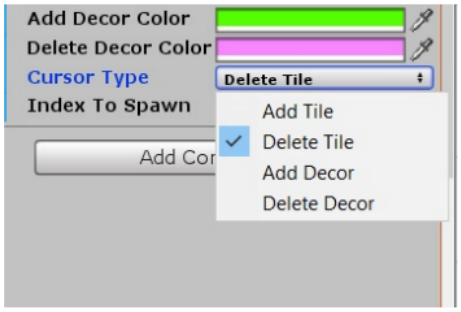

Figura 2.18: Interfaz de selección de herramienta (Nasution et al., 2020)

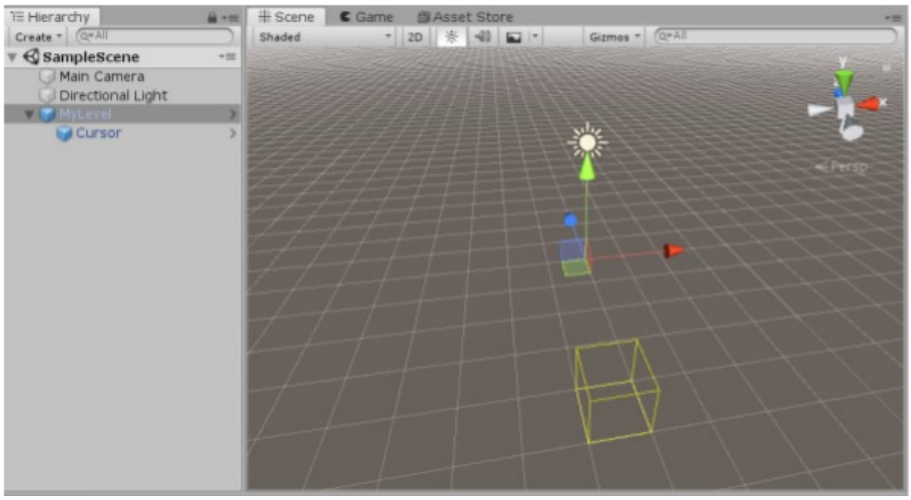

Figura 2.19: Cursor 3D visto en la escena (Nasution et al., 2020)

Entre los puntos a destacar encontramos la figura del cursor 3D. Consideramos que esta es, posiblemente, una de las mejores maneras de informar al usuario sobre la casilla con la que está interactuando. Al tratarse de un entorno 3D, la posición del ratón puede resultar insuficiente a la hora de ubicarse, pues se encuentra siempre en el mismo plano y no tiene en cuenta la profundidad. Sin embargo, con el cursor 3D podemos apreciar claramente la profundidad del cursor, distinguiendo claramente el tile sobre el que realizaríamos las diferentes acciones.

Por otra parte, encontramos algunos puntos débiles en el editor. Como ya hemos comentado anteriormente, el editor es bastante simple, lo que puede desencadenar que los usuarios echen en falta algunas funcionalidades extra a la hora de crear niveles más complejos. Sin embargo, los desarrolladores dejan la puerta abierta a futuras ampliaciones y mejoras.

También hemos notado que los objetos instanciados son añadidos a la escena sin seguir ningún tipo de orden o jerarquía. Creemos que hubiera sido una mejor opción ordenarlos de alguna manera, por ejemplo haciéndolos hijos de algún objeto. Otra funcionalidad que echamos en falta, y va de la mano del apunte anterior, es la ausencia de capas para ordenar los objetos que instanciamos.

En la interfaz también encontramos algunas carencias. La manera en la que se especifican los objetos que se usarán (añadiéndolos a un vector), se selecciona el

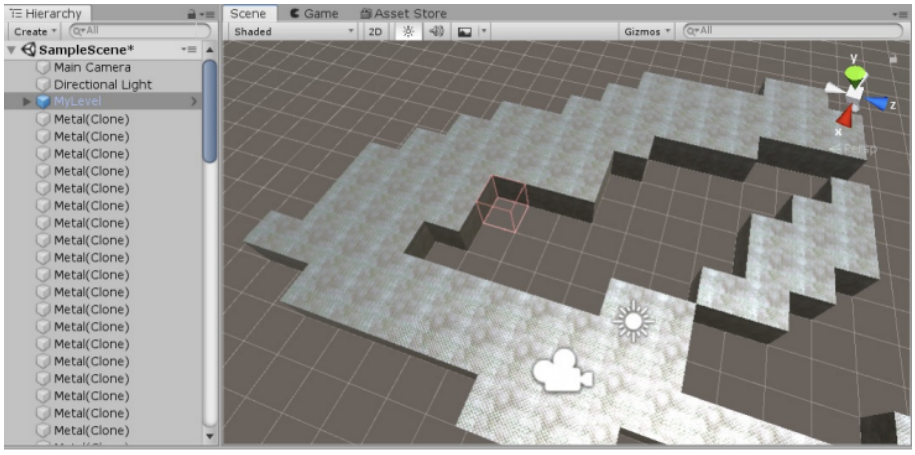

Figura 2.20: Prototipo de un nivel desarrollado con el editor (Nasution et al., 2020)

objeto en cuestión (a través de un índice contenido en una variable) y se escoge la herramienta de dibujado (mediante un menú desplegable) nos parece algo rudimentaria y poco amigable con los usuarios. Creemos que el uso de una paleta como en otros editores y añadir botones para las herramientas de dibujado hubiera sido una mejor opción.

Como conclusión, el editor ofrece las funcionalidades básicas y necesarias para dar forma a niveles basados en bloques 3D, pero tiene un gran margen de mejora en cuanto a interfaz y, tal vez, funcionalidades extra.

2.2.4. Conclusiones

Estos proyectos universitarios ofrecen algunas funcionalidades muy parecidas a nuestro editor, pero también presentan bastantes diferencias entre sí.

- **TEd2D** es una herramienta funcional y bien integrada para la creación de niveles 2D en Unity. Su diseño basado en tiles y *prefabs* lo hace adecuado para proyectos sencillos, con una interfaz clara y personalizable a través del inspector de Unity. La estructura de TileSets permite organizar objetos por categorías, facilitando la edición. Sin embargo, su uso se ve limitado a juegos puramente 2D, ya que no admite recursos tridimensionales, lo que lo aleja de proyectos que requieran compatibilidad con entornos 2.5D o 3D. Además, el sistema de interacción con *clicks* para pintar y borrar puede inducir a errores, al no diferenciar claramente entre funciones opuestas. Pese a sus limitaciones, es un editor útil y fácil de importar a cualquier proyecto en desarrollo.

- **IsoUnity** destaca por ofrecer un editor de mapas específicamente diseñado para juegos 3D con perspectiva isométrica. Su sistema de celdas tridimensionales con alturas variables permite construir niveles complejos de forma modular. La presencia de herramientas como la celda fantasma, el sistema de pintado y la decoración facilita el trabajo del diseñador al proporcionar una representación visual clara antes de cada acción. Aunque está más orientado a la creación de

aventuras gráficas con gráficos retro, sus conceptos técnicos, como el uso de *raycasts* para posicionamiento preciso en cuadrículas 3D, resultan altamente aplicables a otros proyectos. Su enfoque estructurado y su integración con la escena de Unity lo convierten en una solución completa y bien pensada, aunque especializada.

- El **plug-in para Unity** propuesto por Nasution et al. ofrece una solución básica pero efectiva para la creación de niveles 3D basados en bloques, similar a la construcción por vóxeles. Su simplicidad es su mayor fortaleza y también su principal debilidad: permite pintar, borrar y decorar con herramientas esenciales, y destaca especialmente por el uso de un cursor 3D que mejora la precisión en entornos tridimensionales. Sin embargo, su interfaz resulta rudimentaria, al requerir seleccionar objetos mediante índices numéricos y sin jerarquía de organización en la escena. La falta de capas, paletas visuales o agrupación de elementos limita su escalabilidad, aunque se presenta como una base sólida para desarrollos futuros. Su valor reside en la claridad conceptual y la posibilidad de extender sus funcionalidades.

Editor	**TEd2D**	**IsoUnity**	**Plug-in Unity**
Tipo de juego	2D basado en tiles	3D isométrico (aventura gráfica)	3D basado en bloques (vóxeles)
Plataforma	Unity (Windows/macOS)	Unity (Windows/macOS)	Unity (Windows/macOS)
Tipo de elementos	Sprites y prefabs 2D	Mallas 3D con alturas personalizadas	Bloques cúbicos 3D (vóxeles)
Interfaz	Inspector personalizado con selección de TileSet y Prefabs	Inspector con botones de herramientas, celda fantasma y texturizado	Inspector básico con vectores de prefabs y menú de herramientas
Herramientas principales	Pintar/borrar con clics, selección de TileSet	Edición, pintado, decoración, celda fantasma 3D	Añadir/eliminar bloques y decoraciones con cursor 3D
Sistema de capas	No	Parcial (estructura jerárquica de celdas)	No (instancia sin jerarquía ni capas)
Facilidad de uso	Intuitivo y directo, aunque con detalles mejorables	Completo y visualmente informativo (celda fantasma)	Funcional pero con interfaz poco amigable
Limitaciones	Solo compatible con 2D, no soporta modelos 3D	Enfocado en perspectiva isométrica, no apto para scroll lateral	Interfaz rudimentaria, sin jerarquía ni capas, funcionalidades básicas

Tabla 2.2: Comparativa de proyectos similares

Capítulo 3

Descripción del Trabajo

En este capítulo realizaremos una descripción detallada de las características necesarias para el correcto funcionamiento de la herramienta, la implementación de los diversos sistemas que cumplen con estos requisitos, y los problemas que nos hemos encontrado y cómo los hemos resuelto. Para ello, hemos dividido el capítulo en dos secciones. La primera habla sobre las decisiones de diseño que se han tomado antes de comenzar a desarrollar el editor de niveles, especificando así sus funcionalidades y herramientas. La segunda describe la implementación técnica que ha permitido cumplir con las decisiones de diseño.

3.1. Diseño

A continuación procedemos a explicar cómo queremos que se comporte nuestra herramienta y de qué elementos y funcionalidades disponga.

3.1.1. Cuadrícula o grid

En la sección 2.1.1 justificamos la necesidad de disponer de una cuadrícula o grid donde sean ajustados los objetos que componen un nivel 2D. Por ello, al tratarse de un editor de niveles de lógica 2D, consideramos indispensable que nuestra herramienta organice los elementos de dicha manera.

Nuestro editor dispone de una colección de cuadrículas para organizar los objetos instanciados. Cada cuadrícula recibe el nombre de capa, mientras que cada capa está formada por un número variable de baldosas o tiles. Sobre estas baldosas se colocarán los objetos que conformen el nivel, ajustándose a su posición. Además, la baldosa de cada capa puede contener exclusivamente un único objeto. De este modo, si el usuario desea colocar dos objetos en la misma posición deberá usar diferentes capas. En el caso en el que las dimensiones de algún objeto sean mayores a las de las baldosas, el punto del origen del objeto será el que ocupe el centro de la baldosa, pudiendo sobresalir parte de él por los costados.

Estas baldosas forman una disposición rectangular con un número determinado de filas y columnas. El usuario debe poder ajustar el número de baldosas de las filas y las columnas para poder así configurar el tamaño del nivel que se dispone a crear. También debe poder ajustar el tamaño de las baldosas para adaptar la cuadrícula a las diferentes medidas de los objetos que quiera instanciar.

La profundidad de cada capa también debe poder ser modificada. Esta profundidad afecta a la posición de sus objetos en el eje perpendicular al plano del nivel, es decir, a la cuadrícula. Esta característica se diseñó principalmente para permitir una mayor expresión artística, de manera que los artistas puedan colocar elementos decorativos del escenario a profundidades diferentes al plano donde transcurre la acción. Un ejemplo de esto podría ser el colocar los objetos de arte que componen el fondo del escenario. Sin embargo, esta funcionalidad también puede ser utilizada para crear distintas mecánicas o fases del nivel, como un obstáculo que se mueva perpendicularmente a la cuadrícula y cuyo efecto se limite al momento en el que la atraviesa.

El usuario debe poder crear y eliminar capas como desee. Además, solo una capa debe poder ser seleccionada al mismo tiempo. De este modo, cuando el usuario utilice las herramientas de pintar o borrar, sus acciones solo se verán reflejadas sobre los objetos que contenga la capa seleccionada.

Para el correcto funcionamiento de la herramienta, es muy importante asegurar que el estado del editor coincida con el estado de la escena de Unity. Dicho de otro modo, el editor debe saber qué objetos se han instanciado en la escena y qué tiles ocupan. Por ello, hemos implementado varios mecanismos de persistencia de los que hablaremos más adelante.

3.1.2. Lectura de Prefabs

Durante el desarrollo de esta herramienta dimos especial importancia a la experiencia de usuario, queriendo crear sistemas sencillos que permitan hacer cambios ágilmente. Por tanto, la tarea de seleccionar los objetos con los que crearemos el nivel debe ser lo más rápida e intuitiva posible.

En el diseño y creación de niveles, el escenario es rellenado por diferentes elementos, como enemigos, objetos interactuables u objetos que conforman el entorno. Estos elementos, formarán parte de los *assets* del videojuego, es decir, de sus recursos. En Unity, es común tratar estos *assets* como *Prefabs*. Un *Prefab* es un GameObject que se ha establecido como modelo predefinido para poder instanciarlo en la escena y crear tantas copias de él como se desee. Esto implica que el *Prefab* es la base de los GameObjects clonados a partir de él, por lo que un cambio en el *Prefab* supone modificar todas sus instancias —aunque también es posible desvincular un clon de su *Prefab* para poder modificarlo independientemente—.

Un *Prefab* puede abarcar desde una simple malla 3D que conforme un elemento decorativo hasta un complejo enemigo con componentes y scripts personalizados. Sin embargo, esto no suele ser relevante a la hora de dar forma al nivel, puesto que lo único importante es colocar cada GameObject en el lugar correspondiente. Es por

ello que nuestra herramienta debe tratar a todos los *Prefabs* de la misma manera ya que, a ojos del diseñador, todos son elementos que deberán ser colocados sobre el escenario.

Para informar a la herramienta sobre qué *Prefabs* se utilizarán, hemos considerado que la manera más sencilla e intuitiva es hacer uso del sistema de ficheros de Unity. De este modo, el paquete de Unity que distribuye la herramienta contará con un directorio llamado *Prefabs*, donde el usuario podrá crear subdirectorios que se identifiquen con diferentes categorías (figura 3.1). Estas categorías constituyen las diferentes formas de agrupar a los *Prefabs* que el diseñador considere. Así, por ejemplo, podrá crear subdirectorios para las categorías de enemigos, objetos o físicas, entre otras.

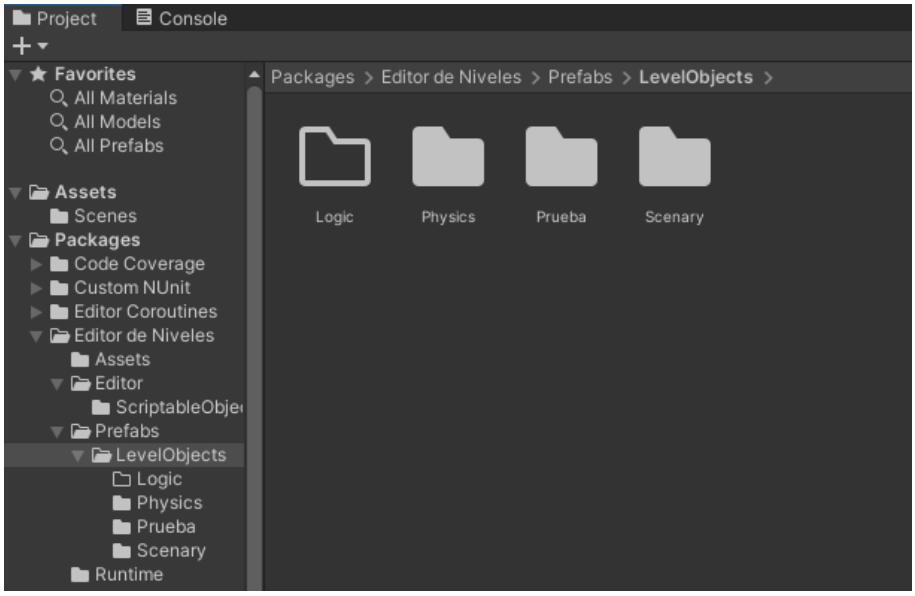

Figura 3.1: Subdirectorios creados para las categorías de la paleta.

