

Trabajo fin de grado

Desarrollo de una extensión de Unity para el soporte
a videojuegos roguelike



Alejandro Pascual Pozo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**Desarrollo de una extensión de Unity para el
soporte a videojuegos roguelike**

Autor: Alejandro Pascual Pozo

Tutor: Carlos Aguirre Maeso

junio 2021

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 15 de Junio de 2021 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Alejandro Pascual Pozo
Desarrollo de una extensión de Unity para el soporte a videojuegos roguelike

Alejandro Pascual Pozo
Av. Monasterio de Silos Nº 44

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi padre y a mi madre

*El noventa por ciento de lo que consideramos imposible es, de hecho, posible;
el otro diez por ciento lo será con el paso del tiempo y el avance tecnológico.*

Hideo Kojima

AGRADECIMIENTOS

Me gustaría agradecer a todas las personas que me han brindado su apoyo durante mis estudios y que me han motivado siempre a seguir adelante. Mis padres, mi hermana y mis amigos tanto dentro como fuera de la facultad han sido un pilar firme para mí durante todos estos años.

También quiero destacar a tantos desconocidos que comparten su conocimiento desinteresadamente y que me han permitido explorar mi interés por el desarrollo de videojuegos de manera autodidacta, así como a los docentes que comparten esta pasión y me han brindado su ayuda y experiencia.

RESUMEN

De unos años a esta parte, la industria del videojuego ha vivido una tendencia hacia el uso cada vez más extendido de motores de juego genéricos y flexibles. Estos motores están a cargo de equipos especializados y los desarrolladores de videojuegos adquieren licencias para usarlos en sus propios proyectos. Los dos grandes referentes de este campo son Unity, que goza de una enorme popularidad entre desarrolladores independientes y equipos pequeños y medianos, y Unreal Engine, que sigue siendo la opción más común en proyectos de alto presupuesto.

La motivación para la creación de esta extensión viene principalmente del deseo de combinar el uso de un motor de juego moderno como Unity con la arquitectura de los roguelikes clásicos. El género roguelike en su forma clásica se desarrolló principalmente en los 80 y los 90, funcionando sobre la terminal con una representación en ASCII. Las enormes diferencias con los entornos de desarrollo y ejecución actuales provocan que sea un reto imitar algunas de las características de estos clásicos sin renunciar a los elementos que consideramos indispensables en un videojuego moderno. Además, el enfoque genérico de Unity carece de soporte base para varias de las características fundamentales del género, como es el uso de un tablero o un esquema un tanto peculiar de turnos.

A lo largo de este documento estudiaremos qué caracteriza a los videojuegos de este género, qué elementos modernos podemos incorporar y cuáles son las dificultades que surgen al hacerlo. Utilizaremos el sistema de extensiones de Unity para empaquetar toda esta funcionalidad en un módulo fácil de compartir e instalar. La extensión desarrollada tendrá como objetivos modificar la arquitectura de Unity para adaptarla a las necesidades del género roguelike, proveer diferentes herramientas para facilitar y agilizar el desarrollo de estos juegos y permitir la incorporación de los elementos necesarios en el desarrollo de un título moderno.

PALABRAS CLAVE

Motor de juego, Unity, extensión, herramientas, videojuegos, roguelike, modernización

ABSTRACT

Over the past decades, the video game industry has experienced a trend towards the increasingly widespread use of generic and flexible game engines. These engines are developed by specialized teams and game developers acquire licenses to use them in their own projects. The two big names in this field are Unity, which is hugely popular among independent developers and small and medium-sized teams, and Unreal Engine, which remains the most common choice for high-budget projects.

The motivation for creating this extension comes mainly from the desire to combine the use of a modern game engine like Unity with the architecture of classic roguelikes. The roguelike genre in its classic form was developed mainly in the eighties and nineties, running on the terminal with an ASCII representation. The enormous differences with the current development and runtime environments make it a challenge to imitate some of the characteristics of these classics without giving up the elements that we consider essential in a modern video game. Furthermore, Unity's generic approach lacks base support for several of the fundamental characteristics of the genre, such as the use of a board or a somewhat peculiar turn scheme.

Throughout this document we will study what characterizes video games of this genre, what modern elements we can incorporate to them and what are the difficulties that arise when doing so. We will use the Unity extension system to package all this functionality in one easy to share and install module. The extension developed will aim to modify the architecture of Unity to adapt it to the needs of the roguelike genre, provide different tools to facilitate and speed up the development of these games and allow the incorporation of the necessary elements in the development of a modern title.

KEYWORDS

Game engine, Unity, extension, tools, video games, roguelike, modernization

ÍNDICE

1 Introducción	1
1.1 Motivación	1
1.2 Objetivos del trabajo	2
1.3 Estructura del documento	3
2 Estado del arte	5
2.1 El género roguelike	5
2.1.1 Orígenes y características iniciales	5
2.1.2 Evolución y hegemonía actual	6
2.1.3 Retorno de las características clásicas	7
2.2 Motores de juego	8
2.3 Desarrollo en Unity	11
2.3.1 Plataformas soportadas	11
2.3.2 Características principales	11
2.3.3 Recursos disponibles	12
2.3.4 Dificultades previstas	12
2.4 Sistema de extensiones de Unity	13
2.4.1 Casos de éxito	13
2.4.2 Empaquetamiento y distribución	14
3 Diseño	15
3.1 Objetivos de la extensión	15
3.2 Requisitos funcionales	16
3.3 Requisitos no funcionales	19
3.4 Arquitectura, módulos y clases	20
3.4.1 Arquitectura y dependencias	20
3.4.2 Módulos	22
3.5 Tecnologías y estándares	27
4 Implementación	29
4.1 Entorno de desarrollo	29
4.2 Implementación de los módulos	29
4.3 Empaquetamiento y registro de la extensión	35

5 Pruebas	37
5.1 Tests unitarios	37
5.2 Tests de integración	38
5.3 Videojuego implementado con la extensión	38
6 Conclusiones y trabajo futuro	41
6.1 Conclusiones	41
6.2 Trabajo futuro	42
Bibliografía	43
Apéndices	45
A Fragmentos de interés del código	47
B Capturas del videojuego implementado con la extensión	51

LISTAS

Lista de códigos

A.1	Función Update del módulo Core	47
A.2	Función OnMouseLeftClick del módulo Input	47
A.3	Función GameLoop del módulo Game.Core	47
A.4	Función ProcessTurns del módulo Game.Core	48
A.5	Funciones Refresh y HandleCtrlMouseLeftClick del módulo UI	48
A.6	Función Process del módulo Game.Actions	49

Lista de figuras

1.1	Editor visual de shaders de Unity	2
2.1	Captura de Rogue	6
2.2	Captura de Stoneshard	7
2.3	Plataformas publicitadas por Unity	11
2.4	Bolt en la Asset Store de Unity	13
2.5	uMMORPG en la Asset Store de Unity	14
3.1	Diagrama de módulos y dependencias	20
3.2	Diagrama de secuencia de un fotograma en la arquitectura de Unity	21
3.3	Diagrama de clases del módulo Core	22
3.4	Diagrama de clases del módulo Input	22
3.5	Diagrama de clases del módulo UI	23
3.6	Diagrama de clases del módulo Game.Core	23
3.7	Diagrama de clases del módulo Game.Turns	23
3.8	Diagrama de clases del módulo Game.Generation	24
3.9	Diagrama de clases del módulo Game.Controllers	24
3.10	Diagrama de clases del módulo Game.Actions	25
3.11	Diagrama de clases del módulo Game.Boads	25
3.12	Diagrama de clases del módulo Game.Entities	26
3.13	Diagrama de clases del módulo Utils	26
3.14	Tecnologías utilizadas	27
4.1	Validador de extensiones de Unity	36

4.2	Borrador de la extensión en Unity Asset Store Publisher	36
5.1	Tests del módulo Game.Borads ejecutándose en el framework de Unity	37
B.1	Intercalado de turnos 1	51
B.2	Intercalado de turnos 2	51
B.3	Interfaz de usuario 1	52
B.4	Interfaz de usuario 2	52
B.5	Eliminación de un agente 1	53
B.6	Eliminación de un agente 2	53
B.7	Agentes de distintas velocidades 1	54
B.8	Agentes de distintas velocidades 2	54
B.9	Scriptable objects	55
B.10	Módulo Tilemap de Unity	55
B.11	Primer paso de la generación de mazmorras	56
B.12	Segundo paso de la generación de mazmorras	56
B.13	Tercer paso de la generación de mazmorras	57
B.14	Cuarto paso de la generación de mazmorras	57

INTRODUCCIÓN

A continuación, veremos la motivación para la realización de este trabajo, explorando los orígenes de uno de los géneros de videojuegos más antiguos y analizando las posibilidades que ofrece la industria en la actualidad. Visto esto, concretaremos los objetivos que persigue este proyecto y, por último, presentaremos la estructura de este documento.

1.1. Motivación

La industria del videojuego es extremadamente joven si la comparamos con otros medios de entretenimiento más tradicionales. Sin embargo, existen pocos ejemplos comparables a la rápida transformación y evolución que ha vivido durante sus décadas de existencia. Esto se debe principalmente a su fuerte vinculación con la tecnología y la computación, que han supuesto una revolución en tiempos recientes.

En este sentido, es interesante volver a las raíces de un género que vivió su época dorada tras la llegada de los primeros ordenadores personales. Los roguelikes, también conocidos en español como videojuegos de mazmorra, comenzaron a ser producidos a finales de los setenta por aficionados a los ordenadores y a la programación. Fueron la evolución de los juegos basados puramente en texto y, en ellos, se intentaron replicar mecánicas propias de los juegos de mesa populares en esos años. Su entorno de desarrollo y ejecución fue tan diferente al actual que muchas de sus características son todo un reto de adaptar a tiempos recientes. Las expectativas del público actual también incluyen muchos elementos que no estaban presentes en aquellos títulos, y que requerirían de cambios profundos en su diseño y arquitectura.

Mucho ha cambiado en la industria del videojuego en los últimos 40 años, y existen ahora muchos beneficios al utilizar motores de juego desarrollados por equipos especializados. Si el desarrollo de un título del género superase las barreras iniciales de adaptación a estas soluciones generalistas, ganaría acceso a infinidad de herramientas para agilizar y potenciar su desarrollo: gráficos 2D y 3D de alta calidad, motores de físicas realistas, editores de animaciones, editores de shaders, sistemas de partículas, compatibilidad en multitud de plataformas y un largo etcétera. Estos motores también

cuentan con un sólido sistema de extensiones, que permiten ampliar y modificar su funcionalidad en apenas unos clics, abriendo un abanico aún más extenso de posibilidades.

Haciendo un uso adecuado de este sistema de extensiones, no solo es posible adaptar la arquitectura de dichos motores para eliminar los problemas más evidentes que surgen al desarrollar un roguelike en ellos, sino que además se puede proveer un conjunto de herramientas que faciliten enormemente la implementación de las mecánicas más comunes del género. Sería muy interesante lograr preservar la relativa sencillez que caracterizó al desarrollo de estos títulos, al menos comparados con otras producciones profesionales actuales, sin necesidad de abandonar las herramientas modernas; ya que es esta sencillez la que ha matenido vivas comunidades de desarrolladores hasta la actualidad [1].

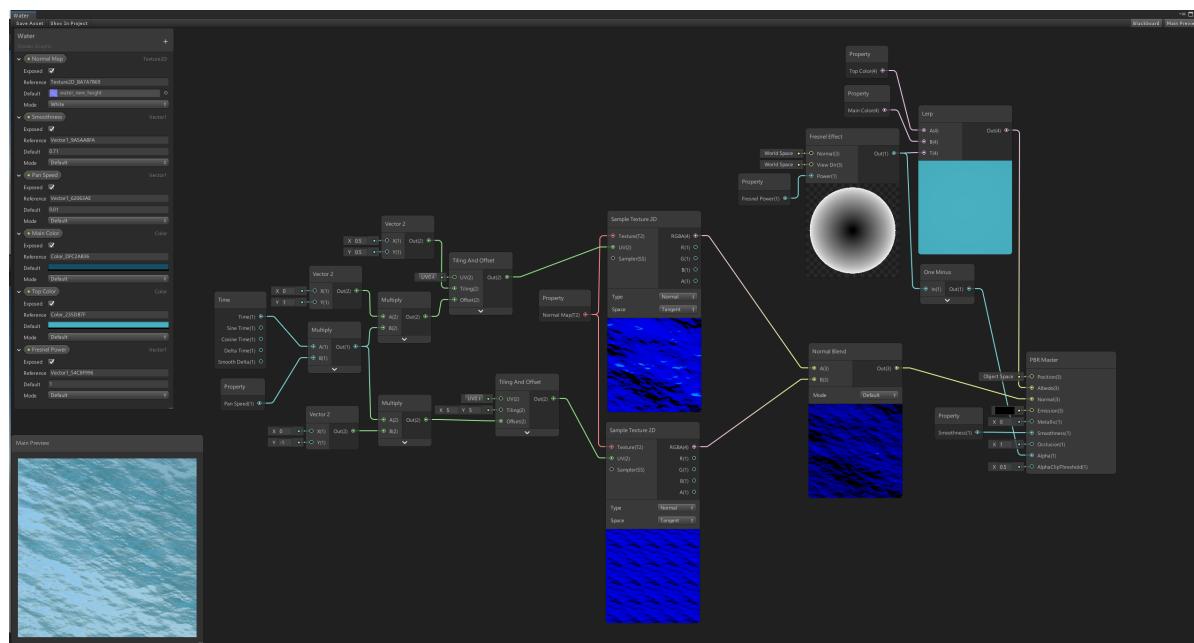


Figura 1.1: Shader de olas en el moderno editor visual de shaders de Unity, creado por nodos.

1.2. Objetivos del trabajo

Los principales objetivos que motivan el desarrollo de este trabajo son los siguientes:

- Permitir adaptar a las tecnologías y flujos de trabajo modernos un género cuyas características se definieron en base a un contexto de hace más de 40 años.
- Explorar las mecánicas principales del roguelike y crear sistemas potentes pero flexibles que faciliten implementar de manera sencilla dichas mecánicas en nuevos proyectos.
- Diseñar una arquitectura bien estructurada que permita incorporar módulos de funcionalidad completamente desacoplados entre sí y de los datos, agilizando el desarrollo y prototipado.
- Utilizar el sistema de extensiones de Unity, un motor de juego moderno, para empaquetar toda esta funcionalidad en una extensión fácil de incorporar al entorno de desarrollo.

1.3. Estructura del documento

Con el objetivo de contextualizar la lectura de cada sección, se proporciona a continuación la estructura que sigue el cuerpo de este documento:

- **Estado del arte.** Empezaremos viendo los títulos recientes más relevantes del género, estudiando sus diferencias con las entregas originales. Analizaremos las distintas alternativas existentes en la actualidad para el uso de un motor de juego, comparando sus ventajas y desventajas y justificando la elección de Unity. También repasaremos el funcionamiento del sistema de extensiones de dicho motor, sobre el cual se desarrollará este proyecto.
- **Diseño.** Estableceremos los objetivos clave en el desarrollo de la extensión y definiremos sus requisitos funcionales y no funcionales, así como las tecnologías y estándares a utilizar. Especificaremos la arquitectura, concretando el diseño de sus módulos y clases y valiéndonos de diversos diagramas UML para representar su estructura de manera intuitiva y visual.
- **Implementación.** Analizaremos el entorno de desarrollo utilizado, repasando la implementación de cada módulo y los retos, soluciones y particularidades encontrados en cada uno de ellos. También veremos la puesta en práctica del empaquetamiento en una extensión de Unity del software desarrollado.
- **Pruebas.** Mediante el uso de pruebas unitarias y de integración, comprobaremos el correcto funcionamiento de la extensión. Además, desarrollaremos un pequeño videojuego que nos permitirá ver en acción todos los sistemas implementados.
- **Conclusiones y trabajo futuro.** Para finalizar, estudiando los resultados obtendremos las conclusiones del proyecto y estableceremos los objetivos que se deberían perseguir para mejorar la extensión mediante trabajo futuro.

ESTADO DEL ARTE

En este capítulo, exploraremos los orígenes y el estado actual del género roguelike. Estudiaremos también las alternativas que ofrece la industria en cuanto a la elección de un motor de juego y veremos en detalle el funcionamiento de Unity y de su sistema de extensiones.

2.1. El género roguelike

2.1.1. Orígenes y características iniciales

El término roguelike surgió a partir del desarrollo de Rogue, videojuego publicado en 1980. El verdadero primer título del género, Beneath Apple Manor, llegó en realidad dos años antes, en 1978. En cualquier caso, los roguelike clásicos se desarrollaron principalmente durante los años ochenta y noventa tras la popularización de los ordenadores personales, si bien ya a finales de los setenta se programaban y jugaban en mainframes universitarios y los primeros modelos de ordenadores personales. La temática ambiciosa de estos juegos, que permitían explorar amplios mundos, combinada con la dificultad de producir gráficos por ordenador, provocó que se desarrollasen principalmente como aplicaciones de terminal. Éstas representaban los diferentes elementos mediante el uso de caracteres de texto, aunque también existían alternativas con gráficos sencillos en cuadrículas.

Las limitaciones de este entorno de ejecución definieron a su vez algunas de las características principales del género, como el movimiento sobre un tablero cuadriculado y la carencia de animaciones para la mayoría de las acciones, que suponía un flujo ágil incluso cuando había una gran cantidad de elementos interactuando entre sí. Sobre estas características, los autores terminaron por construir un género con mecánicas claramente inspiradas en juegos de mesa como Dragones y Mazmorras, utilizando un esquema de turnos y girando en torno a la exploración de mazmorras generadas proceduralmente, es decir, mediante el uso de algoritmos.

Esta última característica tiene su origen en una limitación técnica: la escasa memoria disponible. No era posible almacenar una gran cantidad de mapas de mazmorras, por lo que se desarrollaron algoritmos para su generación automática al momento de ser necesitados. Pronto se descubrió el enorme

potencial que ofrece este planteamiento, ya que permite volver a jugar al mismo juego, pero con contenido completamente nuevo en cada partida. Para potenciar esta mecánica se incorporó rápidamente la muerte permanente, que se convertiría en otro de los principales rasgos del juego al incentivar la rejugabilidad y la valoración de cada partida y personaje como únicos y volátiles.

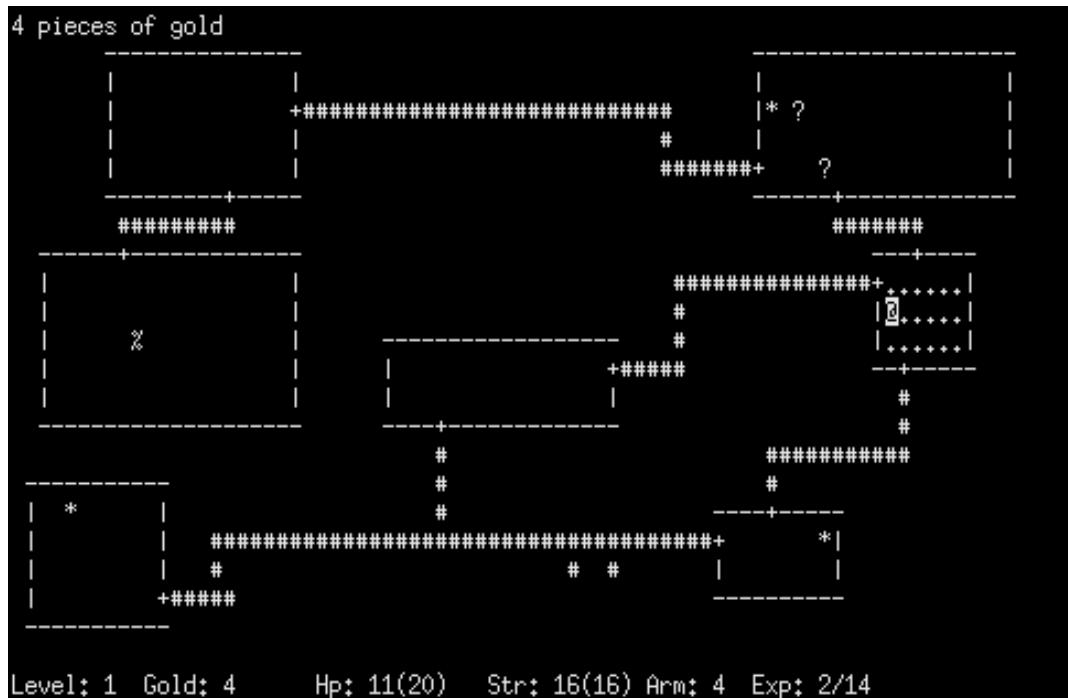


Figura 2.1: Captura de Rogue (1978).

2.1.2. Evolución y hegemonía actual

Con el paso de los años y la evolución de la tecnología, los títulos fueron tomando diferentes rutas en cuanto a las innovaciones que aportaban. Se produjeron una gran cantidad de juegos de estrategia que, si bien compartían el uso de cuadrículas y turnos, renunciaban a la exploración, el contenido procedimental y la limitación de un único personaje; por lo que no son catalogados dentro del género roguelike. Por otro se experimentó con el acercamiento opuesto, surgieron juegos que mantenían la exploración, el contenido generado de manera aleatoria y la muerte permanente; pero abandonando el combate por turnos y el uso de un tablero [2].

Estos últimos gozan de una muy buena salud actualmente, habiéndose visto en la última década una gran cantidad de producciones. *The Binding of Isaac* (2011), uno de sus principales exponentes, había vendido más de cinco millones de copias ya en 2015 [2]. Otros exponentes son *Spelunky* (2008), *Dead Cells* (2017), *Risk of Rain 2* (2019) o *Hades* (2020). Incluso Sony ha utilizado un shooter roguelike, *Returnal* (2021), como una de las primeras grandes producciones exclusivas de PlayStation 5. Ninguno de los anteriores juegos tiene sistemas ni de turnos ni de casillas y encontramos entre ellos dos de scroll lateral, dos en plano cenital y dos shooters en tercera persona.

2.1.3. Retorno de las características clásicas

Pese a que esta nueva corriente aporta unos planteamientos interesantes y ha conseguido atraer a gran cantidad de usuarios, lo cierto es que Rogue es difícilmente reconocible en los roguelikes hegemónicos actuales. En general, la renuncia a las mecánicas clásicas inspiradas en los juegos de mesa populares de aquella época supone que son difícilmente comparables con casi cualquier roguelike clásico. Esto no es un problema en sí mismo y es normal que la industria evolucione, pero siguen existiendo jugadores interesados en aquellas mecánicas, más pausadas y estratégicas, de la misma forma en que siguen vigentes muchos de aquellos juegos de mesa.

Sin tanta atención por parte del público general y con menor número de títulos, sí que hemos visto en los últimos años algunas producciones mucho más similares a estos juegos originales del género. Stoneshard (2020) que sigue en desarrollo en la modalidad de acceso anticipado, conserva las mecánicas de movimiento en un tablero cuadriculado y el sistema de turnos original. La introducción de animaciones y otros elementos más adecuados para un desarrollo moderno lo convierten en la clase de producto a la que este proyecto buscaría facilitar el desarrollo, permitiendo la aparición de mayor variedad en las ambientaciones y enfoques. En el mercado de móvil también encontramos otros títulos que conservan ese enfoque más clásico, aunque en general se trata de proyectos más modestos y que evitan las complicaciones renunciando a la presencia de elementos más modernos.