En estos subdirectorios deberá ubicar los *Prefabs*, quedando así agrupados en categorías (figura 3.2). Los *Prefabs* podrán ser de cualquier tipo de objeto instanciable en una escena de Unity. Esto incluye tanto *sprites* 2D como mallas 3D, pues nuestro editor debe soportar ambos tipos de apartado gráfico. Las categorías, como veremos más adelante, supondrán la manera en la que la herramienta los ordenará para crear el nivel de manera más eficiente. Finalmente, la herramienta leerá cada subdirectorio contenido en el directorio *Prefabs* y creará una categoría por cada uno de ellos. A cada categoría se añadirán los *Prefabs* que se encuentren en el subdirectorio homónimo.

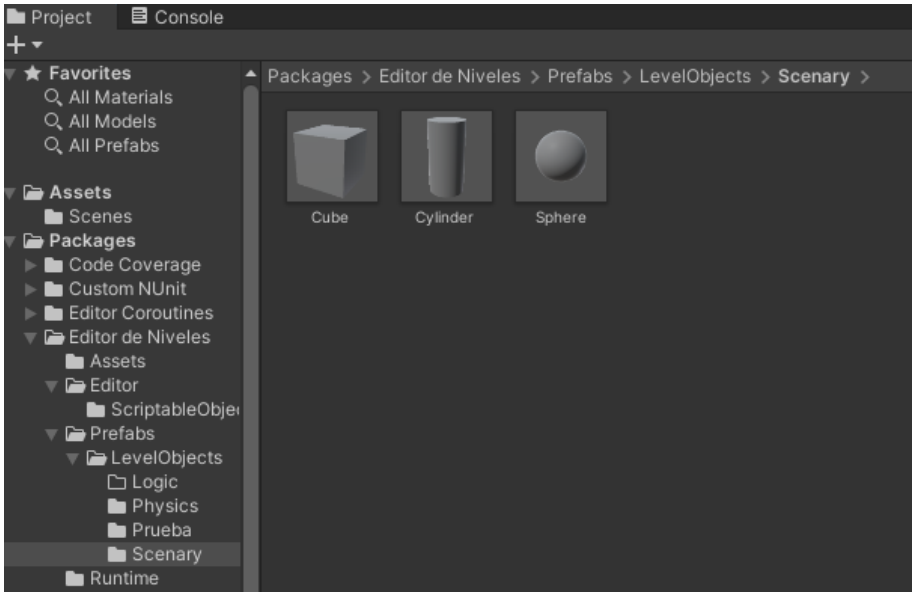

Figura 3.2: *Prefabs* de ejemplo añadidos al subdirectorio de la categoría *Scenary*.

3.1.3. Metadatos

Como explicamos anteriormente, al modificar los parámetros de un *Prefab* desde el inspector de Unity, los cambios también afectan a las instancias de ese *Prefab* ya existentes. Esto puede suponer un problema cuando queremos que las instancias no cambien, pues las únicas soluciones posibles serían crear un nuevo *Prefab* para cada variante del *Prefab* original que necesitemos, o bien desvincular cada clon del *Prefab* instanciado de su *Prefab* original. En proyectos grandes, donde se manejan grandes cantidades de *Prefabs*, cualquiera de estas dos soluciones resultarían muy poco prácticas, ya que añadirían recursos innecesarios al proyecto y ralentizarían el flujo de trabajo.

Para evitar esto, hemos diseñado un sistema con el que especificar ciertos parámetros o atributos de los *Prefabs*. Estos atributos se guardan como **metadatos**, almacenándolos en ***ScriptableObjects***. Un *ScriptableObject* es un objeto serializable de Unity que puede ser usado para almacenar datos de forma persistente. Los *ScriptableObjects* no se añaden como componente de un *GameObject* de la escena, sino que están presentes en forma de *assets* del proyecto. Mediante ellos, podemos especificar propiedades de los *Prefabs*, de manera que, al instanciarlos, posean determinadas características que los distingan del *Prefab* original. Esto permite cambiar características de cada objeto sin modificar su *Prefab*, aplicando estos cambios durante la instanciación del objeto.

Las propiedades a modificar dependerán del tipo de juego y las especificaciones del usuario. Por ejemplo, en una situación en la que se utilice un mismo *Prefab* en dos o más niveles, podría darse el caso en el que sus instancias deban tener una escala o rotación diferentes para adaptarse a las condiciones de cada nivel. Pero otro juego podría tener otras necesidades muy distintas. Es por esto por lo que el usuario podrá decidir qué parámetros guardar, definiéndolos en un script específico.

Por último, cabe destacar que, entre los parámetros a definir que debe incluir el *ScriptableObject* por defecto, es posible especificar el o los *prefabs* a partir de los cuales se instanciarán los GameObjects. Si solo añadimos un *prefab*, será ese el que se instancie. Sin embargo, si añadimos dos o más, cuando instanciemos ese objeto haciendo uso de nuestra herramienta, se añadirá a la cuadrícula un objeto escogido de manera aleatoria entre todos los especificados en el *ScriptableObject*. Esta funcionalidad es especialmente útil a la hora de dar forma al escenario del nivel, donde es común tener varios *prefabs* para el mismo bloque de terreno, cuya única diferencia es la malla 3D. Esto se hace para añadir variedad al apartado visual del nivel, de manera que resulte menos artificial a los ojos del jugador. Dicha tarea resultará muy sencilla y cómoda de realizar gracias a esta última funcionalidad del sistema de metadatos.

3.1.4. Paleta y herramientas de dibujo

Una vez hemos leído los objetos con los que se va a trabajar, necesitamos presentarlos al usuario de forma que pueda seleccionar cuál de ellos pintar. Para ello, consideramos que la mejor forma de representarlo es mediante una paleta, tal y como lo hacen otros editores de niveles como Tiled o *Super Mario Maker*. Una paleta es, en esencia, una ventana en la que aparecen todos los objetos que el usuario dispone para pintar. Estos objetos se presentan en forma de botones interactuables, de manera que cuando el usuario haga *click* sobre uno de ellos se seleccionará el objeto correspondiente.

En proyectos relativamente grandes, el gran número de objetos que conforman el nivel puede resultar abrumador y caótico si se presentan de manera arbitraria. La solución más obvia es agruparlos según alguna característica que los defina, por ejemplo, el tipo de objeto al que pertenecen: enemigos, *power-ups*, terreno, etc. De este modo, en la paleta debe haber un botón para cada una de estas categorías, de manera que cuando el usuario seleccione una de ellas, se mostrarán tan solo los objetos pertenecientes a dicha categoría. Esto permite al usuario filtrar para localizar el objeto que necesita rápidamente. Estas categorías coincidirán tanto en nombre como en contenido con los subdirectorios creados dentro del directorio *Prefabs*, tal y como se explicó en la subsección 3.1.2. Así, el usuario podrá ordenar sus objetos ubicándolos en diferentes subdirectorios —otorgándoles un nombre que represente a la categoría—, mientras el editor realiza esta equivalencia entre subdirectorios y categorías de la paleta. En la figura 3.3 se muestra el aspecto de la paleta con los objetos que se añadieron en la figura 3.2. Esta característica proporciona mucha flexibilidad a la hora organizar la paleta, permitiendo que se adapte a los diferentes tipos de juegos que se pueden desarrollar haciendo uso de nuestra herramienta.

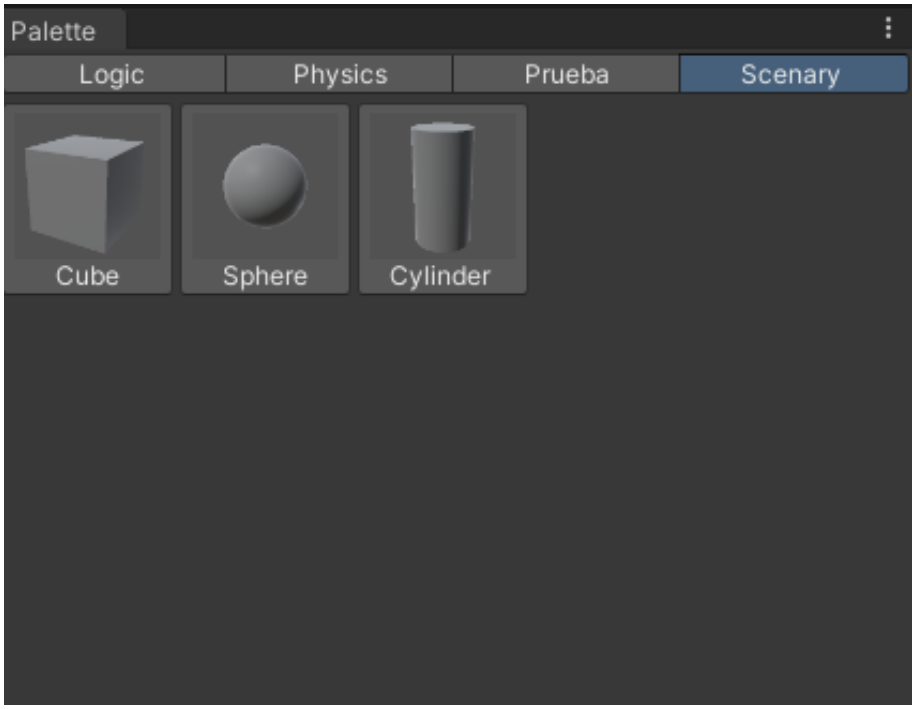

Figura 3.3: Paleta con los *prefabs* de la figura 3.2.

Por otro lado, la paleta debe poder ser reubicable dentro del proyecto de Unity, al igual que lo son el resto de ventanas e inspectores del propio motor. Por ello, hemos decidido implementar la paleta en una ventana de Unity, ya que esto permite tratarlo como un elemento más del *overlay*, pudiendo integrarlo fácil y cómodamente en él.

Una vez el usuario haya seleccionado el objeto con el que desee trabajar, el siguiente paso será instanciarlo sobre el grid de la escena. Para ello, necesitamos unas herramientas de dibujado. Las herramientas de dibujado indispensables son:

- Herramienta de vista o ***View***: permite desplazarse libremente por la cuadrícula. El usuario hará *click* sobre ella y la arrastrará hacia algún costado, permitiendo así ver zonas posiblemente ocultas debido al zoom de la cámara.

- Herramienta de pintar o ***Paint***: permite pintar objetos sobre la cuadrícula. Al tener seleccionada esta herramienta, si el usuario hace *click* sobre algún tile de la cuadrícula, se creará una instancia del objeto que haya seleccionado en la paleta, con los parámetros especificados en sus metadatos (tal y como se explicó en la sección 3.1.3). El objeto se ubicará siempre en el centro del tile, y sobre la capa que se tenga seleccionada, tal y como se explicó en la sección 3.1.1. Como solo puede haber un único objeto en un mismo tile dentro de una misma capa, si el usuario intentase colocar otro objeto nuevo se eliminará el objeto ya existente. Por último, para una mayor eficiencia a la hora de pintar múltiples objetos iguales, permitimos que el usuario pueda mantener pulsado el *click* izquierdo del ratón y lo arrastre sobre la cuadrícula, instanciando así un objeto sobre cada tile que se encuentre.

- Herramienta de borrar o ***Erase***: elimina el objeto, si lo hubiera, contenido en el tile sobre el que se hace *click*, dejando así un hueco libre para que lo ocupe otro.

Así, con la paleta y las herramientas de dibujo, el usuario podrá dar forma al nivel mediante la ubicación de objetos ordenada en tiles. Como mencionamos en la sección 3.1.2, los elementos a dibujar podrán ser tanto 2D como 3D, permitiendo así el diseño de niveles de juegos 2D puros y de juegos 2.5D, como es el caso de *Fulvinter*. Cabe destacar que nuestra herramienta no obliga a crear la totalidad del nivel haciendo uso de ella, sino que puede servir para hacer la estructura básica del nivel, restringiendo los objetos a tiles, mientras que otros elementos, como los decorativos, pueden ser colocados a mano libremente.

3.1.5. Persistencia

Todos estos sistemas no serían útiles si el trabajo del diseñador no fuese persistente, es decir, si no pudiese guardar su progreso al cambiar de escena o cerrar el programa. Los objetos instanciados haciendo uso de la herramienta son *GameObjects* completamente idénticos a los que obtendríamos instanciando manualmente el *prefab*. Por ello, su permanencia no supone un problema, ya que al cerrar la escena de Unity se le pregunta al usuario si desea guardar los cambios.

Sin embargo, la información necesaria para saber qué objetos han sido instanciados haciendo uso de la herramienta y en qué tile están ubicados, está alojada en un script de editor de Unity. Pues bien, esta información, al entrar en modo de juego o *Play Mode* —el modo que proporciona Unity que permite ejecutar el juego desde el propio proyecto para realizar pruebas—, se pierde. Esto se debe a que, en tiempo de ejecución, las variables de un script de editor no persisten en la escena o en los *assets*. Esto se conoce como *Domain Reloading* («Unity Domain Reloading», 2023). Esto implica que, si no lidiásemos con este problema, el usuario se vería obligado a rehacer el nivel cada vez que entra en modo de juego, cambia de escena o cierra Unity, pues toda la información que maneja el editor sobre los objetos instanciados se perdería. Si fuese así, el editor permitiría colocar más de un objeto en cada tile, además de que no sería posible eliminar los objetos instanciados antes de la pérdida de datos.

Es por ello que es necesario un sistema que asegure la persistencia de esta información. Por lo tanto, debemos poder recuperar la información de los objetos instanciados cuando usamos el *Play Mode* de Unity y cuando abrimos una escena en la que previamente habíamos trabajado.

Respecto al primer caso, consideramos que la mejor opción es guardar la referencia a cada objeto instanciado en un *Scriptable Object*. Para explicar por qué esto es posible, primero hemos de introducir el concepto de *InstanceID*. El *InstanceID* es un identificador único para cada objeto de la escena. Actúa como un manejador a la instancia en memoria del objeto («Unity InstanceID», 2025). Pues bien, cuando entramos en *Play Mode*, Unity clona la escena para trabajar sobre ella. Al salir del *Play Mode* y regresar al *Editor Mode*, Unity descarta la copia de la escena y restaura

la original («Enter Unity Play Mode Details», 2025). Debido a esto, el *InstanceID* de los objetos de la escena de antes de entrar y después de salir de *Play Mode* no cambia. Por ello y, volviendo al caso que nos ocupa, es posible guardar las referencias de los objetos de la escena en un *Scriptable Object* cuando nos disponemos a entrar en *Play Mode*, recuperarlas al salir y, así, poder seguir trabajando haciendo uso de nuestra herramienta.

Sin embargo, cuando realizamos un cambio de escena o cuando cerramos el proyecto de Unity, el *InstanceID* de los objetos cambia («Unity InstanceID», 2025). Por lo tanto, no es posible recuperar la información de los objetos instanciados de una manera similar a la comentada anteriormente. La solución a la que llegamos fue recuperar la información de cada GameObject a partir de la escena. De este modo, cuando abrimos una escena en la que hemos trabajado anteriormente, restauramos la información de cada GameObject sobre el lugar que ocupa en la cuadrícula e informamos a nuestro editor de niveles sobre ello. Esto es posible porque los objetos instanciados con nuestra herramienta cumplen con un orden concreto y ordenado en la jerarquía de la escena, lo que facilita su identificación. Gracias a esto, podemos cerrar la escena y la sesión de Unity y proseguir con el trabajo sobre ella más adelante sin ningún problema.

3.2. Implementación

En esta sección procedemos a realizar una descripción técnica de las clases y métodos que dotan a nuestra herramienta de las funcionalidades especificadas en la sección sobre el diseño 3.1. Por ello, hemos decidido componer esta sección de Implementación de las mismas subsecciones que la sección anterior de Diseño. De esta manera, podemos apreciar claramente la correspondencia entre decisiones de diseño e implementaciones técnicas de cada funcionalidad deseada.