Figura 2.2: Captura de Stoneshard (2020).

2.2. Motores de juego

La implementación de este proyecto se desarrollará a través del sistema de extensiones proporcionado por el motor de juego Unity, que será analizado en mayor detalle en la próxima sección. Sin embargo, resulta de interés explorar todas las opciones existentes en la actualidad con el objetivo de justificar la elección de éste, así como de entender mejor su posición en el mercado y los aspectos positivos y negativos de dicha elección.

Unreal Engine

Unreal Engine, cuya primera versión llegó a través del shooter Unreal en 1998, es el motor de juego más asentado entre los grandes desarrolladores de la industria. Solo recientemente ha facilitado la entrada de pequeños proyectos y tradicionalmente ha sido utilizado principalmente en desarrollos de alto presupuesto. Cuenta con una enorme cantidad de videojuegos de éxito elaborados en él [3], perteneciendo éstos a géneros muy variados.

Es una solución generalista que aporta una enorme cantidad de herramientas a los desarrolladores para que adapten el motor a sus necesidades particulares. Soporta multitud de plataformas incluyendo Windows, macOS y Linux, así como las principales consolas, dispositivos móviles y de realidad virtual y navegadores web. Ofrece gráficos 3D de alta calidad junto a tecnologías punteras de trazado de rayos y gestión de la iluminación. Su lenguaje de programación principal es C++, por lo que su codificación es a relativamente bajo nivel y requiere de conocimientos profundos por parte de los desarrolladores. Actualmente se encuentra en su cuarta iteración, Unreal Engine 4 (2014), y está previsto que Unreal Engine 5 esté disponible durante la primera mitad de 2022, con su versión en acceso anticipado ya disponible desde mayo de 2021 [4].

Si bien Unreal Engine es extremadamente popular y hay una enorme cantidad de recursos de la industria invertidos en él, su posición en el mercado no está correctamente alineada con los objetivos de este proyecto. Un título que cuente con un muy alto presupuesto casi con total seguridad desarrollará todos sus sistemas de manera interna, descartando el uso de extensiones de terceros. Además, el público objetivo de esta extensión es poco probable que se decante por un motor que está más enfocado a equipos de gran tamaño.

Unity

Unity, anunciado y lanzado inicialmente como un motor de juego exclusivo de Mac OS X en 2005, es la opción por excelencia de los equipos de desarrollo pequeños y medianos. Poco después de su lanzamiento dieron soporte a Windows y desde hace años ofrecen unas condiciones amigables con los desarrolladores “indie” o independientes, así como con equipos pequeños y medianos que no cuentan con un gran presupuesto. Estas condiciones incluyen la completa omisión del pago de licencias hasta

alcanzar unos beneficios de cien mil dólares anuales, la existencia de una documentación pública [5], la integración sencilla de una tienda interna con assets y herramientas en muchas ocasiones gratuitos y la menor complejidad en comparación con Unreal Engine.

El motor también ofrece gráficos de alta definición y ha sido muy utilizado también en proyectos 2D. Da soporte a prácticamente las mismas plataformas que Unreal Engine y se trata también de una solución de carácter generalista. Su lenguaje principal es C#, por lo que la programación se realiza a un nivel algo más alto. Su desarrollo no se ha dividido en iteraciones diferenciadas y se basa en un sistema de actualizaciones continuas.

Unity se adapta mejor a los objetivos de esta extensión, ya que tanto su público objetivo y como su filosofía coinciden en mayor medida con los que perseguimos. Un gran número de proyectos de éxito se han desarrollado en esta plataforma [6], incluyendo títulos con elevado presupuesto, y también cuenta con una gran popularidad e inversión de recursos. Sin embargo, en este caso, la presencia de equipos pequeños y medianos sí que hace mucho más común recurrir al apoyo en funcionalidades ya existentes para facilitar o incluso posibilitar el desarrollo de nuevos videojuegos.

Godot

Godot es un proyecto libre y de código abierto cuyo primer lanzamiento se produjo en 2014. Es el motor gestionado por la comunidad con mayor constancia e impulso en su desarrollo. Su utilización es completamente gratuita y permite la comercialización de los juegos implementados en él.

El motor ofrece numerosas herramientas para facilitar sus tareas a los desarrolladores. Sin embargo, no cuenta con todas las tecnologías que ofrecen otros competidores, especialmente en materia gráfica. Su programación se realiza en un lenguaje propio, GDScript, que está claramente inspirado en Python, por lo que se trabaja a un alto nivel y resulta sencillo de aprender a utilizar. Cuenta con un equipo permanente de desarrolladores gracias al apoyo económico de la comunidad, si bien no es comparable al tamaño de los equipos de los dos anteriores motores, y también se realizan aportaciones directas de código.

Ésta podría ser la opción ideal de no ser por su relativa inmadurez y carencia de parte del espectro de opciones que encontramos en otras alternativas. Ningún título de popularidad significativa ha sido desarrollado en este motor hasta la fecha, lo que es otra señal del recorrido que tiene todavía por delante. Si en un futuro se solventasen estos problemas podría convertirse en la mejor opción disponible, debido a que la filosofía de su comunidad se basa precisamente en la colaboración en el desarrollo y en el aprovechamiento del trabajo realizado por el resto de miembros.

GameMaker Studio y RPG Maker

GameMaker Studio, actualmente en su segunda iteración, y RPG Maker, cuya primera versión fue lanzada en 1992, son alternativas especializadas en juegos roguelike o de géneros similares y son probablemente la opción en la que más sencillo resultaría el desarrollo de un nuevo título. Sin embargo, precisamente el objetivo de este proyecto es utilizar una extensión para dar acceso a toda la amplia gama de herramientas y tecnologías que ofrecen los potentes motores modernos, por lo que estas opciones no serían un entorno adecuado para ello. Primero, porque ya existen muchas de las herramientas especializadas para su desarrollo y, segundo, porque no ofrecen esa amplia gama de herramientas potentes a las que buscamos dar acceso al género roguelike.

Motores in-house

En la actualidad siguen existiendo numerosos motores in-house, o de desarrollo interno, detrás de títulos de éxito. Motores como REDEngine, presente en Cyberpunk 2077, o Decima, detrás de las entregas de Horizon, han proporcionado algunos de los gráficos y físicas más punteros en los últimos años. Estos motores son la mayor representación de aquello alcanzable por equipos con enormes presupuestos, capaces de desarrollar soluciones completas adaptadas solo a sus propias necesidades. Esto los descarta como potenciales objetivos para este proyecto, ya que en la mayoría de casos se trata de tecnologías cerradas para las que no se pueden desarrollar módulos adicionales. Además, sus usuarios son casi exclusivamente los equipos más grandes de la industria, que jamás utilizarían herramientas o sistemas que no hayan sido desarrollados por ellos.

Motor de juego propio

Otra alternativa por la que se podría optar para el desarrollo de este proyecto es la implementación desde cero de un nuevo motor de juego propio. Sin embargo, por los mismos motivos por los que hemos descartado soluciones especializadas, éste no sería un acercamiento interesante. Dado que uno de los principales objetivos es proveer de las herramientas y tecnologías modernas y punteras al desarrollo de los roguelike, sería necesario realizar todas esas implementaciones por cuenta propia. Esta tarea es completamente inviable para un equipo pequeño y jamás se lograría alcanzar a las tecnologías punteras, que están en constante avance y evolución. Incluso en el caso de que se pudiesen alcanzar de alguna forma, supondría tener a un equipo dedicado de manera permanente al mantenimiento y desarrollo, ya que siempre llegarían nuevas tecnologías que implementar.

2.3. Desarrollo en Unity

Una vez hemos justificado la elección de Unity como motor de juego sobre el que desarrollar el proyecto nos resulta de interés conocer todas sus características. La versión utilizada será la estable más reciente, Unity 2020.3.10f1.

2.3.1. Plataformas soportadas

Las plataformas soportadas por Unity actualmente son:

- Ordenadores personales: Windows, macOS y Linux.
- Consolas: PlayStation, Xbox, Nintendo Switch y Stadia.
- Dispositivos móviles: iOS, Android, tvOS y Android TV.
- Plataformas web: WebGL.
- Realidad virtual y aumentada: más de siete de las alternativas más populares del mercado.



Figura 2.3: Plataformas publicitadas por Unity en su web.

2.3.2. Características principales

Las principales herramientas y características que ofrece son:

- Un editor claro, modular y adaptable a las necesidades de cada desarrollador.
- Pipelines para gráficos 2D y 3D pensados para equipos de diferentes tamaños y presupuestos, así como la posibilidad de crear pipelines propios y editar los existentes.
- Motores de físicas tanto para elementos 2D como para elementos 3D.
- Controladores de alto nivel de la entrada del usuario a través de diferentes interfaces.
- Programación mediante C# a través de sus librerías y recientemente también visual scripting.
- Gestión avanzada del audio con un sistema de emisores y receptores de sonido.

- Editores de imágenes, animaciones, shaders y efectos de partículas.
- Soluciones para el juego multijugador en red.
- Herramientas para la gestión y sincronización del desarrollo colaborativo en paralelo.
- Creación de interfaces de usuario flexibles y adaptables a las necesidades de cada proyecto.
- Inteligencia artificial con implementaciones de pathfinding e incluso aprendizaje automático.
- Servicios opcionales para la recolección de estadísticas, monetización de anuncios, gestión de servidores y otras funcionalidades en la nube.
- Paquetes desarrollados y mantenidos por Unity que expanden la funcionalidad base.
- Extensiones desarrolladas por la comunidad, que también amplían las posibilidades del motor y cuyo sistema de soporte es especialmente relevante para la viabilidad de este proyecto.
- Una tienda interna donde obtener e instalar extensiones u otros assets de pago o gratuitos.

Un paquete que podría ser de especial utilidad durante la implementación de la extensión permite además gestionar con mayor facilidad los assets de un tablero de juego, aunque actualmente parte de su funcionalidad se encuentra todavía en fase de pruebas.

2.3.3. Recursos disponibles

Unity cuenta con una comunidad de gran tamaño, por lo que, además de su documentación [5], también serán de gran utilidad sus foros [7] y su portal educativo, Unity Learn [8]. Muchos productos exitosos han sido desarrollados en el motor [6], por lo que tendremos muchas referencias que consultar para hacernos una idea de hasta dónde puede llegar el motor y cuáles son sus límites. Además, en su web podemos encontrar la sección Made With Unity [9], que nos presenta diferentes proyectos junto a detalles de su implementación.

2.3.4. Dificultades previstas

La mayor dificultad prevista viene de la solución generalista de Unity al bucle de juego. El acercamiento de Unity descentraliza la lógica dándole su propia función equivalente al bucle de juego a cada uno de los agentes. Si bien este planteamiento puede ser muy positivo en otro tipo de géneros, conlleva muchos problemas si se intenta combinar con un sistema de turnos. Es especialmente negativo en el caso de los roguelikes ya que, como veremos, muchas acciones deben ocurrir en paralelo y sincronizadas entre sí.

Otra característica que deberá ser construida prácticamente desde cero es el tablero, ya que aunque podremos beneficiarnos de alguna herramienta de Unity, este tipo de mecánicas de movimiento no cuentan con soporte oficial.

2.4. Sistema de extensiones de Unity

Como ya hemos visto, el sistema de extensiones de Unity es lo que posibilita el desarrollo de la funcionalidad adicional que busca incorporar este proyecto. A continuación, veremos algunos casos de éxito a través de este sistema y estudiaremos cómo se empaquetan y distribuyen las extensiones.