3.2.1. Diagrama de Clases

La implementación de nuestro editor está dividida entre diversas clases que interactúan entre sí. Estas a su vez pueden ser organizadas dependiendo de su funcionalidad, como podemos ver en la figura 3.4.

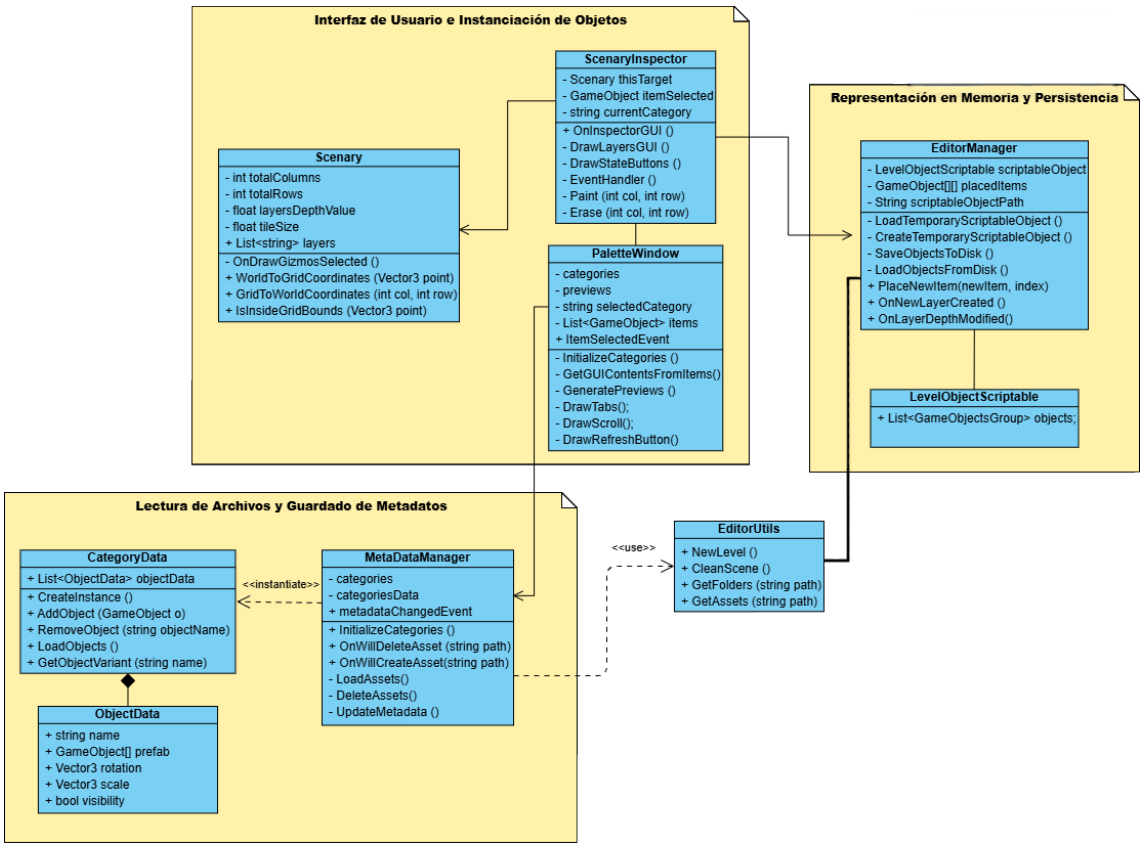

Figura 3.4: Diagrama de clases de nuestro editor

Los tres sistemas entre los que podemos dividir a nuestro editor son los siguientes:

- Interfaz de Usuario e Instanciación de Objetos: Las clases de este sistema se encargan de mostrar al usuario toda la parte visual (Cuadrícula, inspector, paleta). Se relacionan entre sí para comunicar las acciones del usuario, como los atributos del nivel o la selección de objetos. También se encargan de la instanciación de objetos cuando el usuario pinte en pantalla, ya que ***ScenaryInspector*** es el encargado de realizar cambios sobre el nivel.

- Representación en Memoria y Persistencia: Este sistema, compuesto principalmente por ***EditorManager***, se encarga de contener en memoria las referencias a los objetos del nivel durante el uso del editor, aplicando los cambios comunicados por ***ScenaryInspector***, y de asegurarse de que esta información se almacena entre estados de Unity y entre sesiones de trabajo.

- Lectura de Archivos y Guardado de metadatos: El sistema de Lectura de Archivos comprueba los objetos existentes en nuestra carpeta de ***Prefabs*** y se encarga de integrarlos en la lógica del editor. También comprueba la creación y eliminación de estos objetos en la carpeta, así como de la de las categorías. Este sistema está enlazado íntegramente con el Guardado de Metadatos, que almacena los parametros de instanciación de los objetos, ya que durante la creación o destrucción de un elemento debemos actualizar los metadatos existentes.

Estos sistemas se explorarán de forma detallada en la siguiente sección.

3.2.2. Lógica de Grid, capas y profundidad

A continuación procederemos a describir la lógica que compone los niveles de nuestro juego y las clases que la conforman. Estas clases son ***Scenary***, ***ScenaryInspector*** y ***EditorManager***. Parte de la implementación de estas clases está basada en el libro *Extending Unity with Editor Scripting* (Tadres, 2015).

Scenary es nuestra única clase de ***Runtime*** (una clase asociada a un objeto de la escena). Contiene los parámetros dimensionales básicos para definir las dimensiones de la cuadrícula y las capas que contiene. La clase ***Scenary*** va asociada al objeto del mismo nombre, el cual se instancia al crear un nivel y será el objeto padre a partir del cual crearemos nuestra jerarquía (Tadres, 2015).

Sus parámetros son: el número de columnas (***totalColumns***), número de filas (***totalRows***), el tamaño de las casillas (***tileSize***), la lista de capas (***layers***) y el espacio entre capas (***LayersDepthValue***), como podemos ver en la figura 3.5.

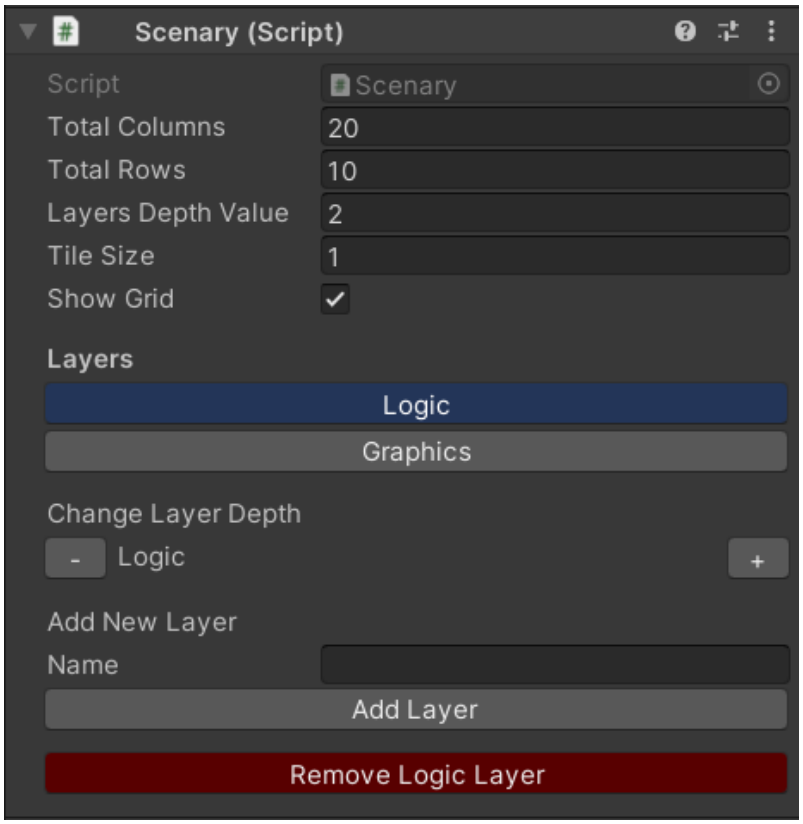

Figura 3.5: Vista en el inspector de la clase Scenary

Con estos parámetros podemos construir una representación visual del nivel, dibujando la cuadrícula mediante la clase ***UnityEngine.Gizmos***. Utilizamos el método ***Gizmos.DrawLine()*** para dibujar las filas, columnas y fronteras, mostrándolas directamente en la vista de ***Scene***, como se aprecia en la figura 3.6. El grid puede ser ocultado a través del inspector de *Scenary*.

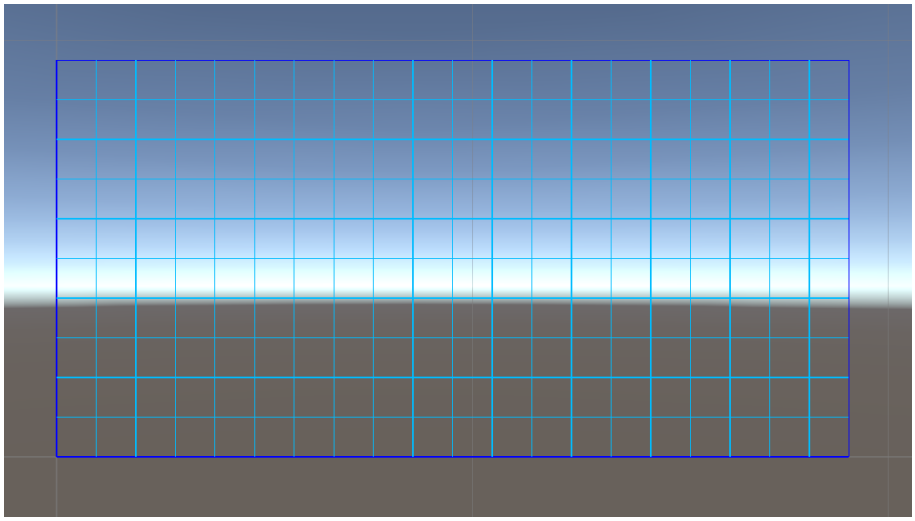

Figura 3.6: Cuadrícula del nivel en su estado inicial

Como ***Scenary*** es una clase de ***Runtime***, no puede realizar cambios en el estado de edición de Unity más allá del dibujado de ***Gizmos*** que vimos anteriormente. Por ello necesitamos la clase ***ScenaryInspector***, que hereda de ***UnityEditor.Editor*** y toma a ***Scenary*** como ***Target***, utilizando sus atributos para implementar funcionalidades en el estado de edición (Tadres, 2015).

Además, ***ScenaryInspector*** define la interfaz visual de los botones de pintado y borrado, junto con su lógica de funcionamiento, incluyendo la instanciación de objetos al pintar y la comunicación con la clase ***EditorManager***.

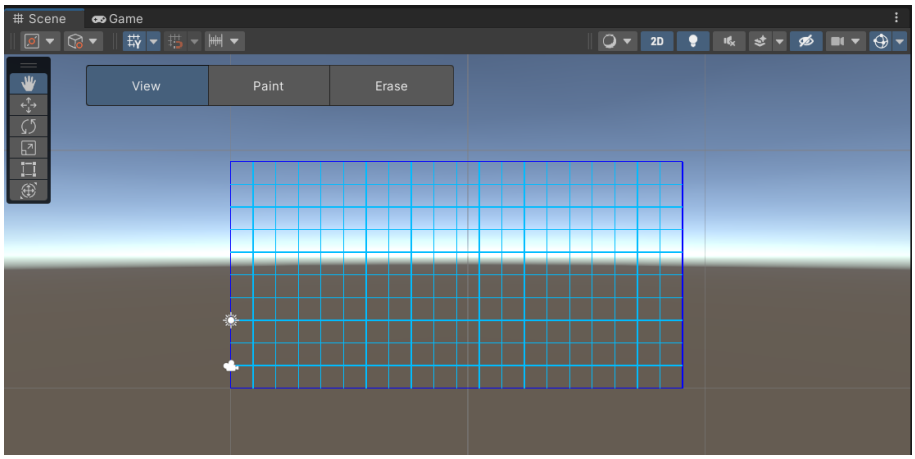

Figura 3.7: Vista de la escena con los botones del editor

Como podemos ver en la figura 3.7, la herramienta dispone de tres botones: ***View***, ***Paint*** y ***Erase***. Cada uno de ellos corresponde a una herramienta distinta de nuestro editor, como describimos en la sección 3.1.4, representadas por el tipo enumerado ***EditorState*** que se almacena en ***ScenaryInspector***.

Para implementar estas herramientas, necesitamos hacer uso de la clase ***PaletteWindow***, que proporciona los objetos disponibles para instanciar, y ***EditorMa-***

nager, que gestiona el array de objetos ya colocados en el nivel.

Dentro de ***ScenaryInspector*** se define un listener para el evento ***ItemSelectedEvent*** de ***PaletteWindow***, el cual notifica del último objeto seleccionado por el usuario. Este objeto se almacena en la variable ***itemSelected*** y será el objeto a pintar (Tadres, 2015).

El método ***ScenaryInspector.EventHandler()*** se encarga de detectar si el usuario hace *click* sobre el nivel y, en ese caso, ejecutar las acciones correspondientes según el estado actual:

- ***View*** permite al usuario desplazarse por la escena sin modificar ningún elemento, arrastrando el ratón para moverse. Para implementar este comportamiento, basta con asignar la herramienta actual de Unity (***UnityEditor.Tools***) a ***UnityEditor.Tool.View***, la cual ya proporciona la funcionalidad deseada.
- ***Paint*** crea instancias del objeto seleccionado al hacer *click* en la cuadrícula. Para ello, primero obtenemos la posición en el mundo del cursor y la convertimos a coordenadas de la cuadrícula mediante el método ***WorldToGridCoordinates()*** de la clase ***Scenary***. Con la columna y fila resultantes, llamamos al método ***Paint(int col, int row)***. Este método verifica si la posición calculada está dentro de la cuadrícula y, en caso afirmativo, obtiene el índice y la capa correspondientes. Si aún no existe un objeto instanciado para esa capa, se crea como hijo del objeto ***Scenary***. A continuación, se instancia el objeto almacenado en ***itemSelected***, se ubica en la jerarquía de la escena bajo la capa adecuada y se traspasa a ***EditorManager*** mediante el método ***PlaceNewItem()***.
- ***Erase*** elimina los objetos de la capa seleccionada sobre los que se haga clic. Al igual que en ***Paint***, primero se calculan las coordenadas de la cuadrícula. Luego, se invoca el método ***Erase(int col, int row)***, que comprueba que la posición sea válida y notifica a ***EditorManager*** llamando a su ***método*** DeleteItem().

La representación en memoria de los objetos del nivel se gestiona desde la clase ***EditorManager***. Es decir, esta clase se encarga de almacenar y organizar las referencias a todos los objetos del nivel, permitiendo su acceso y manipulación. Para ello, utiliza el array bidimensional de ***GameObjects*** llamado ***placedItems***: la primera dimensión corresponde a las capas que contienen al menos un objeto, mientras que la segunda almacena los objetos de la capa seleccionada.

Anteriormente mencionamos los métodos ***PlaceNewItem()*** y ***DeleteItem()***, utilizados durante el pintado y borrado de objetos.

- ***DeleteItem()*** recibe la capa actual y el índice de la posición a eliminar. Primero comprueba si esa posición está ocupada y, de ser así, destruye el objeto correspondiente.
- ***PlaceNewItem()*** recibe el objeto a insertar, junto con el índice de su posición y la capa correspondiente. Verifica si la posición ya está ocupada en esta capa,

y si es así, elimina el objeto existente utilizando ***DeleteItem()***. Después se añade el nuevo objeto al array.

Cuando una capa es creada, eliminada, o su profundidad se modifica, ***ScenaryInspector*** invoca a los métodos ***OnNewLayerCreated()***, ***OnLayerDeleted()*** o ***OnLayerDepthModified()*** de ***EditorManager***, respectivamente.

- ***OnNewLayerCreated()*** redimensiona el array ***placedItems*** para dar cabida a la nueva capa.
- ***OnLayerDeleted()*** elimina los objetos contenidos en la capa borrada y ajusta el tamaño del array en su primera dimensión.
- ***OnLayerDepthModified()*** recorre los objetos de la capa correspondiente y transforma su posición en el eje perpendicular a la cuadrícula.