2.4.1. Casos de éxito

Bolt

Bolt es una extensión que introduce una solución de visual scripting, o programación visual o por bloques, a Unity. No trata de sustituir a C# en todos los niveles, sino que busca mejorar el flujo de trabajo de equipos en los que no todo el mundo tenga un perfil técnico de programación o que simplemente deseen una mayor jerarquización de su lógica. Permite encapsular scripts escritos en C# dentro de bloques, que luego pueden ser conectados de manera visual para definir la lógica del programa; estableciendo, por ejemplo, una máquina de estados para la gestión de misiones o tutoriales.

Su popularidad provocó que en 2020 Unity decidiese adquirir el proyecto para hacerlo gratuito para todos los usuarios y para seguir desarrollándolo y mejorándolo, llegando incluso a integrarlo como parte oficial del motor [10]. Es un claro caso de éxito mediante la modificación del flujo de trabajo y la aportación de nuevas herramientas a través del sistema de extensiones.

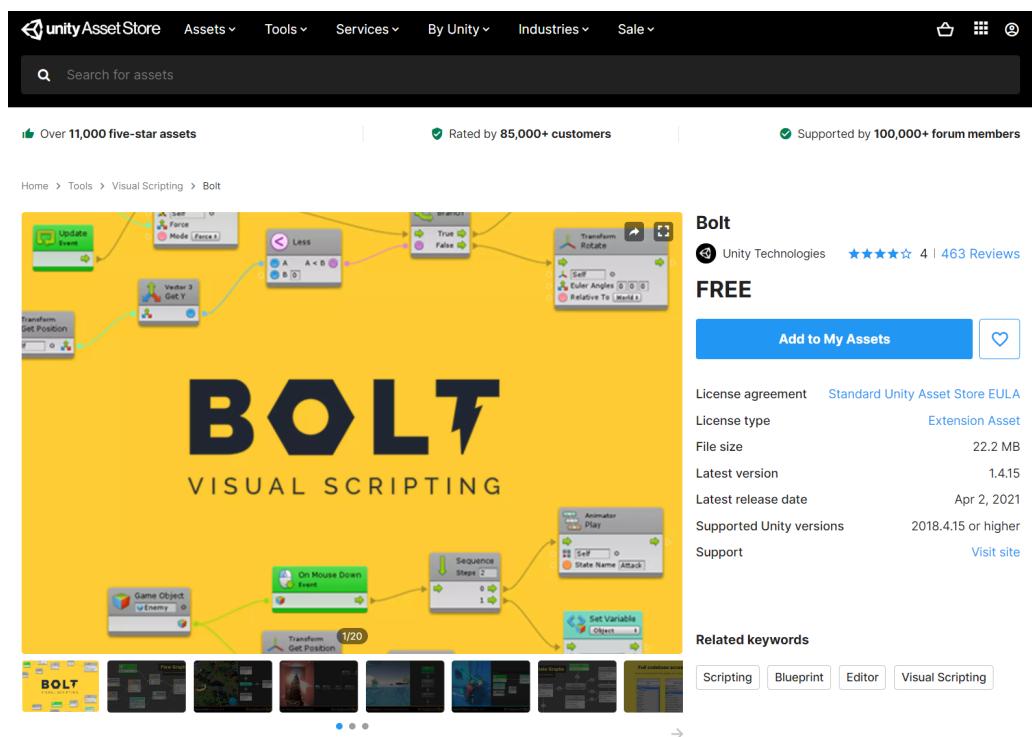


Figura 2.4: La extensión Bolt en la Asset Store de Unity.

uMMORPG

También con una gran popularidad y con un enfoque muy similar al de este proyecto, encontramos uMMORPG, que proporciona una gran cantidad de herramientas para facilitar el desarrollo de juegos del género MMORPG (Massively Multiplayer Online Role-Playing Game). Numerosos desarrolladores han decidido realizar sus proyectos en el marco de esta extensión, cuya monetización ha permitido continuar su desarrollo y mejora. Nos permite comprobar la viabilidad de este tipo de acercamientos, especialmente si tenemos en cuenta que el género roguelike no sufre las complicaciones del juego multijugador masivo en línea.

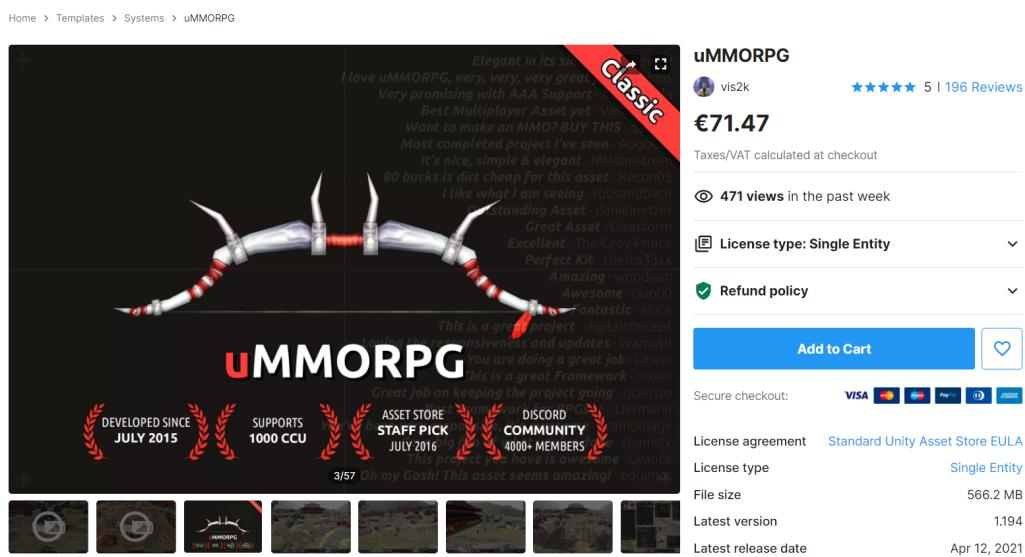


Figura 2.5: La extensión uMMORPG en la Asset Store de Unity.

2.4.2. Empaquetamiento y distribución

El proceso para crear una extensión en Unity es el mismo que para cualquier otro tipo de asset:

1. Se registra un boceto del paquete.
2. Se añaden todos los elementos que pertenecen a dicho paquete desde el editor. Estos pueden ser modelos 3D, ficheros de sonido o animaciones; pero en el caso de una extensión también se incorporarán scripts que añadan funcionalidad al videojuego o que modifiquen el comportamiento del editor.
3. Se rellenan todos los detalles y metadatos referentes al paquete.
4. Se solicita la aprobación de Unity para añadirlo a su Asset Store.

Tras recibir la correspondiente aprobación de Unity, la extensión estará disponible en su tienda y será fácilmente localizable e instalable por el resto de usuarios del motor.

DISEÑO

A lo largo de este capítulo quedará definido el diseño de la extensión. Esto incluirá los objetivos que persigue su desarrollo, requisitos funcionales y no funcionales, diagramas de diseño y arquitectura y una especificación de las tecnologías, metodologías y estándares que se van a utilizar.

3.1. Objetivos de la extensión

Empezaremos definiendo los objetivos que determinarán el resto del diseño. Lograrán un mayor nivel de concreción en las siguientes secciones, por lo que se limitarán a marcar las características y funcionalidades principales que deben estar presentes en el producto final. Estos objetivos son:

- O-1.** Proporcionar un bucle de juego, o game loop, que ofrezca un flujo de trabajo alternativo al de Unity, mejor adaptado a las necesidades de una arquitectura por turnos.
- O-2.** Configurar un gestor de turnos que determine qué agente debe actuar en cada momento.
- O-3.** Dar soporte a un tablero que implemente un sistema de capas, permitiendo definir las acciones e interacciones que ocurren entre los elementos que lo habiten.
- O-4.** Facilitar el aprovechamiento de las peculiaridades del tablero en cuadrícula, optimizando y minimizando el trabajo requerido por los assets que lo representen a él y a sus elementos.
- O-5.** Otorgar implementaciones base de las entidades y agentes que pueblan el tablero.
- O-6.** Desarrollar controladores que determinen las acciones de los agentes en función de si se encuentran bajo el control del jugador o si, por el contrario, obedecen a sus características propias.
- O-7.** Gestionar la entrada del usuario para incorporarla correctamente al nuevo bucle de juego.
- O-8.** Soportar en el sistema de acciones composiciones recursivas de éstas, tanto en secuencia como en paralelo, para posibilitar acciones finales ilimitadamente complejas.
- O-9.** Permitir el uso de acciones asíncronas que tengan una duración que comprenda múltiples iteraciones del bucle de juego, garantizando la sincronización y coherencia del sistema.
- O-10.** Generar mapas de mazmorras proceduralmente en base a la configuración indicada.
- O-11.** Implementar herramientas de interfaz de usuario que permitan al jugador explorar la información presente en el tablero de juego y en los elementos contenido en él.

- O-12.** Separar de manera estricta los datos de la funcionalidad, en todos los aspectos posibles.
- O-13.** Empaquetar y distribuir la funcionalidad anterior a través del sistema de extensiones.
- O-14.** Dar acceso a la funcionalidad implementada a través del uso o extensión de las clases e interfaces provistas en el paquete, permitiendo realizar modificaciones.

3.2. Requisitos funcionales

Una vez hemos definido los objetivos que persigue el desarrollo de la extensión, podemos proceder a especificar todos los requisitos de funcionalidades y características concretas que deben estar presentes en la implementación de cada uno de sus componentes. Estos requisitos son:

Requisitos de la arquitectura y el bucle de juego

RF-1. Interfaz con el desarrollador.

La funcionalidad proporcionada mediante esta extensión será accesible a través del uso y ampliación de las interfaces y clases proporcionadas en los scripts incluidos.

RF-2. Centralización del bucle de juego.

Se proporcionará una alternativa al modelo descentralizado de Unity, seguirá una arquitectura jerarquizada en la que los distintos módulos irán ejecutando sus subsistemas y elementos de manera secuencial y en un orden coherente y preestablecido.

RF-3. Orden de ejecución del bucle de juego.

Los principales sistemas se ejecutarán en el siguiente orden:

1. Captura de la entrada del usuario.
2. Gestión de turnos y activación de los controladores correspondientes.
3. Procesamiento de las acciones actualmente en ejecución o en cola.
4. Actualización de la interfaz de usuario.

RF-4. Gestión de los turnos.

Cuando no existan acciones en ejecución o en cola que bloqueen la creación de nuevas acciones, se utilizará el gestor de turnos para determinar el agente que debe actuar. El gestor tendrá en cuenta las acciones pasadas y las estadísticas de los agentes.

Requisitos del tablero

RF-5. Topología del tablero.

El tablero estará estructurado de manera cuadriculada, dividiéndose en casillas. Vendrá definido por su altura y anchura, proporcionadas en número de casillas.

RF-5.1. Carga de la topología de un tablero desde un fichero de datos.

La topología de un tablero, y cualquier otra característica intrínseca a él, podrá ser definida en un fichero de datos.

RF-6. Elementos del tablero.

Las casillas del tablero podrán ser ocupadas por entidades, agentes y terreno.

RF-7. Modificación del tablero.

La implementación del tablero permitirá añadir, mover y eliminar entidades y agentes con facilidad, así como modificar el tipo de terreno presente en cada una de las casillas.

RF-7.1. Garantía de coherencia del tablero.

Antes de realizar cualquier modificación se comprobará que se trate de un cambio válido, ignorándola en caso contrario.

RF-8. Observación del tablero.

Se implementarán métodos de consulta que proporcionen de manera sencilla toda la información que el resto de módulos requieran sobre el estado del tablero.

RF-9. Estructuración del tablero en capas.

Cada tipo de elemento del tablero contará con su propia capa, en la que quedará registrada su presencia. Se definirán las interacciones entre elementos de diferentes capas en base a sus propiedades. Por ejemplo, un agente volador se podrá situar sobre un abismo.

RF-10. Optimización de la capa de terreno del tablero.

El terreno, al contrario que las entidades y los agentes, no tendrá ninguna clase de estado asociado, sino que se tratará de una propiedad de la casilla. Se gestionará de manera completamente automática la instanciación de los modelos 2D o 3D asociados a dicho terreno y su correcta interactuación y organización con los terrenos de las casillas adyacentes. Estos comportamientos se especificarán mediante la definición de reglas y la asignación de los modelos.

Requisitos de las entidades y los agentes**RF-11. Características de las entidades.**

Las entidades quedarán definidas por su posición en el tablero y su modelo 2D o 3D asociado. Sistemas que requieran funcionalidades más complejas, como por ejemplo la presencia de un inventario dentro de un cofre, expandirán la funcionalidad de esta implementación base.

RF-11.1. Carga de las características de una entidad desde un fichero de datos.

Las características de una entidad podrán ser definidas en un fichero de datos.

RF-12. Características de los agentes.

Los agentes poseerán todas las características de una entidad y, además, tendrán una serie de estadísticas comunes a todos los agentes.

RF-12.1. Estadísticas básicas comunes a todos los agentes.

Como mínimo, los agentes poseerán las siguientes estadísticas, necesarias para su interactuación: nombre, tiempo de recarga de turno, prioridad de turno, vida máxima, ataques, ser fantasma y ser vagabundo.

RF-12.2. Carga de las características de un agente desde un fichero de datos.

Las características de un agente podrán ser definidas en un fichero de datos.

Requisitos de los controladores

RF-13. Características comunes a los controladores.

Los controladores, independientemente de su tipo, poseerán un método para obtener de ellos una acción asociada a un agente concreto. De esta forma, se podrá resolver el turno de un agente solicitando al controlador adecuado una acción en su nombre.

RF-14. Controlador jugador.

Cuando el turno corresponda a un agente controlado por el jugador, se utilizará la entrada inyectada en este controlador al inicio del bucle de juego para generar su acción.

RF-15. Controlador no jugador.

Cuando el turno corresponda a un agente no controlado por el jugador, se ponderarán las características de dicho agente, así como su entorno, para decidir qué acción generar.

Requisitos del sistema de acciones

RF-16. Procesamiento de las acciones.

Las acciones generadas por los controladores se encolarán y procesarán en los sucesivos bucles de juego. Por lo general, no se podrán crear nuevas acciones hasta finalizar las encoladas.

RF-17. Tipos de acciones básicas.

Como mínimo, se desarrollarán las siguientes acciones básicas: acción de creación, acción de toma de control del jugador, acción de movimiento, acción de ataque y acción nula.

RF-18. Acciones compuestas y recursividad.

Se dará soporte a acciones que sean una combinación de otras acciones, para su ejecución tanto en serie como en paralelo. Esto permitirá incorporar acciones dentro de otras de manera recursiva e ilimitada, para la representación de comportamientos complejos.

RF-18.1. Garantía de coherencia de las acciones.

La capacidad de ejecutar acciones en paralelo conlleva el riesgo de que se produzcan condiciones de carrera o combinaciones incompatibles. No se comprobará de manera previa a su aplicación la garantía de imposibilidad de conflictos, por lo que al utilizarse deberá hacerse un uso cuidadoso. Sí se comprobará la validez de cualquier acción antes de aplicarse, ignorándose en caso de ser inválida. Por lo tanto, el juego nunca llegará a un estado incoherente, pero será posible que no todas las acciones sean procesadas.

RF-18.2. Paralelización de la acción principal.

La acción raíz, sobre la que se integrarán todo el resto de acciones que estén siendo procesadas en un momento dado, será una acción en paralelo. Esto permitirá ejecutar los turnos de varios agentes de manera paralela bajo determinadas circunstancias.

Requisitos de la generación procedural de contenido

RF-19. Generación procedural de mapas de mazmorras.

El generador debe soportar diferentes tamaños y números de salas y garantizar la conexividad.

RF-20. Presencia de entidades y agentes en los mapas de mazmorras.

Las mazmorras generadas proceduralmente no solo definirán el terreno, marcando las salas y pasillos, sino que también establecerán puntos de aparición para entidades y agentes.

Requisitos de la interfaz del jugador**RF-21. Administrador central de la interfaz del jugador.**

Un gestor central de la interfaz del jugador se encargará de interpretar algunas de las entradas del jugador y actualizará cuando corresponda el resto de elementos, proporcionándoles esta información ya procesada.

RF-22. Indicador de casillas.

El indicador de casillas proporcionará información al jugador sobre la casilla señalada por el cursor, mostrando por ejemplo si el agente controlado actualmente podría moverse a ella.

RF-23. Inspector de agentes.

El inspector mostrará las estadísticas del agente situado en la casilla señalada por el cursor.

RF-24. Indicador del agente bajo control.

Una pequeña indicación visual señalará al agente controlado por el jugador en cada momento.

RF-25. Controlador de la cámara.

Se proporcionará una implementación básica para el controlador de la cámara, capaz de seguir al agente bajo control en cada turno.

3.3. Requisitos no funcionales

Adicionalmente, podemos especificar otras características y criterios con los que juzgaremos la implementación realizada, creando por tanto unos estándares que se deberán alcanzar.

RNF-1. Capacidad de modificación.

Pese a que el uso habitual no lo requiera, se considerará crucial tener acceso a todos los elementos que determinen el funcionamiento de la extensión, permitiendo modificarla y expandirla.

RNF-2. Desacoplamiento entre módulos.

Además de por la salud del desarrollo, se considerará imprescindible garantizar la modularidad y el desacoplamiento para favorecer su modificación y expansión. Se deberán utilizar los medios y tecnologías necesarios para ofrecer garantías de desacoplamiento por diseño.

RNF-3. Desacoplamiento entre funcionalidad y datos.

El diseño estará enfocado a que todas las diferencias entre distintos elementos de una misma categoría estén reflejadas únicamente en el fichero de datos del que se hayan cargado.

RNF-4. No limitación de las plataformas soportadas por Unity.

Ni los métodos de entrada ni ningún otro elemento de la extensión debe limitar las posibilidades de los desarrolladores en cuanto a las plataformas disponibles para sus productos.

3.4. Arquitectura, módulos y clases

Con los objetivos y requisitos de la aplicación correctamente definidos, podemos plantear un diseño que los satisfaga. A continuación, veremos cómo se estructuran los módulos y las dependencias que existen entre ellos, así como los componentes de cada uno de estos módulos.

3.4.1. Arquitectura y dependencias

El siguiente diagrama nos muestra los distintos módulos de la aplicación y a cuáles de ellos tiene como dependencia cada uno.

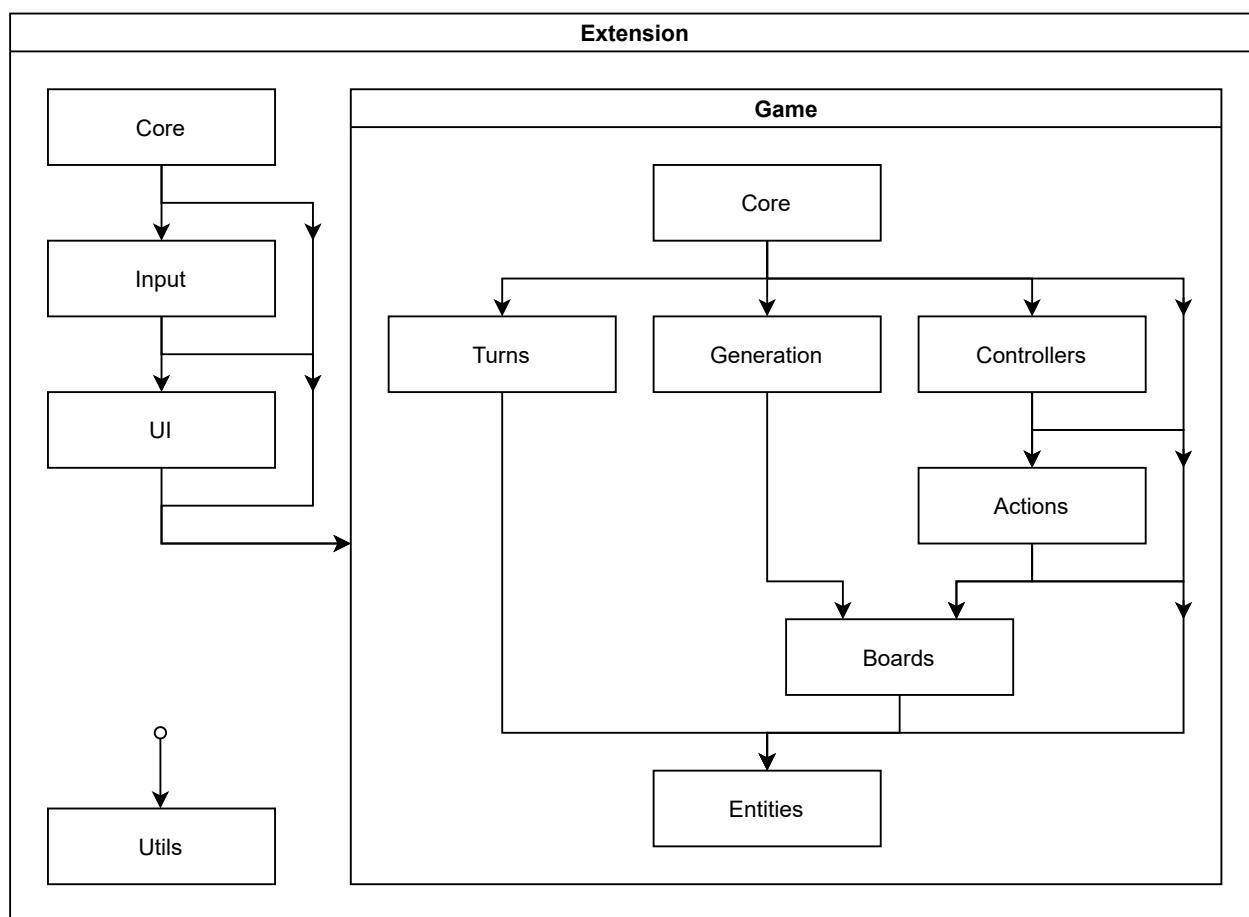


Figura 3.1: Diagrama de módulos y dependencias.

Podemos observar cómo no existe ninguna dependencia circular y cómo cada módulo tiene únicamente acceso a aquellos otros módulos que intervienen directamente en su lógica, logrando un desacoplamiento valioso para el desarrollo, mantenimiento y expansión de los mismos.

También podemos observar un diagrama de secuencia de un fotograma en la arquitectura de Unity.