Como veremos en la sección 3.2.6, ***EditorManager*** también se encarga de la persistencia de los objetos, guardando los contenidos de ***placedItems*** durante el estado de juego y entre sesiones de trabajo.

3.2.3. Lectura de Prefabs desde el sistema de ficheros

Como hemos explicado en la sección 3.1.2, hemos creado un sistema de lectura de ***Prefabs*** para facilitar la adición de objetos al entorno de la herramienta. Para esto, empleamos la clase ***MetaDataManager***, que se encarga tanto de este proceso como del guardado de los *metadatos* de los objetos leídos, como veremos en la siguiente sección 3.2.4.

Llamamos categoría a un conjunto de objetos que estarán disponibles para pintar en el editor. Estas categorías se almacenan en el diccionario ***categories*** de ***MetaDataManager***, lo que permite un acceso sencillo a los objetos una vez cargados.

La creación inicial de las categorías se realiza en el método ***InitializeCategories()***, invocado desde el constructor de la clase. Para ello, se emplea el método ***GetFolders()*** de la clase auxiliar ***EditorUtils***, que genera el diccionario vacío y recorre los subdirectorios disponibles. Cada subdirectorio se añade al diccionario como una nueva categoría, mientras que los objetos de tipo Prefab que contiene se incorporan a la lista correspondiente de esa categoría.

También tenemos en cuenta la creación y destrucción de ficheros y objetos dentro del directorio ***Prefabs***. Podemos detectar si un asset ha sido creado o destruido mediante los callbacks ***OnWillCreateAsset()*** y ***OnWillDeleteAsset()***. Si uno de estos métodos es llamado, extraemos el camino del asset correspondiente y comprobamos si se trata de nuestra carpeta de Prefabs. Si es el caso, se examina la extensión del archivo para distinguir entre ficheros, ***Prefabs***, u otros tipos de assets que no necesitamos procesar.

Si nos encontramos ante un directorio, debemos crear o destruir la categoría correspondiente. En el caso de haberla creado, tenemos que comprobar si el directorio

contiene objetos de tipo ***Prefab*** que añadir a la categoría. En el caso de haberla destruido, nos aseguramos de borrar toda referencia a esta categoría y a los objetos que contenía.

Por lo contrario, si el asset creado o destruido es un objeto de tipo ***Prefab***, simplemente lo añadimos o eliminamos de su categoría.

Cabe destacar que tanto la carga como la eliminación de assets en memoria no se pueden realizar directamente en los callbacks, ya que se llaman en el momento previo a la modificación del asset. Por lo tanto, necesitamos una lista de objetos a cargar, ***assetsToLoad***, y otra lista para los objetos a eliminar, ***assetsToDelete***. Añadiremos los assets a estas listas durante el callback, esperando al método ***Update()*** para recorrerlas y modificar las categorías de la manera que corresponda.

3.2.4. Sistema de guardado de metadatos

En la sección 3.1.3 explicamos los problemas que puede suponer la modificación de un ***Prefab*** cuando éste ya ha sido instanciado. Para evitar esto, guardamos ciertos atributos de cada *Prefab* como *metadatos*, almacenándolos en ***ScriptableObjects***.

Para representar a estos *metadatos*, hemos creado la clase ***ObjectData***, la cual contiene los atributos a tener en cuenta durante la instanciación del objeto, como su ***GameObject***, escala, rotación, etc. Esta clase es completamente personalizable por el usuario, de manera que pueda contener los atributos y componentes que necesite.

Hemos decidido que estos *metadatos* se agrupen por categorías, por lo que se almacenarán en una lista de ***ObjectData*** en nuestra clase de ***ScriptableObject*** llamada ***CategoryData***.

Como vimos en la sección 3.2.3, la clase ***MetaDataManager*** lee el contenido del directorio ***Prefabs*** y crea las categorías correspondientes en el método ***InitializeCategories()***. Esta información no solo se almacena en el diccionario ***categories***, sino que también se utiliza para generar las instancias de ***CategoryData*** mediante el método ***CategoryData.CreateInstance()***.

Este método recibe el nombre de la categoría y la lista de objetos que contendrá. Si ya existe una instancia de ***CategoryData*** creada en una sesión anterior, se comprueba si el número de elementos de su lista coincide con el de la lista de entrada; en caso contrario, se considera desactualizada y se sobrescribe. Si no existe tal instancia, se crea y se rellena su lista de ***ObjectData*** invocando al método ***LoadObjects()***.

LoadMetadata() convierte una lista de ***GameObjects*** en ***ObjectData***. Por tanto, este método debe ser modificado por el usuario cuando cambie los atributos de la clase ***ObjectData***.

Cuando ***MetaDataManager*** observe que se ha creado o destruido un objeto, se encargará de que la instancia de ***CategoryData*** correspondiente permanezca actualizada, llamando su método ***AddObject()*** o ***RemoveObject()*** respectivamente.

El método ***LoadObjects()*** crea la lista de objetos de la categoría en memoria

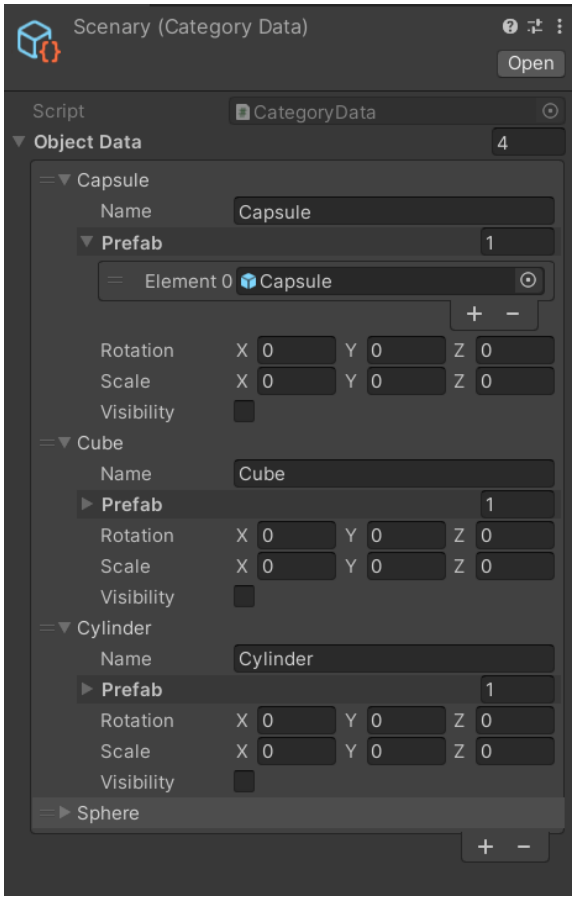

Figura 3.8: Una instancia de CategoryData con los *metadatos* de varias primitivas

basándose en los valores de su ***ObjectData***. Este método debe ser modificado por el usuario cuando cambie las variables de **ObjectData**.

3.2.5. Representación de la Paleta

En la sección 3.1.4 explicamos el concepto de la paleta, un elemento de interfaz visual que permite al usuario seleccionar objetos de manera rápida y fluida. Esta paleta está representada por la clase ***PaletteWindow***, la cual hereda de la clase ***EditorWindow***. Por lo tanto, posee las funcionalidades de cualquier ventana de Unity, como ser arrastrada, maximizada y añadida al *Layout* como un elemento más (Tadres, 2015).

Al igual que ***MetaDataManager***, ***PaletteWindow*** mantiene un diccionario de categorías que contiene los objetos a pintar. Este diccionario se obtiene llamando al método ***RetrieveCategories()*** de ***MetaDataManager***, invocado durante el método ***InitializeCategories()*** de ***PaletteWindow***.

RetrieveCategories() se asegura de que el diccionario de objetos esté actualizado con la información de los *metadatos*, utilizando el método ***LoadObjects()*** de cada ***CategoryData()***.

Cuando ***MetaDataManager*** detecta la creación o eliminación de una categoría u objeto, lanza un evento llamado ***metadataChangedEvent***. ***PaletteWindow*** posee un *listener* a este evento que utiliza para mantener su información actualizada en tiempo real.

Tras obtener el diccionario, se crea otro del mismo tamaño que almacenará las texturas que mostraremos como imagen en la interfaz. A este diccionario lo llamamos ***previews***. También se establece la categoría actual como la primera categoría del diccionario, en ***selectedCategory***, y se inicializa la lista ***items***, que contiene los objetos de la categoría actual, extrayéndolos del diccionario.

Para rellenar el diccionario ***previews*** utilizamos el método ***GeneratePreviews()***, que recorre la lista ***items*** y extrae la textura de sus previews, mediante el método ***AssetPreview.GetAssetPreview()***. Si ese objeto no tiene una textura válida, se le asigna una por defecto.

Como vemos en la figura 3.9, la paleta tiene una pestaña o *tab* por cada categoría, una barra de *scroll* en el caso de que haya muchos objetos, y un botón de ***Refresh*** para actualizar la información que contiene cuando se hayan realizado cambios a los *metadatos*.

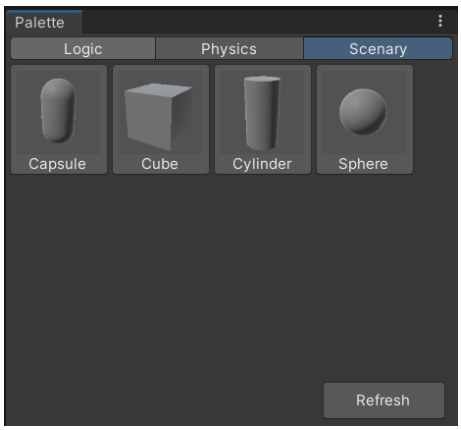

Figura 3.9: Paleta con las primitivas básicas de Unity

El botón de ***Refresh*** se encarga de que el diccionario ***categories*** vuelva a ser rellenado con los objetos que podamos extraer de los *metadatos* existentes, actualizando así la paleta en caso de que haya habido algún cambio.

Cuando se hace clic sobre una preview, ***PaletteWindow*** identifica el objeto correspondiente y lanza el evento ***ItemSelectedEvent()***, al cual escuchará ***ScenaryInspector()*** para actualizar su objeto a instanciar (Tadres, 2015).

3.2.6. Persistencia del estado del nivel

Como explicamos en la sección 3.1.5, necesitamos varios sistemas para guardar el progreso del usuario, de manera que nuestro editor retenga la información del estado del nivel entre estados del editor de Unity (Persistencia durante la ejecución),

y cuando se cambia de escena o entre sesiones de trabajo (Persistencia fuera de la ejecución).

La clase encargada de la persistencia es ***EditorManager***, ya que es la que gestiona el manejo en memoria de los objetos del nivel.

3.2.6.1. Persistencia durante la ejecución

Con persistencia durante la ejecución nos referimos al guardado del estado del nivel al cambiar entre el modo de edición y el modo de juego en Unity.

Dado que nuestras clases están orientadas al modo de edición, no pueden conservar información en memoria durante la ejecución. Para resolverlo, almacenamos los datos esenciales del editor, los objetos instanciados, en un ***ScriptableObject***.

Para ello, hemos creado la clase ***LevelObjectScriptable***, que guarda una lista de ***GameObjectsGroup***. Cada ***GameObjectsGroup*** representa una capa y contiene la lista de ***GameObjects*** pertenecientes a ella.

EditorManager define un listener para el evento ***playModeStateChanged*** de ***EditorApplication***. Cuando este listener se activa, invoca al método **OnPlayModeStateChanged()**, que evalúa el estado al que está cambiando el editor de Unity:

- ***ExitingEditMode***: al salir del modo de edición y entrar en modo de ejecución se llama al método ***SaveObjectsToDisk()***, el cual inicializa la lista de ***LevelObjectScriptable*** y la rellena creando un ***GameObjectsGroup*** por cada capa, almacenando en él los objetos correspondientes.
- ***EnteredEditMode:*** Al volver al modo de edición se invoca al método ***LoadObjectsFromDisk()***, que vuelve a inicializar al array ***placedItems*** y lee los objetos almacenados en ***LevelObjectScriptable***, guardandolos en el array.

3.2.6.2. Persistencia fuera de la ejecución

La persistencia fuera de la ejecución es la que permite el guardado al cambiar de escena o cerrar el programa. Inicialmente se intentó seguir una implementación parecida a la persistencia durante la ejecución utilizando ***LevelObject***, pero esta implementación tuvo que ser descartada ya que las referencias a objetos de la escena desaparecen de memoria cuando la escena deja de estar cargada.

Así, la solución más práctica terminó siendo utilizar la persistencia de escena de Unity, confiando en el motor para el guardado y carga de los elementos de la escena. De esta manera, para realizar el guardado, notificamos a Unity cuando se realizan cambios en el nivel, de manera que se incluyan en el guardado de la escena.

Para la carga del guardado, creamos en ***EditorManager*** un *listener* al evento de Unity ***OnSceneOpened***, el cual nos notifica cuándo una escena se abre. Este listener invoca al método ***OnSceneOpened()*** de ***EditorManager***, el cual contiene la funcionalidad de carga.

Este método toma el objeto ***Scenary*** bajo el que se encuentra la jerarquía del nivel, y lee cada objeto que sea hijo directo de este. Si el formato del objeto corresponde a una capa, se asigna en memoria como la capa correspondiente y se leen sus hijos, extrayendo de ellos la columna y fila que ocupan y asignándoles su posición en el array ***placedItems***.

3.2.7. Uso del Editor

Una vez implementadas estas funcionalidades, el editor estaba listo para ser probado. Nuestra meta es poder crear niveles tanto para juegos 2D como para juegos 2.5D. Para comprobar si nuestro editor es capaz de hacer esto, nos disponemos a intentar recrear niveles de los juegos Fulvinter y Wizara.

3.2.7.1. Fulvinter

Fulvinter es un juego 2.5D de plataformas cooperativo con vista lateral. El objetivo del jugador es avanzar por los niveles, esquivando obstáculos y colaborando con su compañero para superar los saltos y trampas que se encuentren.

Obtuvimos una versión del proyecto de parte de nuestro director, Ismael Sagredo, a partir de la cual pudimos comprobar las funcionalidades de nuestra herramienta.

Como nuestro editor es un paquete de Unity, podemos introducirlo al proyecto de Fulvinter sencillamente. Para esto, solo necesitamos mover la carpeta de nuestro editor a la carpeta de paquetes del proyecto. Una vez hecho, comprobamos que todo funcionaba correctamente.

A la hora de decidir qué nivel recrear, nos decantamos por el nivel ***Level01.S01.Art***, que se encuentra directamente después del nivel inicial y tutorial del juego. Tomamos esta decisión basándonos en los contenidos de este nivel, el cual tiene algunos de los elementos clave del juego, como las semillas (que actúan a modo de monedas), puntos de tutorial y de reaparición, y muros que los jugadores deberán escalar mediante saltos en la pared. Podemos ver este nivel en la figura 3.10.

Figura 3.10: Nivel Level01.S01.Art de Fulvinter

Para crear un nuevo nivel con nuestro editor, utilizamos la sección del menú de Unity llamada ***Level Editor***, en la que se encuentra el botón ***NewLevel*** de nuestra herramienta. Así, se crea una escena nueva con un objeto ***Scenary***, que representa nuestro nivel.

Los niveles de Fulvinter están compuestos principalmente por bloques de forma cuboide, divididos en tres categorías: Lógica, físicas y gráficos. Por lo tanto, a la hora de organizar los objetos que vamos a utilizar, hemos decidido dividirlos en esas tres categorías. En nuestra carpeta Prefabs, creamos una subcarpeta por categoría, preparando así el editor para introducir los objetos que necesitemos más adelante.

A continuación, examinamos qué capas componen el nivel original, y qué objetos se encuentran en cada una de ellas. Podemos ver estas capas en la figura 3.11.