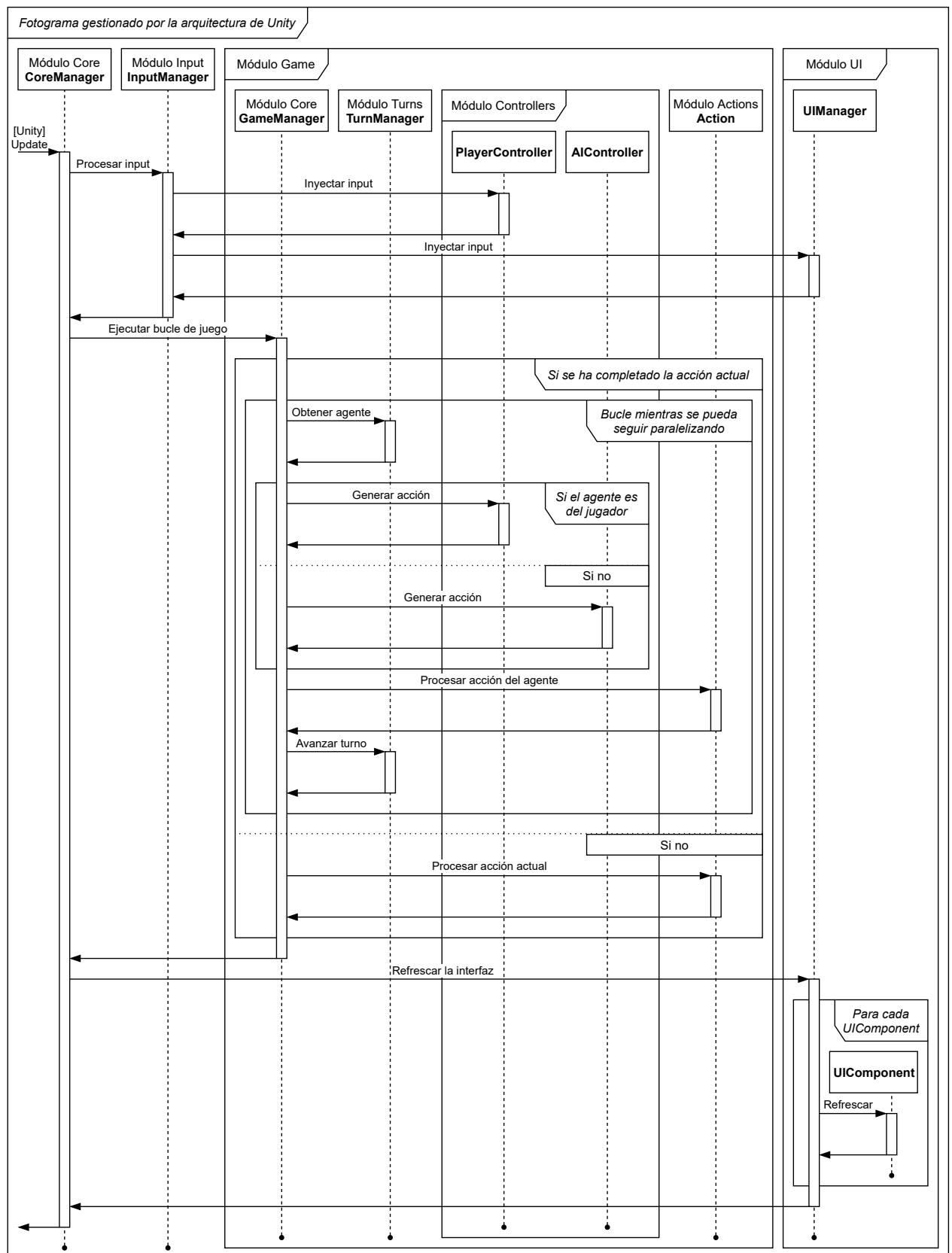


Figura 3.2: Diagrama de secuencia de un fotograma en la arquitectura de Unity.

3.4.2. Módulos

Se presenta a continuación el diseño de cada módulo mediante sendos diagramas de clases.

Convenios establecidos

Todas las clases, interfaces y otras estructuras y tipos proporcionados por Unity se considerarán equivalentes a tipos primitivos. Es decir, se indicará junto a la variable el nombre del tipo al que pertenecen y no se establecerá ningún tipo de conexión UML con otro elemento independiente. Esto también se aplicará en el caso de los tipos definidos en el paquete Utils, ya que se trata de elementos simples y muy extendidos en el uso de la aplicación, por lo que su representación tradicional en UML reduciría enormemente la legibilidad de los diagramas. La etiqueta [UM] significa *Unity Method*, señala aquellos métodos que Unity ejecuta como parte de su arquitectura.

Se ha dividido el diagrama de clases de la aplicación en sendos diagramas de los módulos debido al elevado número de clases y la escasa claridad al representar todo el contenido en una única página.

Módulo Core

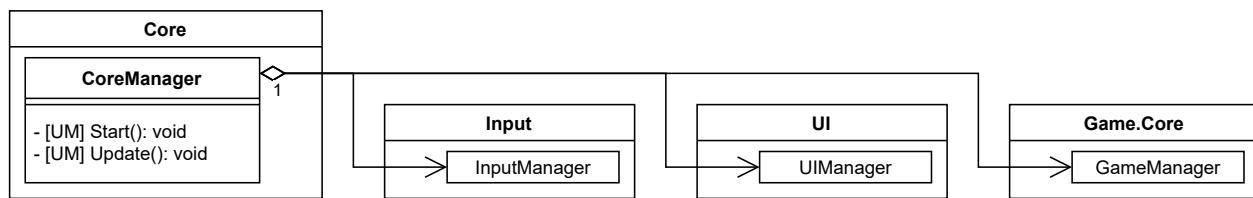


Figura 3.3: Diagrama de clases del módulo Core.

Módulo Input

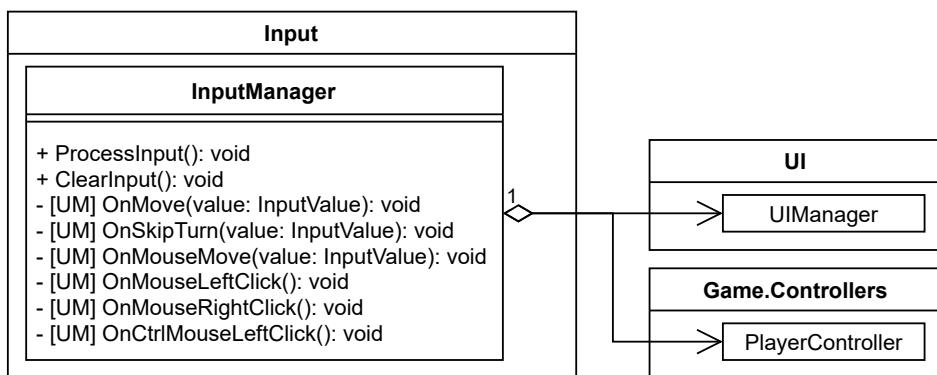


Figura 3.4: Diagrama de clases del módulo Input.

Módulo UI

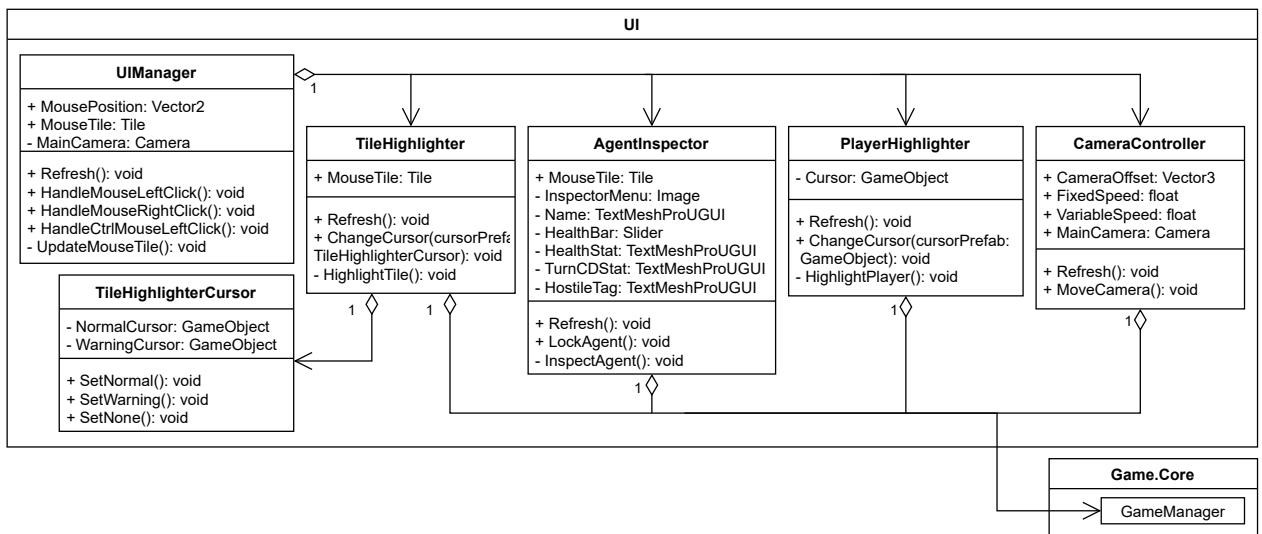


Figura 3.5: Diagrama de clases del módulo UI.

Módulo Game.Core

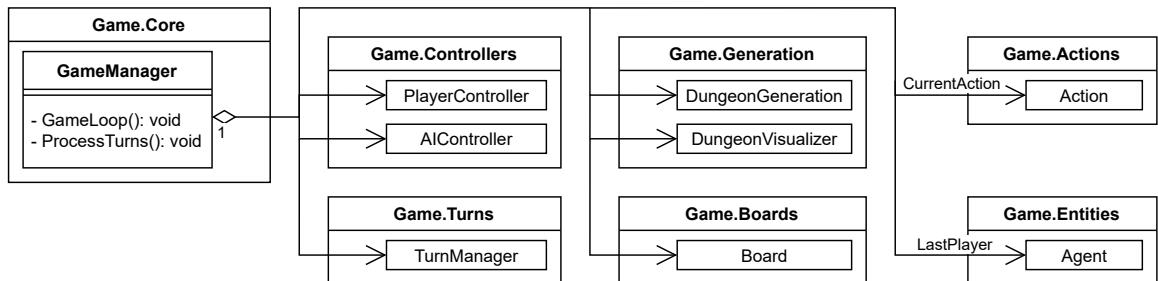


Figura 3.6: Diagrama de clases del módulo Game.Core.

Módulo Game.Turns

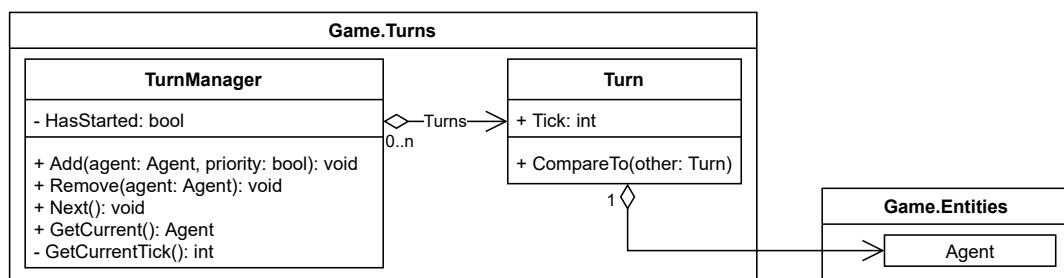


Figura 3.7: Diagrama de clases del módulo Game.Turns.