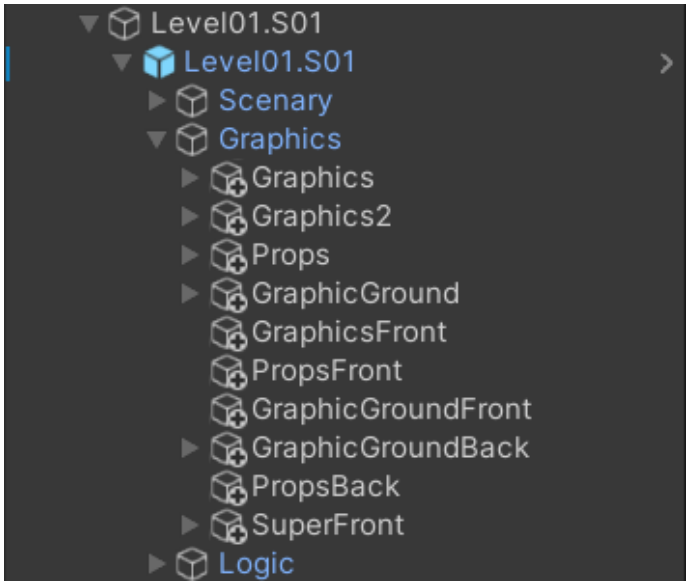

Figura 3.11: Jerarquia de las capas del nivel Level01.S01.Art

- ***Logic***: Contiene objetos que definen las mecánicas del juego, como las semillas, los puntos de reaparición y objetos clave para el funcionamiento de la cámara.

- ***Scenary***: Contiene los bloques físicos que controlan las colisiones del jugador con el entorno.

- ***Graphics***: En esta capa se encuentran todos los bloques visuales del escenario y otros objetos que dan al juego su aspecto característico. Esta capa se encuentra organizada mediante varias subcapas.

Examinando estas capas pudimos extraer los objetos necesarios para construir el nivel, introduciéndolos en las subcarpetas que representan sus categorías para poder trabajar con ellos. Una vez hecho esto, el estado de la paleta es el que se puede ver en las figuras 3.12 y 3.13.

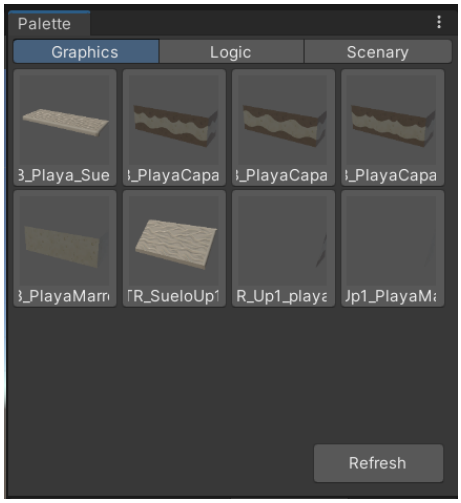

Figura 3.12: Paleta con los objetos gráficos de Fulvinter

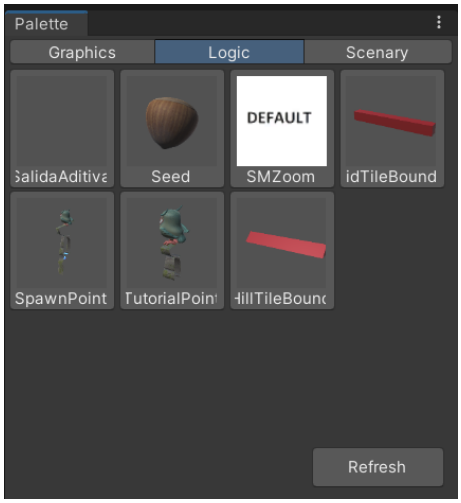

Figura 3.13: Paleta con los objetos lógicos de Fulvinter

Ahora que tenemos los objetos con los que vamos a trabajar, podemos crear un nuevo nivel desde nuestra herramienta. Iremos pintando el nivel capa por capa para recrear el original.

Asignamos a ***Scenary*** el tamaño adecuado, en este caso, 130 columnas y 54 filas, y creamos las capas ***Logic***, ***Graphics*** y ***Scenary***, siguiendo el ejemplo original. Después, recreamos la capa física y lógica, ya que poseen menos objetos y sirven como una buena base para construir el resto del nivel.

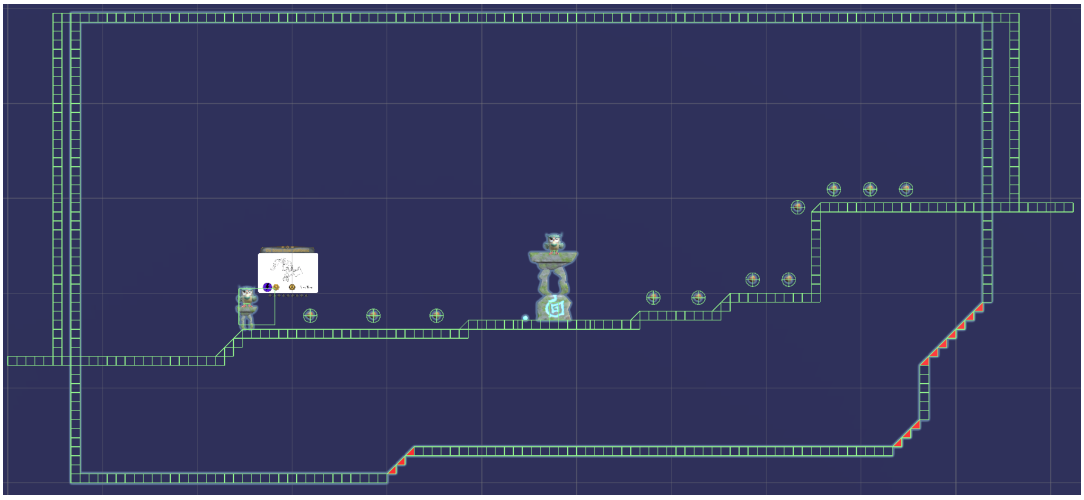

Figura 3.14: *Layout* del nivel con los objetos lógicos y los bloques físicos

Una vez tenemos esto, el nivel ya es técnicamente jugable, ya que los objetos gráficos no afectan a la funcionalidad del juego. Realizamos una prueba para comprobar que todo funcionase correctamente, añadiendo a los personajes jugables y consiguiendo completar el nivel con ellos.

Después, procedemos a completar aquellas capas gráficas que se encuentren en la misma profundidad que el jugador, para que haya así una estructura de nivel visible y navegable.

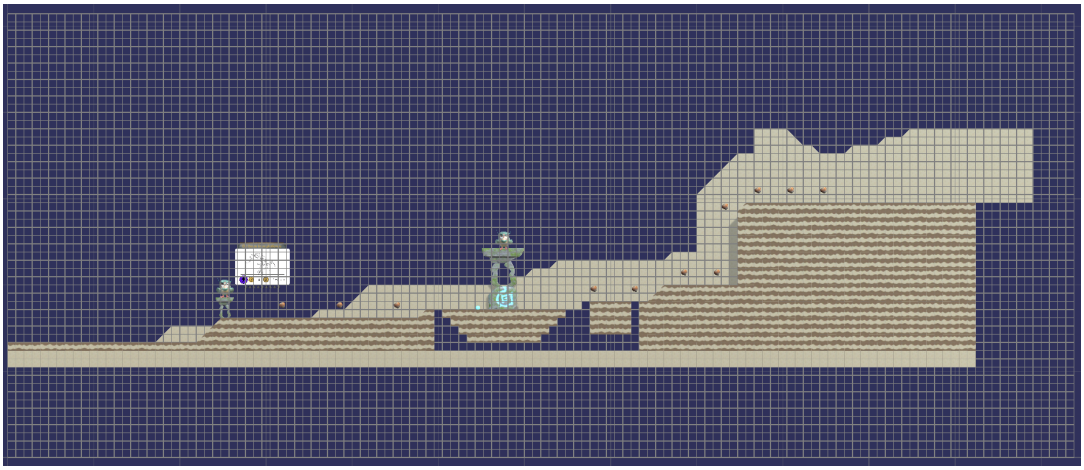

Figura 3.15: Primeras capas gráficas del nivel

Como podemos ver, el patrón de bloques del suelo y las paredes se alterna entre tres objetos distintos con ligeras variaciones, de manera aleatoria. Para reducir el tiempo que toma pintar un patrón así, hemos utilizado una funcionalidad de nuestro sistema de metadatos, la cual permite asignar varios ***Prefabs*** al mismo objeto, de manera que durante el pintado se pueda instanciar uno de ellos escogido al azar. Podemos ver cómo se ve nuestro ***ObjectData*** al realizar este proceso en la siguiente figura 3.16.

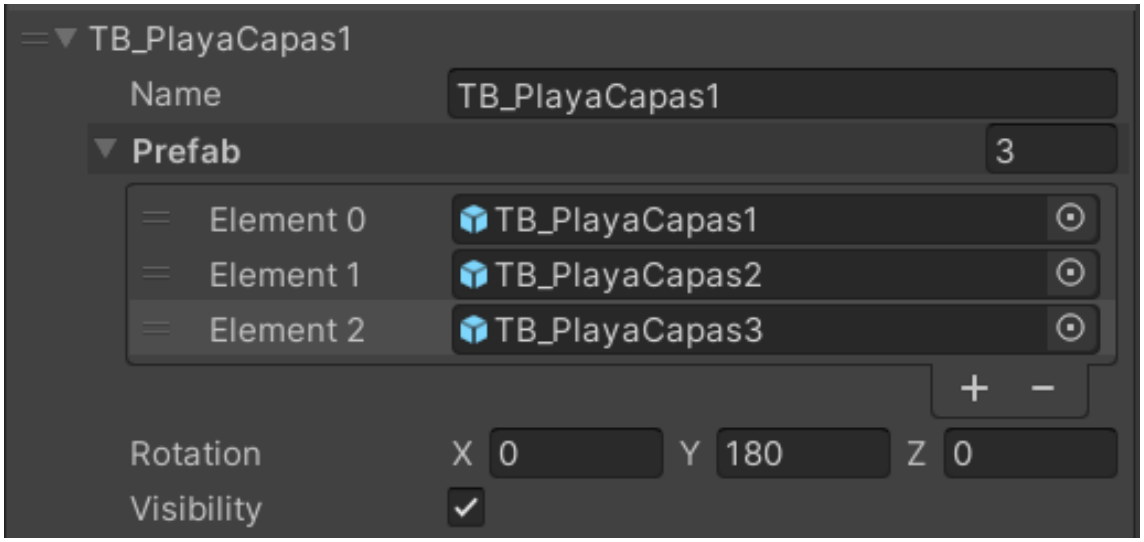

Figura 3.16: *ObjectData* con varios *Prefabs*

Por último, debemos completar aquellas capas gráficas cuya profundidad es distinta, como las que se encuentran al fondo o al frente. Es aquí donde podemos probar nuestra implementación de profundidad, pudiendo crear capas en la coordenada Z que queramos y pudiendo moverlas en ese eje. Una vez hemos pintado estas capas, el nivel queda completado, sin ninguna diferencia significativa con el original.

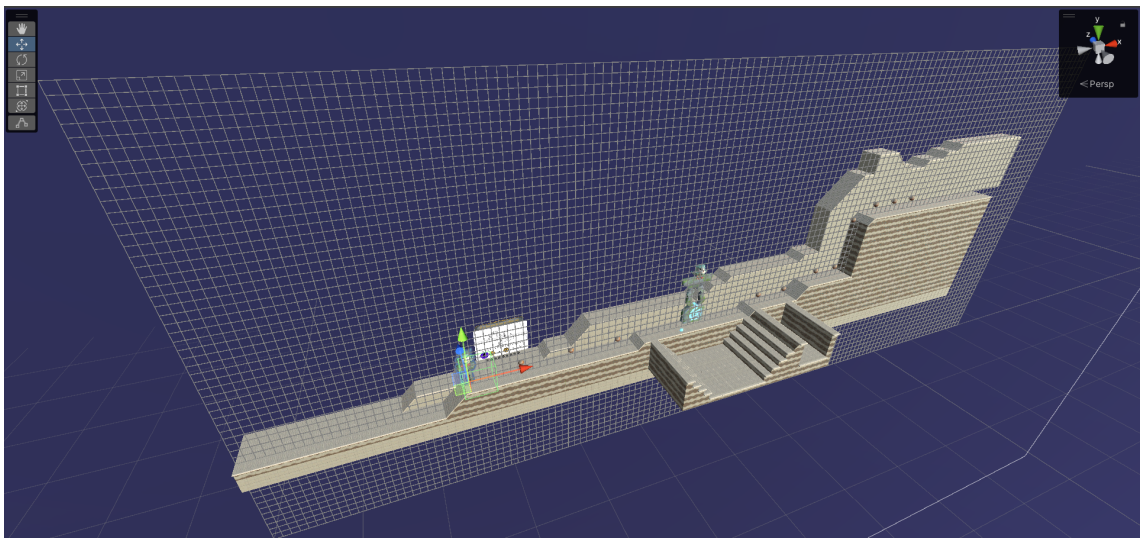

Figura 3.17: Resultado final del nivel

Esto nos demuestra no solo que nuestro editor funciona, sino que contiene las funcionalidades necesarias para recrear las funcionalidades principales de los niveles originales de Fulvinter. Algunas funcionalidades más avanzadas no se pudieron incluir por falta de tiempo, como la creación de zonas de agua o las plataformas móviles. El flujo de trabajo fue ágil y sencillo, permitiendo crear esta réplica muy rápidamente.

3.2.7.2. Wizara

Wizara es un juego 2D de plataformas de tipo *metroidvania*, con niveles caracterizados por estar divididos en distintas salas y minijuegos que se utilizan para desbloquear habilidades. Wizara fue desarrollado para la asignatura de Proyectos 1 del Grado de Videojuegos, por el equipo Flip&Floop, del cual formó parte Sol Flora López, coautora de este proyecto.

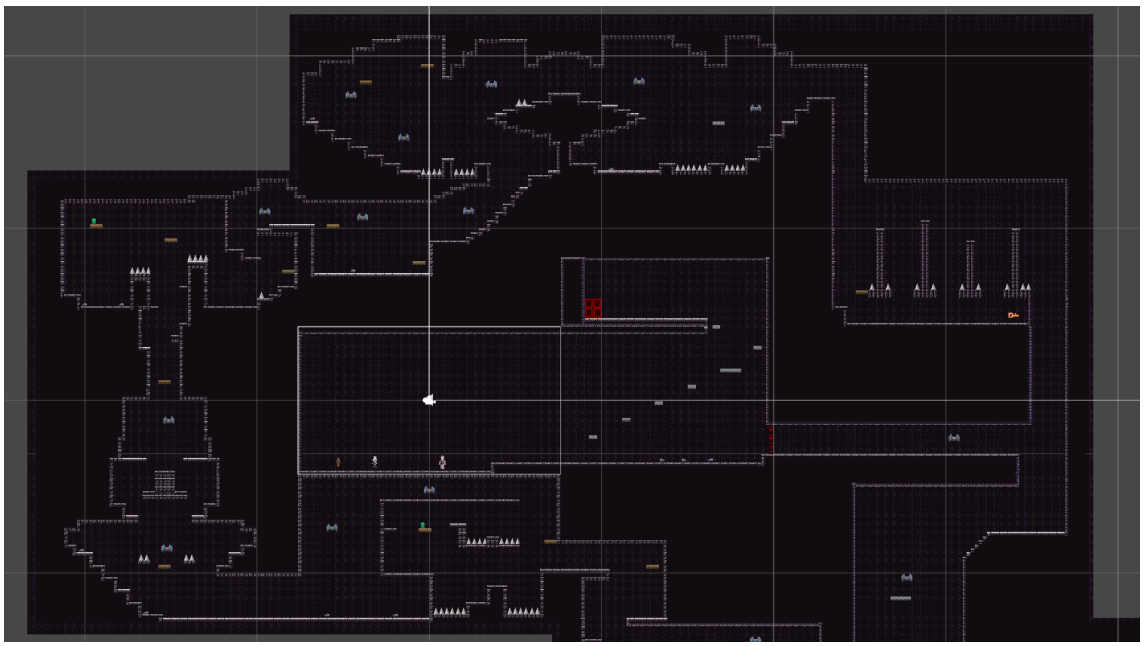

Figura 3.18: Sección de un nivel de Wizara

Para comprobar que nuestro editor es utilizable a la hora de desarrollar niveles en dos dimensiones, hemos decidido crear un nivel completamente original de este proyecto.

Para ello, introducimos nuestro paquete al proyecto de Unity y creamos un nuevo nivel. Antes de empezar a dibujar con nuestra herramienta, necesitamos saber qué objetos vamos a necesitar e introducirlos en la carpeta de ***Prefabs***.