Módulo Game.Generation

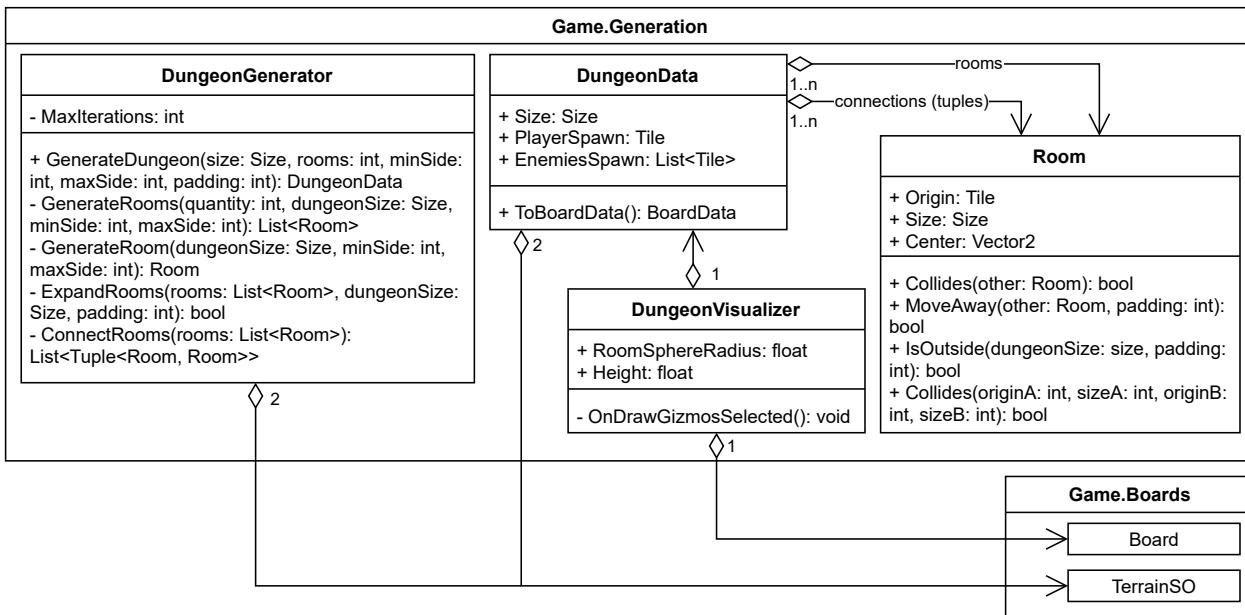


Figura 3.8: Diagrama de clases del módulo Game.Generation.

Módulo Game.Controllers

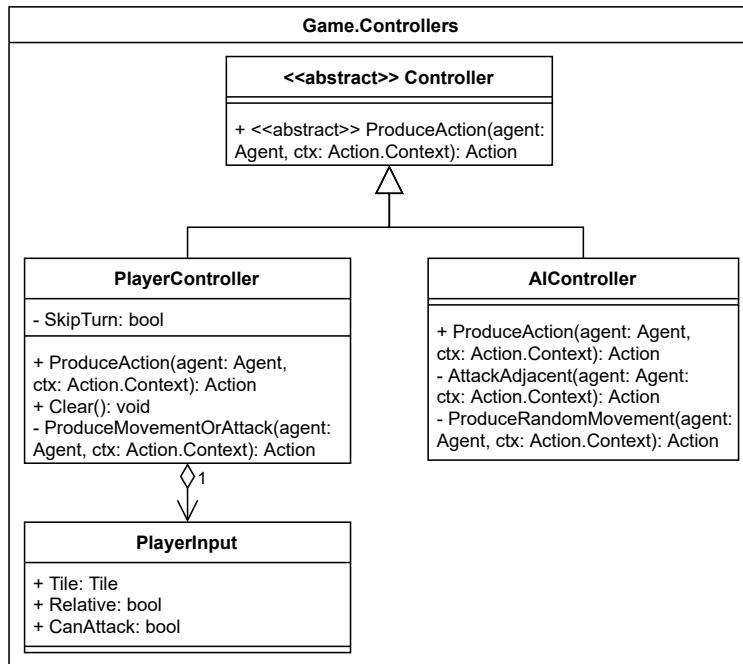


Figura 3.9: Diagrama de clases del módulo Game.Controllers.

Módulo Game.Actions

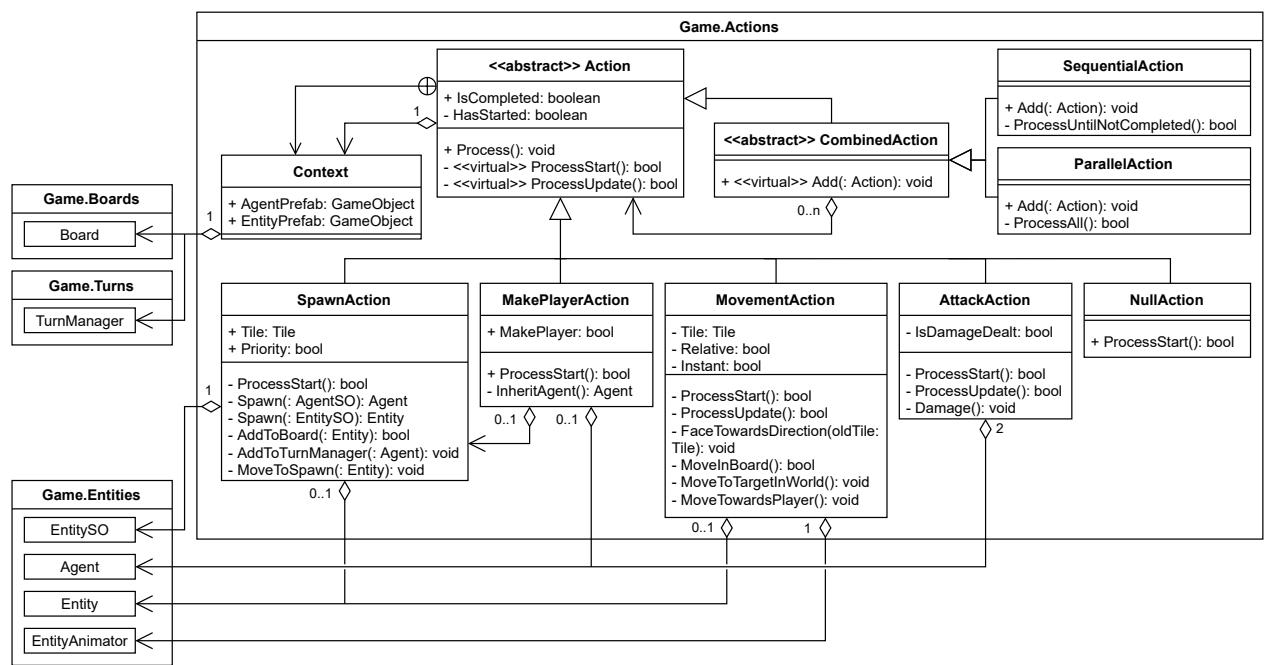


Figura 3.10: Diagrama de clases del módulo Game.Actions.

Módulo Game.Boards

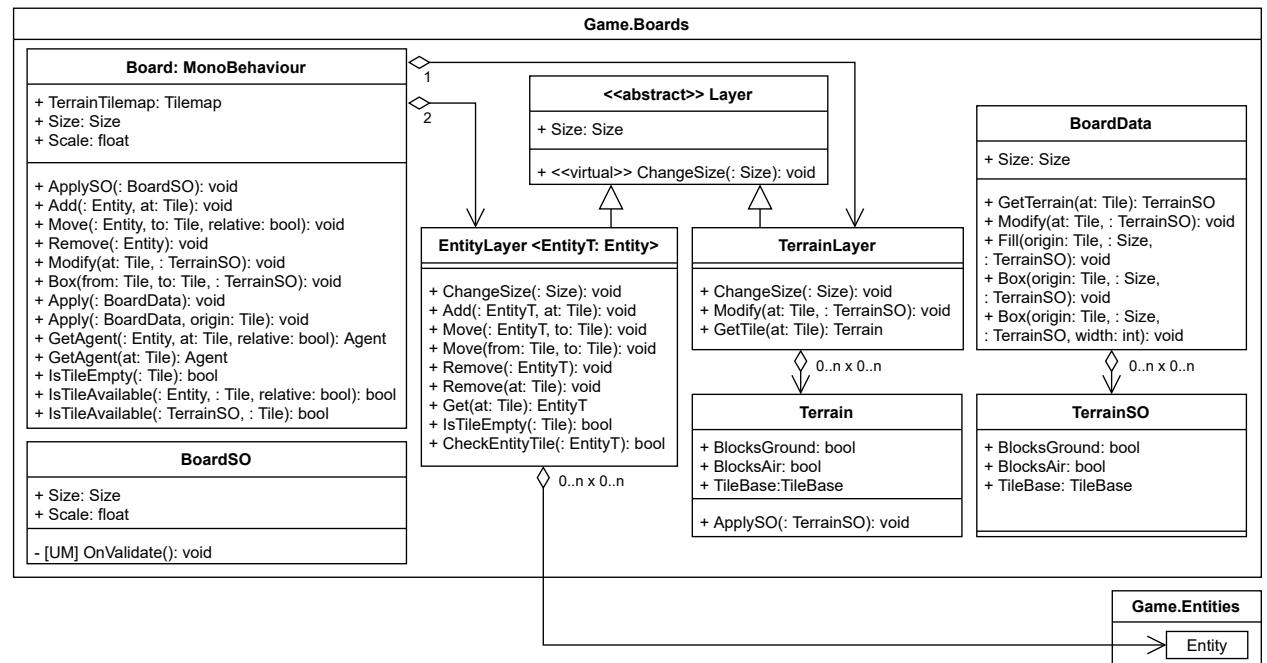


Figura 3.11: Diagrama de clases del módulo Game.Boards.

Módulo Game.Entities

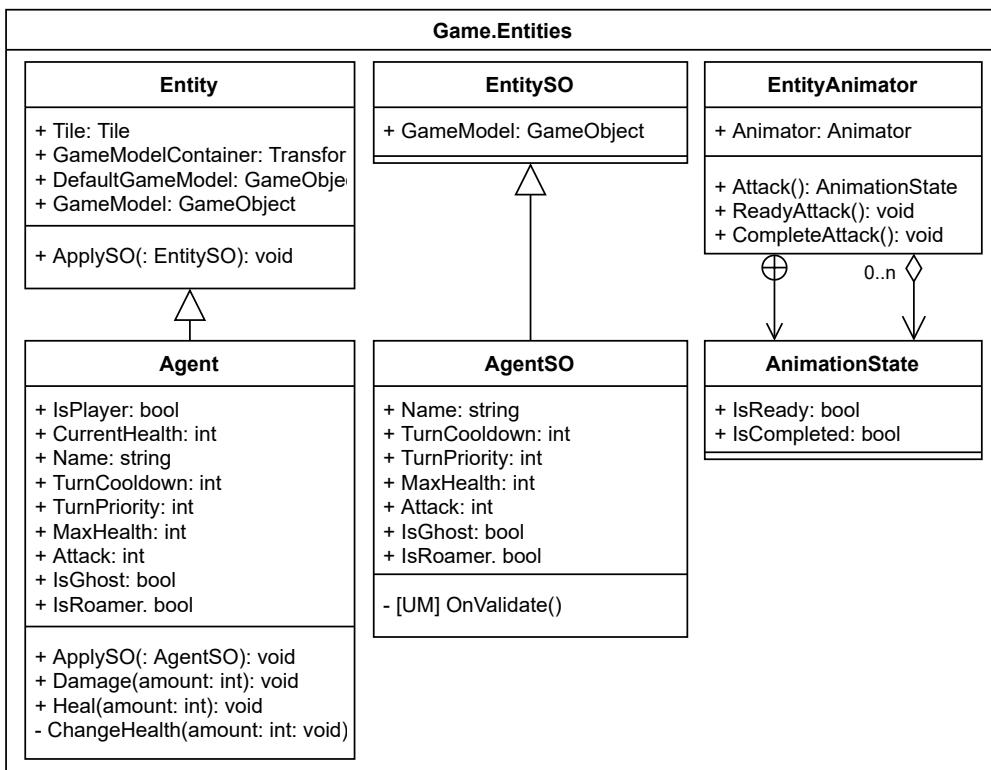


Figura 3.12: Diagrama de clases del módulo Game.Entities.