Es aquí donde nos encontramos con la primera diferencia respecto al flujo de trabajo que utilizamos en el nivel de Fulvinter. Como Wizara es un juego en dos dimensiones, las piezas del escenario están compuestas por ***Sprites*** que forman parte de la paleta de un ***Tilemap*** (visto en la sección 2.1.2).

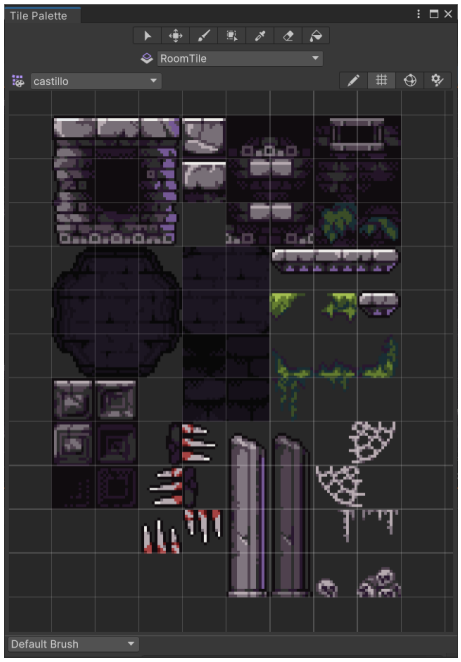

Figura 3.19: Paleta de *Tilemap* de Wizara

Por lo tanto, para utilizar estos bloques debemos convertirlos primero a ***Prefabs***. Este proceso es sencillo, ya que podemos arrastrar los ***sprites*** que queramos a la escena, de manera que se conviertan en ***GameObjects***, y luego guardarlos en nuestra carpeta de ***Prefabs***, pero resulta tedioso si la cantidad de sprites es muy alta.

A continuación, identificamos qué otros objetos necesitamos introducir a nuestro editor para crear el nivel. Además del escenario, necesitaremos al jugador, los enemigos y otros objetos interactuables, así como el fondo de nivel y una manera de crear las colisiones. El resultado de este proceso puede verse en la figura 3.20

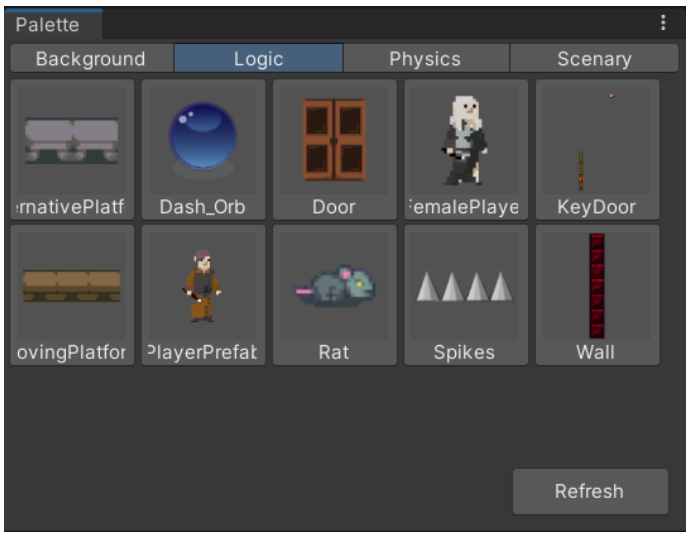

Figura 3.20: Paleta de nuestro editor en Wizara

Ahora que tenemos todo lo que necesitamos, nos disponemos a crear un nivel. Primero, necesitamos una base para la estructura que tenemos en mente. Las salas de Wizara tienen una proporción establecida de 16:9, y el tamaño de las casillas es de 0.25 unidades de Unity. Con estos datos, podemos calcular que el tamaño de una sala es de 64 columnas por 36 filas; hemos decidido crear un nivel con 9 salas, en una estructura de 3 por 3, por lo que el tamaño total de nuestra cuadrícula es de 192 columnas y 108 filas.

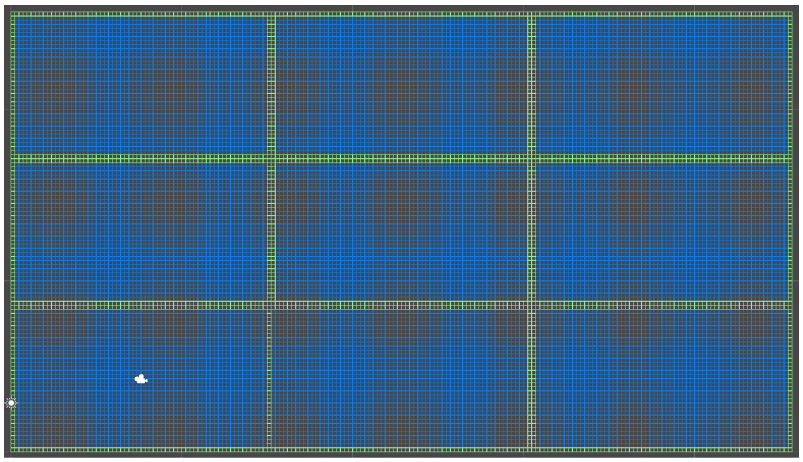

Figura 3.21: División de las salas del nivel de Wizara

Empezamos a trabajar en la primera sala, pintando los muros y paredes del nivel con bloques gráficos para crear la forma que queremos. Después, añadimos el fondo y los bloques de colisión. Cabe destacar que, en un proyecto de Unity en dos dimensiones, tener bloques de colisión separados puede hacer que el jugador quede atascado entre ellos. Para arreglar esto, aplicamos un componente de ***Composite Collider*** a la capa de físicas y hacemos que todos los bloques de la capa formen parte de este ***Collider***.

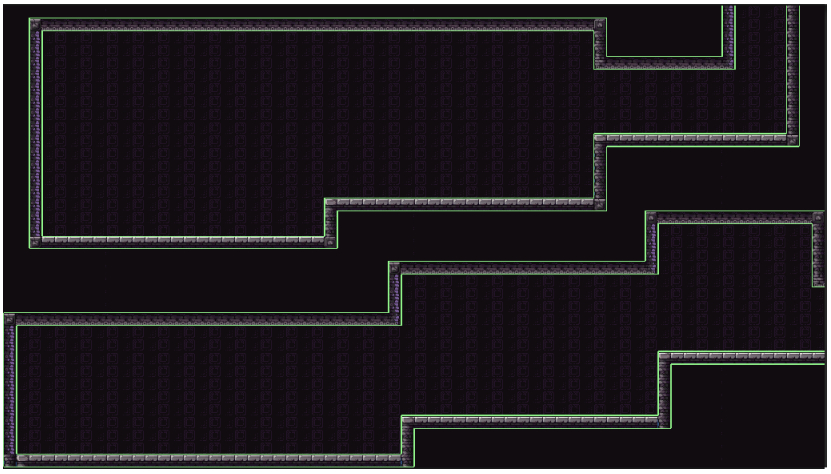

Figura 3.22: Estructura de la primera sala

Seguimos trabajando en el resto de salas, intentando crear un nivel coherente, en el que el jugador pueda desbloquear las habilidades necesarias para seguir avanzando.

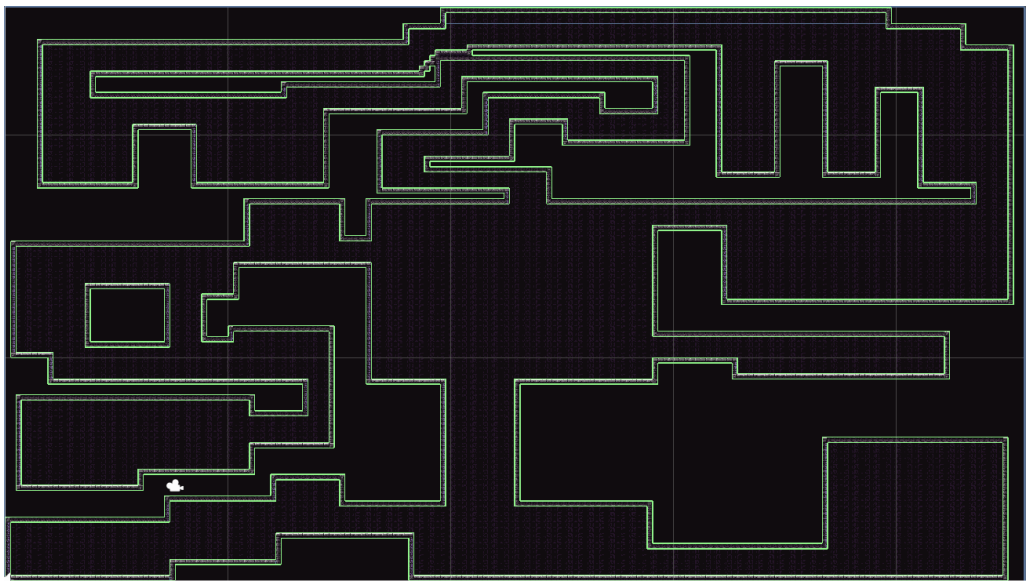

Figura 3.23: Estructura del nivel

Por último, debemos añadir al jugador, los enemigos y otros objetos interactuables del nivel. Muchos de estos elementos poseen comportamientos complejos, por lo que debemos configurarlos una vez los hayamos colocado.

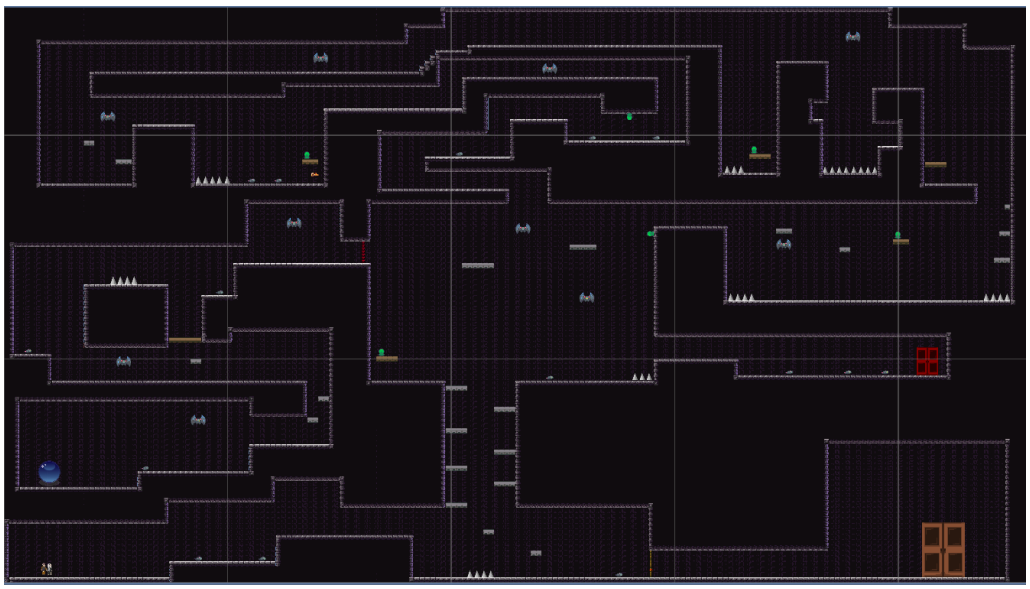

Figura 3.24: Nuestro nivel de Wizara completo

En la creación de este nivel, hemos podido comprobar que nuestro editor es capaz de ser utilizado con éxito en el desarrollo de videojuegos en dos dimensiones. En la figura 3.24 podemos apreciar el nivel recreado de *Wizara*, el cual es completamente jugable.

Capítulo 4

Conclusiones y Trabajo Futuro

4.1. Conclusiones

La idea del desarrollo de este Trabajo de Fin de Grado, como ya se ha mencionado anteriormente, surgió de los inconvenientes que implicaba usar Tiled como editor de niveles durante el desarrollo del videojuego *Fulvinter*. Por ello, nuestro tutor Ismael propuso la idea de crear un editor de niveles, con las funcionalidades indispensables de otros editores como Tiled, pero integrado en el propio proyecto de Unity. Aunque el objetivo principal es obtener un editor para *Fulvinter*, hemos dado especial importancia al hecho de que el editor sea compatible con otros videojuegos similares.

El resultado es una herramienta, en forma de paquete de Unity, para la edición de niveles de videojuegos 2D y 2.5D de *scroll* lateral. La herramienta puede ser fácilmente importada al proyecto de Unity de un juego de *scroll* lateral en fase de desarrollo, permitiendo un diseño y creación de niveles ágil y rápido, donde los niveles los conforman múltiples bloques y objetos ubicados sobre las baldosas de una cuadrícula.

La integración de este editor en los propios proyectos de Unity, además de acelerar el proceso en sí de construir el nivel, facilita el trabajo entre diferentes grupos del equipo de desarrollo de los videojuegos, como los diseñadores de niveles y artistas. Debido a que estos dos grupos pueden trabajar sobre el mismo proyecto, y al construir los niveles mediante bloques individuales, tendrán una mejor coordinación cuando surjan cambios inesperados. Por poner un ejemplo, durante el desarrollo de *Fulvinter*, cuando los diseñadores decidían cambiar parte de la lógica del nivel y rediseñarlo desde Tiled, los artistas, que ya habían adaptado el arte del nivel a la geometría o disposición anterior, debían modificar la malla 3D completa de la zona afectada. Haciendo uso de nuestra herramienta, y construyendo los niveles en base a bloques reutilizables, este problema se aborda de una manera más eficiente.

Además, consideramos que nuestra herramienta puede ser empleada en las clases sobre Diseño de Videojuegos que se imparten en la Facultad de Informática. Para justificar esta afirmación, expondremos nuestra experiencia personal vivida en

nuestro primer curso del grado de Desarrollo de Videojuegos.

Una de las prácticas que tuvimos que realizar durante la asignatura de Diseño de Videojuegos, perteneciente al primer curso del grado, consistía en diseñar una serie de niveles del estilo de los juegos de *Super Mario Bros*. Para ello, el guion de la práctica exigía utilizar un editor de niveles de Mario, parecido a *Super Mario Maker* pero desarrollado por la comunidad. Este editor, aunque era muy completo en cuanto a cantidad de bloques, objetos y enemigos, se distribuía como aplicación independiente. Pues bien, dado que el primer curso del grado se centra, en gran medida, en aprender a utilizar Unity, creemos que haber podido realizar esta práctica en el propio motor hubiera supuesto una gran ventaja. Con nuestra herramienta, los alumnos podrían ser capaces de diseñar niveles del estilo de los juegos de *Super Mario Bros* dentro del propio Unity. Esto focalizaría el aprendizaje de los alumnos en Unity —por supuesto, sin dejar de lado la enseñanza de diseño de niveles—, ayudando a familiarizarse aún más con el motor, y evitando sobrecargarles con el uso de herramientas innecesarias. Recordemos que la asignatura se imparte en el primer semestre del primer curso del grado, momento en el que los alumnos pueden verse abrumados fácilmente, pues se les presenta una gran cantidad de conceptos y herramientas nuevos. Por ello, creemos que el empleo de nuestro editor de niveles podría llegar a ser una buena opción.

A continuación, procedemos a documentar los objetivos de la herramienta conseguidos, especificados en la sección 1.2, describiendo el resultado obtenido:

- Debe proporcionar una cuadrícula 2D, formada por un número de casillas especificado por el usuario, donde se puedan colocar los diferentes elementos del nivel.

  Nuestro editor proporciona una cuadrícula o grid bidimensional, cuyas dimensiones pueden ser modificadas por el usuario, describiendo el número de tiles que la componen. Además, el tamaño de los tiles también puede personalizarse.

- Debe ofrecer las herramientas para colocar o pintar los objetos sobre la cuadrícula y para eliminarlos.

  Nuestro editor ofrece las herramientas de *Paint* y *Erase* para añadir objetos sobre los tiles de la cuadrícula y poder eliminarlos, respectivamente. Los objetos son ajustados al centro del tile sobre el que se encuentre el ratón en el momento del dibujado.

- El usuario debe poder crear diferentes capas sobre las que pintar en la cuadrícula, de manera que pueda organizar los objetos y aplicar ciertos cambios sobre ellos.

  Nuestra herramienta ofrece la posibilidad de crear múltiples capas, con nombres personalizados, donde se pueden ubicar los objetos del nivel de manera ordenada. Los objetos de cada capa, al compartir el mismo objeto padre, pueden ser tratados como una sola entidad y aplicarles cambios a todos ellos al mismo tiempo, como modificar sus posiciones u ocultarlos.

- Se debe poder modificar la profundidad de cada una de las capas en el eje perpendicular al plano de la cuadrícula. Esta función está principalmente dirigida a los elementos que componen el arte del nivel, pues el editor está enfocado en juegos con lógica 2D cuya acción transcurra en el plano de la cuadrícula.

  Desde la interfaz personalizada de nuestro editor, es posible especificar un valor número que represente la profundidad de las capas. Haciendo uso de dos botones con los símbolos de *más* (+) y *menos* (-), podemos desplazar cada capa, en dicha cantidad, en la dirección perpendicular al plano del grid, en ambos sentidos. Por supuesto, esta acción puede repetirse, permitiendo un mayor grado de personalización de la profundidad entre capas.

- El usuario debe disponer de una paleta con los diferentes objetos que pueda colocar en la cuadrícula, ordenados en categorías.

  Nuestra herramienta proporciona una paleta para ubicar los objetos. En la parte superior de ella aparece un botón para cada categoría. Al hacer *click* sobre uno de ellos, se nos mostrarán únicamente los objetos pertenecientes a esa categoría, justo debajo de dichos botones. Cada uno de estos objetos está contenido en un botón, el cual permite seleccionar el objeto que queremos ubicar sobre el grid. La paleta, al estar contenida en una ventana de Unity, es fácilmente reubicable dentro del editor de Unity, otorgando así flexibilidad a los usuarios.

- El usuario debe poder especificar qué objetos forman parte de la paleta y agruparlos en sus categorías de una manera sencilla.

  Gracias a nuestro sistema de lectura de *prefabs* (sección 3.1.2), el usuario puede crear tantas categorías como desee y añadir los objetos que pertenecen a cada una de ellas. Esto lo puede hacer de una manera sencilla e intuitiva, pues nuestro sistema aprovecha el sistema de ficheros de Unity para que el usuario ordene los objetos en subdirectorios que representan las categorías.

- El usuario debe poder modificar las propiedades de los objetos que se van a colocar sin tener que alterar su prefab.

  Mediante nuestro sistema de metadatos (sección 3.1.3), el usuario puede aplicar ciertos cambios sobre los objetos que se instanciarán posteriormente. Para ello, podrá especificar el valor de ciertos parámetros sobre cada *prefabs*, de manera que cada instancia de él, generada posteriormente, reflejará dichos cambios. Todo ello se realiza sin modificar el *prefab* original. Aunque los parámetros que se ofrecen por defecto son limitados, el usuario podrá ampliar esta lista, modificando ciertas partes localizadas de código.

- La herramienta debe poder ser utilizada sin necesidad de modificar código.

  La herramienta es perfectamente funcional sin necesidad de añadir o modificar código alguno. Sin embargo, durante su desarrollo hemos priorizado una buena estructura de código y seguido buenas prácticas al programar. De este modo, nuestro editor puede resultar relativamente sencillo de ampliar, permitiendo así

a los usuarios más experimentados añadir nuevas funcionalidades o modificar las existentes.

- La herramienta debe ser configurable para poderse adaptar a diferentes juegos.

  Como ya hemos comentado anteriormente, nuestra prioridad ha sido la compatibilidad de nuestro editor con el mayor número posible de juegos. Este logro hemos podido probarlo tras crear dos niveles completamente funcionales de dos juegos distintos.

Con todo esto, podemos concluir que nuestro editor de niveles proporciona las herramientas y funcionalidades indispensables para el diseño y creación de niveles de videojuegos de *scroll* lateral 2D y 2.5D. El hecho de poder integrarlo en el proyecto de Unity solventa los problemas que supone crear parte de los niveles en otros editores externos, favoreciendo la colaboración entre los diferentes equipos de trabajo. Aunque nuestra herramienta tiene margen de mejora, quedamos satisfechos con el trabajo realizado, los objetivos cumplidos y los resultados obtenidos.

4.2. Trabajo futuro

Aunque nuestro editor de niveles cumple con todos los requisitos que nos habíamos propuesto, tiene margen de mejora. Consideramos que, para poder distribuirlo como una herramienta para un uso, en cierto modo, profesional, deberían aplicarse algunas mejoras y adiciones.

Comenzaremos mencionando algunas limitaciones de la interfaz de usuario. Al utilizar la paleta podemos apreciar que, al seleccionar una categoría, su botón queda iluminado de un color diferente al resto. Sin embargo, este tipo de *feedback* no está presente en los botones de los objetos, lo que podría confundir al usuario. También nos hubiera gustado darle una apariencia más personalizada a la interfaz del editor, pues empleamos los colores y formas que ofrece Unity por defecto.

Respecto a los botones para crear y seleccionar capas, creemos que son bastante completos y cumplen perfectamente con su cometido. Sin embargo, echamos en falta la posibilidad de modificar el nombre de una capa ya creada. Aunque esto no tiene por qué suponer un gran problema, actualmente la única manera de corregir un nombre erróneo es eliminar la capa y crear otra con el nombre deseado. Si esa capa contiene muchos objetos, eliminarla podría ser un gran inconveniente.

También nos gustaría mencionar un replanteamiento en la forma en que almacenamos y recuperamos la información de los objetos de la escena, tal y como se explicó en las secciones 3.1.5 y 3.2.6.2. La idea consiste en almacenar en un componente específico de cada objeto intanciado toda la información necesaria para el editor, como su posición en el grid y su capa. Cuando cargamos la escena, el editor recuperará la información de cada objeto a partir de este componente, de una manera similar a la actual: es decir, leyendo cada objeto de la escena que sea hijo de *Scenary*. De esta manera, el editor tan solo deberá saber si cada posición del grid ha sido ocupada por algún objeto o no, siendo indiferent

Por otro lado, no hemos tenido tiempo suficiente para realizar pruebas con usuarios sobre el empleo de nuestra herramienta para la creación de niveles. Dado que el desarrollo del editor concluyó en el verano, las vacaciones no han favorecido a la disponibilidad de alumnos y/o profesores de la Facultad. Con más tiempo por delante, nos hubiera gustado diseñar un plan de pruebas para obtener *feedback* de diferentes usuarios. Creemos que de esta manera hubiese sido posible detectar más *bugs*, además de hipotéticas carencias en la usabilidad de la herramienta, como fallos en el diseño de la interfaz, o posibles ampliaciones con funcionalidades de otros editores similares que los usuarios echen en falta.

Introduction

Level design is one of the most labor-intensive tasks in video game development, especially in those types of games where the level itself is part of the core gameplay mechanics, such as platformers or metroidvanias —the popular name given to games whose mechanics are inspired by *Super Metroid* or *Castlevania: Symphony of the Night*. In this kind of game, having a dedicated editor is crucial to reducing development time and enabling rapid iteration of levels. In this context, our work presents ***Fulvinter Maker***, an editor for 2D/2.5D side-scrolling games, inspired by the editor used in the video game *Fulvinter*, currently in development, but integrated directly into the Unity game engine. Our editor allows the creation of games with the aforementioned characteristics more quickly than with the tools offered by Unity itself. Despite the name, the editor is designed to be used in any type of game and not exclusively for *Fulvinter*, although it draws upon the ideas that the game's development team considered necessary for their external editor.

Motivation

During the last two decades, the use of cross-platform commercial game engines has progressively gained popularity in the video game development scene. In 2024, 90% of the games released on the Steam platform were developed using engines of this category (Video Game Insights, 2025). This is due to several reasons. One of them is the rise of so-called independent or *indie* games (Obedkov, 2024), which are developed by small teams, usually with limited capital resources. Companies dedicated to this sector do not usually create their own game engine to develop their titles; instead, they choose alternatives such as Unity or Unreal Engine, commercial engines accessible to the public. As a result, the companies that own these engines have shifted their business model from pay-per-use to revenue-based payment, which democratizes access. Moreover, these engines offer multiple advantages: they reduce development time, allow independent studios to access technologies that would otherwise be too expensive for them to obtain on their own, and, after years of optimization and improvements, they provide a wide set of tools that facilitate their configuration and adaptation to the specific needs of each studio.

Another reason for the popularity of these engines is the increasing economic

cost of video game development. For this reason, it is a sound business strategy to release games on multiple platforms, such as Windows systems, game consoles, or mobile devices. These engines make this task easier, as they include agreements with various platforms to integrate their APIs and simplify cross-platform development.

However, although these engines are very versatile and include numerous tools for configuration, some types of games require very specific tools that facilitate their development. This is the case of platformers or metroidvanias, one of the most prolific genres of recent years. These kinds of games, highly dependent on level design, are prone to changes even in later stages of the *whitebox* design phase —a stage in which the game is prototyped without including graphical elements—, so creating levels using reusable blocks is advantageous when modifying them during the polishing stages. An editor that allows designers to place and adjust the elements and objects of the level in the cells of a large grid would greatly facilitate level design. This idea is supported by statements from Takashi Tezuka, designer of numerous Nintendo video games, including many 2D platformers in the *Super Mario Bros.* series. Tezuka stated that he wished he had a tool like *Super Mario Maker*, which fulfills the aforementioned characteristics, during the development of the first "Super Mario Bros." games (Webster, 2019). However, the default editors included in most engines fall short in this regard, making the development of such video games a tedious task.

This is the case of *Fulvinter*, a cooperative platformer inspired by Norse mythology and developed in Unity. This engine is highly versatile, but it does not provide tools to easily adjust basic tiles or blocks for level construction. These pieces can be called tiles or voxels if they correspond to purely 2D or 3D resources, respectively. The use of these units is widespread in 2D games, but less so in 3D ones. Nevertheless, in 2D logic games with 3D graphics, commonly referred to as 2.5D games, this type of functionality greatly simplifies both level editing and subsequent modifications. We can see this issue, and the advantage of a custom editor based on tiles or voxels, by looking at the video game *Super Mario Maker*, a level editor turned into a game that was released on Nintendo Wii U, 3DS, and, with its sequel, Nintendo Switch.

During the development of *Fulvinter*, external level editing tools such as Tiled were used due to the absence of similar functionalities in Unity. Using this program, levels were created by individually placing multiple objects in grid cells and organizing them into different layers. Afterwards, a file containing all the level information was generated, which then had to be imported into the Unity project to be processed by several custom scripts in order to generate the scene with all its elements. However, this translation process between the map created in Tiled and the exported game has multiple drawbacks. For instance, scenes must be re-exported whenever a level is edited in Tiled, and certain functionalities must be manually created for each specific game, as they are not easily generalizable.

This Bachelor's Thesis aims to address these issues and make the work of level designers easier by providing a tool integrated into the Unity environment itself, allowing level elements to be generated directly in the scene. In this way, the simplicity and convenience of editors like Tiled can be brought into the creation of 2.5D

games in Unity.

Objectives

The objective of this Bachelor's Thesis is to develop a tool, in the form of a Unity package, that can be imported into the project of a 2D side-scrolling game —whether featuring 2D or 3D graphics; that is, games whose visuals are based on sprites or 3D meshes— and that provides the essential utilities and conveniences of other level editors such as Tiled. In this way, the tool must meet the following requirements:

- It must provide a 2D grid, composed of a number of cells specified by the user, where different level elements can be placed.
- It must offer tools to place or paint objects on the grid and to remove them.
- The user must be able to create different layers on which to paint on the grid, so that objects can be organized and certain changes can be applied to them.
- It must be possible to modify the depth of each layer along the axis perpendicular to the grid plane. This feature is mainly intended for the elements that make up the level art, since the editor is focused on 2D logic games whose action takes place on the grid plane.
- The user must have a palette with the different objects that can be placed on the grid, organized into categories.
- The user must be able to specify which objects are included in the palette and group them into categories in a simple way.
- The user must be able to modify the properties of the objects to be placed without having to alter their prefab.
- The tool must be usable without requiring code modification.
- The tool must be configurable so it can be adapted to different games.

Work Plan

The following details the tasks carried out each month to achieve our objectives.

November 2024:

- Research the design of other level editors

December 2024:

- Research the Unity API for *Editor Scripting*

January 2025:

- Start development of the level editor tool
- Implementation of grid logic

February 2025:

- Implementation of paint and erase editing modes

March 2025:

- Serialization of instantiated objects to ensure persistence

April 2025:

- Grouping of palette objects into categories
- Writing object information to disk upon entering play mode

May 2025:

- Implement the ability to create multiple layers for painting

June 2025:

- Implement the ability to modify the depth of layers
- Object metadata system
- Begin writing the report

July 2025:

- Implement the system to add objects to the palette
- Implement the ability to adjust tile size
- Implement loading of already instantiated object information when opening the scene
- Report writing

August 2025:

- Bug fixing
- Recreate the *Fulvinter* and *Wizara* levels using our editor
- Report writing

Conclusions and Future Work

Conclusions

The idea for the development of this Bachelor's Thesis, as previously mentioned, arose from the inconveniences involved in using Tiled as a level editor during the development of the video game *Fulvinter*. For this reason, our supervisor Ismael suggested the idea of creating a level editor with the essential functionalities of other editors such as Tiled, but integrated directly into the Unity project. Although the main objective was to obtain an editor for *Fulvinter*, we placed particular importance on ensuring that the editor would also be compatible with other similar video games.

The result is a tool, in the form of a Unity package, for editing levels of 2D and 2.5D side-scrolling video games. The tool can be easily imported into the Unity project of a side-scrolling game under development, enabling fast and efficient level design and creation, where the levels consist of multiple blocks and objects placed on the tiles of a grid.

Integrating this editor into Unity projects, besides accelerating the process of building levels itself, facilitates collaboration among different groups in the video game development team, such as level designers and artists. Since these two groups can work on the same project, and because levels are built from individual blocks, they achieve better coordination when unexpected changes arise. For example, during the development of *Fulvinter*, when designers decided to change part of the level's logic and redesign it in Tiled, the artists, who had already adapted the level art to the previous geometry or layout, were forced to modify the entire 3D mesh of the affected area. By using our tool and constructing levels based on reusable blocks, this problem is addressed in a more efficient way.

Moreover, we believe that our tool could be used in Video Game Design classes taught at the Faculty of Computer Science. To justify this claim, we will present our personal experience from the first year of the Video Game Development degree.

One of the assignments we had to complete during the Video Game Design course, part of the first year of the degree, consisted of designing a series of levels in the style of *Super Mario Bros.* games. The assignment guidelines required the use of a Mario level editor, similar to *Super Mario Maker* but developed by the

community. This editor, although very complete in terms of blocks, objects, and enemies, was distributed as a standalone application. Since the first year of the degree is largely focused on learning Unity, we believe that being able to carry out this assignment directly within the engine would have been a great advantage. With our tool, students would be able to design levels in the style of *Super Mario Bros.* games inside Unity itself. This would concentrate the students' learning on Unity—without neglecting level design education—helping them become more familiar with the engine and avoiding the overload of using unnecessary tools. Let us remember that this course is taught in the first semester of the first year, a time when students can easily feel overwhelmed, as they are introduced to a large number of new concepts and tools. For this reason, we believe that using our level editor could be a good option.

Next, we document the objectives achieved by the tool, as specified in Section 1.2, describing the results obtained:

- It must provide a 2D grid, composed of a number of cells specified by the user, where different level elements can be placed.

  Our editor provides a two-dimensional grid whose dimensions can be modified by the user, defining the number of tiles it contains. In addition, the tile size can also be customized.

- It must offer tools to place or paint objects on the grid and to remove them.

  Our editor offers the *Paint* and *Erase* tools to add objects to the grid tiles and remove them, respectively. Objects are aligned to the center of the tile under the mouse cursor at the moment of placement.

- The user must be able to create different layers on which to paint on the grid, so that objects can be organized and certain changes can be applied to them.

  Our tool allows the creation of multiple layers with custom names, where objects can be organized within the level. The objects in each layer, sharing the same parent object, can be treated as a single entity, allowing collective changes to be applied to them, such as repositioning or hiding them.

- It must be possible to modify the depth of each layer along the axis perpendicular to the grid plane. This feature is mainly intended for elements that make up the level art, since the editor is focused on 2D logic games whose action takes place on the grid plane.

  Through the custom interface of our editor, it is possible to specify a numerical value representing the depth of the layers. By using two buttons with the *plus* (+) and *minus* (-) symbols, each layer can be moved by that amount along the axis perpendicular to the grid plane, in both directions. This action can be repeated, allowing a higher degree of customization of depth between layers.

- The user must have a palette with the different objects that can be placed on the grid, organized into categories.

Our tool provides a palette for placing objects. At the top of it, a button appears for each category. Clicking on one of them displays only the objects belonging to that category, just below the buttons. Each object is contained in a button that allows selecting the object to be placed on the grid. Since the palette is embedded in a Unity window, it can be easily repositioned within the Unity editor, thus granting flexibility to users.

- The user must be able to specify which objects are included in the palette and group them into categories in a simple way.

  Thanks to our *prefab* reading system (Section 3.1.2), the user can create as many categories as desired and add the objects belonging to each of them. This can be done in a simple and intuitive way, as our system leverages Unity's file system so that the user can organize objects into subdirectories representing the categories.

- The user must be able to modify the properties of the objects to be placed without having to alter their prefab.

  Through our metadata system (Section 3.1.3), the user can apply certain changes to the objects that will later be instantiated. For this, the user can specify the value of certain parameters for each *prefab*, so that each subsequent instance reflects those changes. All of this is done without modifying the original *prefab*. Although the default parameters are limited, the user can extend this list by modifying specific localized parts of the code.

- The tool must be usable without requiring code modification.

  The tool is fully functional without the need to add or modify any code. However, during its development we prioritized a good code structure and followed programming best practices. This way, our editor can be relatively easy to extend, allowing more experienced users to add new functionalities or modify existing ones.

- The tool must be configurable so it can be adapted to different games.

  As already mentioned, our priority was the compatibility of our editor with as many games as possible. We were able to validate this achievement by creating two fully functional levels for two different games.

With all this, we can conclude that our level editor provides the essential tools and functionalities for the design and creation of levels for 2D and 2.5D side-scrolling video games. The ability to integrate it into Unity projects solves the problems associated with creating parts of the levels in external editors, fostering collaboration among the different teams. Although our tool still has room for improvement, we are satisfied with the work done, the objectives achieved, and the results obtained.

Future Work

Although our level editor meets all the requirements we set for ourselves, it still has room for improvement. We believe that, in order to distribute it as a tool for somewhat professional use, certain improvements and additions should be applied.

We will begin by mentioning some limitations of the user interface. When using the palette, we can see that when selecting a category, its button is highlighted in a different color than the rest. However, this type of *feedback* is not present in the object buttons, which could confuse the user. We also would have liked to give the editor's interface a more personalized appearance, since we relied on Unity's default colors and shapes.

Regarding the buttons for creating and selecting layers, we believe they are quite complete and perfectly fulfill their purpose. However, we miss the possibility of renaming an already created layer. Although this may not pose a major problem, at present the only way to correct an incorrect name is to delete the layer and create another one with the desired name. If that layer contains many objects, deleting it could be a significant inconvenience.

We would also like to mention a reconsideration of the way we store and retrieve information about the objects in the scene, as explained in Sections 3.1.5 and 3.2.6.2. The idea consists of storing, in a specific component of each instantiated object, all the information necessary for the editor, such as its position in the grid and its layer. When loading the scene, the editor would retrieve the information from each object through this component, in a way similar to the current one—that is, by reading each object in the scene that is a child of *Scenary*. In this way, the editor would only need to know whether each position in the grid has been occupied by an object or not, regardless of the rest.

On the other hand, we did not have enough time to carry out user testing regarding the use of our tool for level creation. Since the development of the editor concluded in the summer, the vacation period did not favor the availability of students and/or professors from the Faculty. With more time, we would have liked to design a testing plan to obtain *feedback* from different users. We believe that in this way it would have been possible to detect more *bugs*, as well as hypothetical shortcomings in the usability of the tool, such as flaws in the interface design or possible extensions with functionalities from other similar editors that users might miss.

Contribuciones Personales

David Casiano Flores

David ha contribuido adecuadamente durante el desarrollo del proyecto. Estudió en profundidad el libro *Extending Unity with Editor Scripting* (Tadres, 2015), extrayendo la información que podría ser de utilidad para el desarrollo del editor de niveles.

Comenzó familiarizándose con la API para escribir código del editor de Unity, buscando la documentación pertinente y ayudándose del libro citado anteriormente.

David implementó la cuadrícula o grid del editor y creó una primera versión de la paleta, con sus funcionalidades básicas. Esta paleta incluía la funcionalidad de mostrar objetos en ella, pudiendo ser *clickados* para seleccionarlos.

Implementó la base del script *EditorManager*, el cual se encarga de almacenar la información necesaria sobre los objetos instanciados en el grid, asegurándose de que un único objeto ocupase cada tile. También creó una primera herramienta de pintado, de manera que pudiera probarse la instanciación de objetos sobre el grid.

Implementó las funciones auxiliares para realizar cálculos sobre posiciones en el grid. También creó una vista de inspector personalizada para el objeto el cual contiene el grid, en el cual podía configurarse algunos parámetros como el número de filas y columnas del grid.

David implementó el botón el cual crea una nueva escena de Unity que incluya todos los elementos necesarios para empezar a crear un nivel usando nuestro editor. Más adelante, refactorizó el código de algunos scripts para seguir una estructura más ordenada y amable con futuras ampliaciones.

También comenzó con la persistencia del nivel creado, guardando en un *ScriptableObject* la información de los objetos instanciados necesaria para recuperarlos cuando salimos del modo editor para entrar en el modo de ejecución del juego, y viceversa. Por lo tanto, también añadió al script *EditorManager* la funcionalidad para escribir y leer la información en los *ScriptableObjects*.

Implementó toda la lógica del editor para gestionar diferentes capas para pintar. De este modo, adaptó el código existente para que fuese compatible con el sistema

multicapa, permitiendo así añadir y eliminar objetos de cada una de ellas. También añadió elementos al inspector del grid para poder añadir y seleccionar una capa determinada. Implementó la posibilidad de nombrar a las capas nuevas con el nombre que el usuario desee.

Más adelante, implementó la posibilidad de modificar la profundidad de las capas, reposicionando cada objeto contenido en la capa que se haya desplazado. De la mano de esto, añadió al inspector del grid elementos para especificar la cantidad en la que se desea desplazar la capa, así como los botones para desplazarla en ambos sentidos.

David también amplió la funcionalidad de escritura en *ScriptableObjects* de la información necesaria para asegurar su persistencia, de manera que sea compatible con el sistema multicapa.

Por último, en cuanto a en lo que a implementación de la herramienta se refiere, implementó la opción para redimensionar las baldosas o tiles del grid. Debido a esto, modificó la lógica necesaria para reposicionar cada objeto ya instanciado a su nueva ubicación, fruto de una redimensión del tamaño de los tiles.

En cuanto a la redacción de esta memoria, David comenzó escribiendo el resumen del proyecto, para continuar con el capítulo 1 de Introducción. Escribió en su totalidad el capítulo 2 de Estado de la Cuestión. Para ello, realizó un extenso estudio sobre las opciones de edición de niveles presentes en el mercado, como Tiled y el Tilemap de Unity. También se documentó sobre el uso de *Super Mario Maker* y *Mega Man Maker* para crear niveles de los juegos de dichas franquicias. Realizó un estudio sobre su funcionamiento, así como de su interfaz, destacando los aspectos de mayor interés para el desarrollo de nuestra herramienta. También buscó otros proyectos similares al nuestro realizados con anterioridad, describiendo el problema que pretendían resolver y analizando sus puntos fuertes y débiles.

David también redactó la sección 3.1 sobre Diseño del capítulo Descripción del Trabajo. En ella, describió cómo debía comportarse nuestra herramienta en cuanto a las funcionalidades que queríamos que tuviera. De este modo, explicó también las decisiones de diseño tomadas.

Por último, David concluyó con gran parte de la redacción del capítulo 4 sobre las Conclusiones y Trabajo Futuro, recapitulando los objetivos planteados y conseguidos y sintetizando los puntos fuertes y puntos débiles de nuestro editor de niveles. También detalló algunos aspectos incompletos o no realizados sobre el proyecto, en gran medida debido a la falta de tiempo, y propuso algunas mejoras de cara a posibles ampliaciones futuras de la herramienta.

Sol Flora López Antón

Sol Flora contribuyó al proyecto en gran medida durante toda su duración. Investigó para encontrar documentación sobre proyectos similares y distinguir cuáles podrían ser útiles para el desarrollo de la herramienta.

Junto a David, implementó la lógica de pintado y borrado de objetos en la

escena, siguiendo el libro *Extending Unity with Editor Scripting* (Tadres, 2015). Creó el sistema que convierte pulsaciones del ratón en coordenadas del editor y la posibilidad de borrar objetos.

Después, se dispuso a trabajar en la permanencia, creando una base que se desarrollaría más adelante.

Creó la funcionalidad de la ventana de paleta, dividiendo los objetos en categorías y encontrando la manera de mostrarlos en pantalla, de manera que el usuario pudiera seleccionarlos. Una vez que las categorías funcionaban, se encargó de que el usuario pudiera moverse entre ellas al representarlas como pestañas de la ventana.

Más adelante, Sol Flora creó el sistema de lectura de archivos, primero para comprobar cuándo una categoría u objeto era añadido, añadiendo el objeto a la paleta y creando la categoría correspondiente, y luego extendiéndolo para el caso de que un elemento fuera eliminado, en cuyo caso este debía ser quitado de la paleta.

A partir de este sistema, Sol Flora se dedicó a crear un sistema de guardado de metadatos, creando las clases correspondientes para facilitar el almacenamiento de los datos correspondientes como ***ScriptableObjects***, extendiendo la lectura de archivos para automatizar este proceso de guardado e implementando las funcionalidades necesarias para convertir objetos en metadatos y viceversa.

Tras esto, mejoró la paleta para conseguir que esta se actualizase en tiempo real tras la creación o destrucción de un elemento en el sistema de archivos. También implementó un botón de actualización de la paleta en el caso de que se hubieran cambiado los metadatos de un objeto.

A continuación se encargó de buscar una solución al problema de la persistencia fuera de la ejecución, probando varias implementaciones y examinando la causa del error. Dedujo correctamente la causa, la eliminación de las referencias en memoria cuando Unity cambia de escena, y encontró una implementación funcional, confiando en la persistencia de Unity.

Una vez todas las funcionalidades habían sido implementadas, Sol Flora se dedicó a probar la herramienta y a arreglar los errores que pudo encontrar, principalmente en la carga de metadatos.

Tras esto, se encargó de convertir la herramienta en un paquete de Unity a partir del proyecto en el que se estaba trabajando.

Sol Flora fue también la encargada de probar la herramienta, creando los niveles de Fulvinter y Wizara. Para el nivel de Fulvinter, examinó los niveles originales para encontrar un buen candidato a recrear. Después, identificó los objetos necesarios para el desarrollo del nivel y trabajó en todo el proceso de creación de este, creando las capas necesarias y pintando los objetos de acuerdo al original.

Para el nivel de Wizara, analizó los niveles originales para diseñar un nivel completamente nuevo desde cero. Una vez que este diseño había sido creado, se dispuso a pintarlo con el editor.

Mientras creaba estos niveles, observo los fallos del editor que surgían durante un uso extendido y trabajo en arreglarlos.

Durante la redacción de esta memoria, los esfuerzos de Sol Flora se centraron principalmente en el capítulo Descripción del Trabajo. Identificó la necesidad de dividir el capítulo en sus secciones de Diseño 3.1, Implementación 3.2 y Uso del Editor 3.2.7.

Después, se centró en la sección de Implementación 3.2, dividiéndola en las funcionalidades correspondientes y repasando el código para realizar una descripción detallada del desarrollo del editor. Creó también el Diagrama de Clases 3.2.1, teniendo en cuenta la relación entre las clases del proyecto y su agrupación en distintos sistemas.

Por último, Sol Flora describió su proceso a la hora de crear niveles para Fulvinter y Wizara en la sección 3.2.7, exponiendo así los resultados obtenidos con la herramienta y el posible flujo de trabajo que nuestro usuario puede esperar.

Bibliografía

The Abbey of Crime Extensum [Recuperado el 9 de septiembre de 2025]. (2016). https://store.steampowered.com/app/474030/The_Abbey_of_Crime_Extensum/

Cameron, S. H. (2024). *Unity 2022 by Example: A project-based guide to building 2D and 3D games, enhanced for AR, VR, and MR experiences*. Packt Publishing Ltd.

da Silva Beserra, I. N. (2015). TEd2D: Um editor para criação de cenários de jogos 2D com Unity.

Enter Unity Play Mode Details [Recuperado el 9 de septiembre de 2025]. (2025). https://docs.unity3d.com/6000.1/Documentation/Manual/configurable-enter-play-mode-details.html

Game UI Database – Super Mario Maker 2 [Recuperado el 9 de septiembre de 2025]. (2025). https://www.gameuidatabase.com/gameData.php?id=27

GameObject Brush (2D Tilemap Extras Package Manual) [Recuperado el 9 de septiembre de 2025]. (2023). https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/GameObjectBrush.html

Introduction — Tiled Manual [Recuperado el 9 de septiembre de 2025]. (2023). https://doc.mapeditor.org/en/stable/manual/introduction/

Introduction to Tilemaps [Recuperado el 9 de septiembre de 2025]. (2025). https://learn.unity.com/tutorial/introduction-to-tilemaps

La abadía del crimen [Recuperado el 9 de septiembre de 2025]. (1987). https://es.wikipedia.org/wiki/La_abad%C3%ADa_del_crimen

Lien, T. (2014). *Mario Maker started out as a tool for Nintendo's developers* [Recuperado el 9 de septiembre de 2025]. https://www.polygon.com/2014/6/13/5805472/mario-maker-started-out-as-a-tool-for-nintendos-developers/

List of best-selling Nintendo Switch video games [Recuperado el 9 de septiembre de 2025]. (2025). https://en.wikipedia.org/wiki/List_of_best-selling_Nintendo_Switch_video_games

Nasution, S., Nasution, A. H., & Hakim, A. L. (2020). Tile-based game plugin for unity engine.

Obedkov, E. (2024). *Indie games come close to AA/AAA games in revenue on Steam for first time ever, grossing $4 billion in 2024 so far* [Recuperado el 9

de septiembre de 2025]. https://gameworldobserver.com/2024/10/16/indie-games-revenue-steam-vs-aaa-titles-vg-insights

Pérez Colado, I. J., & Pérez Colado, V. M. (2014). Un conjunto de herramientas para Unity orientado al desarrollo de videojuegos de acción-aventura y estilo retro con gráficos isométricos 3D.

Play Guide — Super Mario Maker 2 Wiki [Recuperado el 9 de septiembre de 2025]. (2019). https://supermariomaker2.fandom.com/wiki/Play_Guide

Santamaría Barcina, A., & Alexiades Estarriol, A. (2015). Reconstrucción de una video-aventura isométrica clásica utilizando IsoUnity como herramienta de producción multiplataforma.

Super Mario Maker 2 – Critic Reviews [Recuperado el 9 de septiembre de 2025]. (2019). https://www.metacritic.com/game/super-mario-maker-2/critic-reviews/?platform=nintendo-switch

Tadres, A. (2015). *Extending Unity with Editor Scripting*. Packt Publishing Ltd.

Unity Domain Reloading [Recuperado el 9 de septiembre de 2025]. (2023). https://docs.unity3d.com/2022.2/Documentation/Manual/DomainReloading.html

Unity InstanceID [Recuperado el 9 de septiembre de 2025]. (2025). https://docs.unity3d.com/ScriptReference/Object.GetInstanceID.html

Video Game Insights. (2025). *The Big Game Engines Report of 2025* [Recuperado el 9 de septiembre de 2025]. https://vginsights.com/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf

Webster, A. (2019). *One of Nintendo's top designers says he always wanted a tool like Super Mario Maker* [Recuperado el 9 de septiembre de 2025]. https://www.theverge.com/2019/6/28/18759918/super-mario-maker-2-nintendo-switch-takashi-tezuka-interview