Módulo Utils

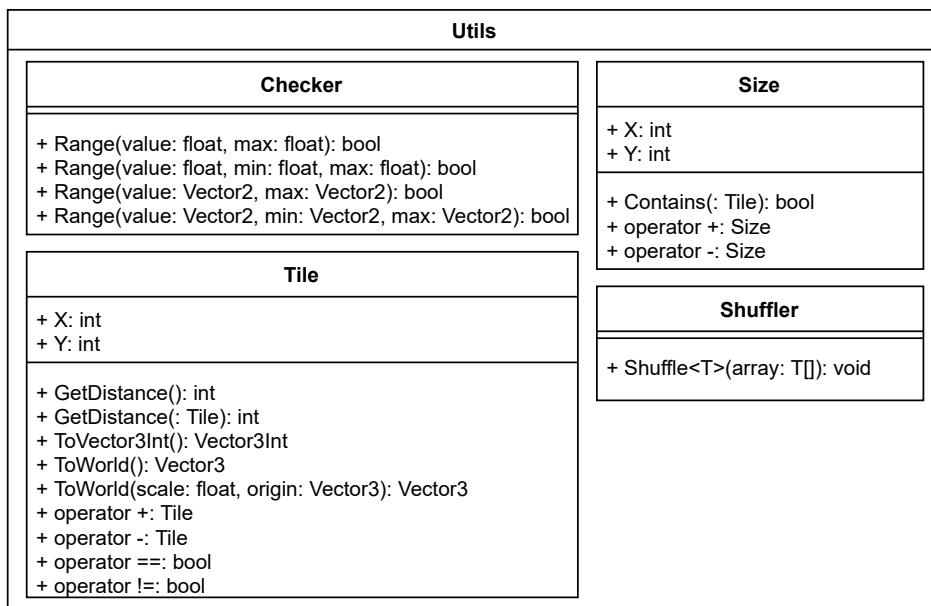


Figura 3.13: Diagrama de clases del módulo Utils.

3.5. Tecnologías y estándares

Se presentan a continuación las principales tecnologías que se utilizarán para la implementación del diseño elegido.



Figura 3.14: Tecnologías utilizadas.

C# 8.0, nullables y assembly definitions

La versión de Unity utilizada es la 2020.3.10f1, ya que es estable e incluye soporte a largo plazo pero contiene versiones recientes de algunos paquetes clave para el diseño de esta extensión. C# 8.0 [11] viene aparejado a esta versión de Unity, con interesantes características para favorecer las buenas prácticas en la programación.

Una de las novedades clave que aporta esta versión del lenguaje de programación, es la posibilidad de diferenciar entre referencias que pueden ser nulas y referencias que no [12]. El correcto uso de esta funcionalidad permitirá evitar la comprobación manual y los errores causados por referencias nulas inesperadas.

Otra característica muy enfatizada en el diseño de esta extensión es el desacoplamiento. Para ello se utilizará no solo el sistema de namespaces de C#, sino que se incorporará el uso de assembly definitions. Estas últimas permiten crear barreras estrictas que determinan qué partes del código tienen acceso a otras, compilando por separado cada uno de los módulos en los que se divide la aplicación. Esto garantizará que las relaciones entre módulos establecidas por el diseño tengan que ser respetadas o modificadas de manera consciente, además de mejorar los tiempos de compilación.

Scriptable objects

Para el desacople entre funcionalidad y datos se utilizará una de las herramientas proporcionadas por Unity, los scriptable objects. Éstos permiten definir una estructura de datos y después crear ficheros que sigan dicha estructura. Utiliza el formato Yaml, por lo que los ficheros cuentan con cierta legibilidad. Sin embargo, el motivo de su uso reside en el soporte nativo de Unity, que proporciona un editor específico para estos archivos adaptado al formato especificado previamente, y permite acceder a sus datos en tiempo real con mucha facilidad.

EditorConfig

Otra de las herramientas utilizadas para fomentar las buenas prácticas durante la implementación será EditorConfig. Esta tecnología, soportada por numerosos editores e IDEs, permite definir estándares respecto al estilo del código y los usos recomendados de las funcionalidades del lenguaje de programación utilizado.

Microsoft proporciona un fichero de configuración con los estándares recomendados para el uso de C# junto a EditorConfig [13]. Roslyn, el compilador de este lenguaje de programación, proporciona una API para el análisis automatizado del código, por lo que en todo momento se obtendrá feedback sobre el correcto seguimiento de los estándares elegidos.

Git

Para el control de versiones se utilizará Git. Unity proporciona su propia solución para esta tarea, Unity Collaborate, más adecuada para trabajar en el desarrollo de videojuegos con equipos heterogéneos que incluyan artistas y otros perfiles no técnicos. Sin embargo, para el desarrollo de esta extensión, consistente principalmente de ficheros de código, la sencillez de Git y su independencia de la plataforma han determinado su elección.

Para gestionar de manera apropiada la ocasional presencia de ficheros de mayor tamaño, como por ejemplo los assets que se deseen utilizar en las pruebas, se utilizará Git Large File Storage (LFS). Esta extensión de Git sustituye estos archivos de mayor tamaño por punteros, almacenando los originales por separado y gestionando su correcto reemplazo siempre que sean necesarios.

IMPLEMENTACIÓN

El desarrollo de la extensión se ha realizado en el marco de las metodologías ágiles. Las rápidas iteraciones apoyadas en herramientas como Kanban han permitido refinar el producto paso a paso. Uno de los objetivos principales de este acercamiento ha sido ir añadiendo funcionalidad por capas, sin comprometer la calidad del código o la facilidad de uso de la extensión por parte de los usuarios. A lo largo de este capítulo analizaremos el entorno de desarrollo, estudiaremos los detalles de la implementación de los módulos y veremos el empaquetado de la funcionalidad en una extensión.

4.1. Entorno de desarrollo

El sistema de extensiones de Unity está estructurado de tal forma que el desarrollo de una extensión es casi idéntico al proceso que seguiría la creación de un videojuego. Por lo tanto, el principal entorno de desarrollo ha sido Unity, que a su vez da soporte a varios IDEs y editores, de entre los cuales se ha optado por Visual Studio Code.

Para la creación de los assets utilizados para realizar comprobaciones y pruebas se han utilizado diversas herramientas. Como editores de imágenes se han elegido Gimp y Aseprite, el modelado 3D se ha realizado en Blender y MagicaVoxel y para la edición de audio se ha optado por Audacity.

Como tablero Kanban se ha utilizado Trello, un servicio online especializado en esta técnica.

4.2. Implementación de los módulos

Una de las ventajas de no tener dependencias circulares entre módulos es que se puede elegir un orden óptimo para su implementación. De esta forma, es posible lograr que el desarrollo de cada módulo se realice cuando todas sus dependencias cuenten de una implementación disponible. A continuación, se muestran los detalles de cada módulo en el orden aproximado en el que han sido desarrollados, aunque se debe tener en cuenta que la iteración propia de las metodologías ágiles supone que el orden no es absolutamente estricto.

La nueva arquitectura, caracterizada por un bucle de juego centralizado, ha encajado con gran facilidad con muchos rasgos inspirados en el funcionamiento de los juegos de mesa, principalmente con la diferenciación entre entes inertes y las entidades que los controlan [14]. En las próximas secciones estableceremos el rol de cada módulo en esta analogía.

Módulo Game.Entities

Para la implementación de este módulo se ha utilizado la herencia para representar la relación entre los agentes y las entidades. Estas últimas registran su posición actual para agilizar algunas consultas y se encargan de crear y destruir el modelo, ya sea 2D o 3D, que las representa. Los agentes expanden estas características añadiendo estadísticas y funciones para interactuar con ellas, así como conservando su estado actual en cuanto a su salud u otras alteraciones de las estadísticas base. Las características de entidades y agentes se cargan desde su correspondiente fichero utilizando la tecnología de scriptable objects.

También se ha implementado un componente adicional que es posible incorporar a cualquier entidad o agente. Este componente puede ser utilizado por las acciones para iniciar nuevas animaciones y consultar si éstas han sido completadas.

Como se puede comprobar, la lógica implementada en las entidades y agentes es mínima y si limita exclusivamente a hacer un seguimiento de su estado actual. Esto se debe a que, al contrario que en la arquitectura estándar de Unity, en este caso los entes son totalmente pasivos. Podemos compararlos con las figuras de un juego de mesa, ya que son inertes y nunca realizarán acciones por su cuenta, sino que requieren de la intervención de terceros para modificar su estado.

Módulo Game.Bords

Siguiendo el diseño, el tablero ha sido implementado a través de un sistema de capas: una para los agentes, otra para las entidades y una última para el terreno. Las capas de las entidades y agentes comparten su implementación y almacenan un array de dimensión dos de referencias a sus respectivos elementos. Esto implica la imposibilidad de situar dos agentes o dos entidades en la misma casilla, pero permite que, por ejemplo, un agente se siente en una silla. La capa de terreno, sin embargo, se limita a almacenar su tipo. Tanto las dimensiones del tablero como las características de cada tipo de terreno se cargan de sendos ficheros de los respectivos tipos de scriptable objects.

Se ha conseguido adaptar el paquete de Unity que proporciona soporte a cuadrículas de assets a las necesidades de la extensión, pese a que está enfocado al uso 2D y no al 3D. Esto permite definir unos ficheros con reglas sencillas que determinan si, por ejemplo, se debería utilizar el asset que representa la esquina de un muro o el que representa una simple segmento. Cuando el tablero cambia el tipo del terreno de una casilla, se actualizan de manera apropiada sus vecinas en tiempo real.

También se proporciona una estructura de datos en la que almacenar toda la información contenida por un mapa, facilitando su serialización y permitiendo su carga. Tecnologías como la generación procedural utilizan esta estructura para almacenar la salida de los algoritmos de creación de mazmorras, especificando el tipo de terreno presente en cada casilla.

El tablero, como cabría esperar, también es muy similar a su contraparte en un juego de mesa, especialmente en el caso de aquellos juegos en los que se pueden combinar diferentes piezas para formar una configuración personalizada. Su funcionalidad se limita a comprobar e impedir aquellos intentos de modificar el tablero a un estado incoherente.

Módulo Game.Turns

El gestor de turnos gira en torno a una lista enlazada y a un sistema de pasos o ticks. Se comienza en el tick cero y, cuando se resuelve el turno de un agente, se añade su próximo turno en el tick resultante de sumar al actual el tiempo de recarga de turno del agente. Después, se inserta ese nuevo turno en la posición correspondiente de la lista enlazada, empezando por detrás para mayor eficiencia.

$$\text{NuevoTurno.Tick} = \text{ViejoTurno.Tick} + \text{ViejoTurno.Agente.TiempoRecargaTurno}$$

Podemos exemplificar el funcionamiento con un caso hipotético. Supongamos que el turno del agente A es en el tick 100, el del agente B en el 150 y el del agente C en el 250. Como le toca actuar al agente A nos encontramos en el tick 100 y, después de que éste actúe, crearemos su nuevo turno. Si suponemos que su tiempo de recarga de turno son 100 ticks, su nuevo turno será en el tick 200 y deberá ser insertado entre los turnos de los agentes B y C. Para continuar, solo debemos eliminar el antiguo turno de la lista y fijarnos en el nuevo primer elemento, en este caso el turno del agente B.

Módulo Game.Actions

Las acciones son los elementos encargados de modificar el estado del juego y están basadas en el patrón de diseño Command [15]. Pese a que solo exponen un método público para su procesamiento, internamente se diferencia entre una función que solo es procesada la primera vez y otra que es procesada cuantas veces sean necesarias hasta terminar. Las acciones pueden sobrescribir una o ambas de estas funciones y si cualquiera de sus ejecuciones retorna un valor booleano de verdad, se interpretará que la acción ha sido completada y ha finalizado. La lógica de algunas acciones, como la que crea nuevos agentes o entidades, se encuentra exclusivamente en su ejecución inicial. Otras acciones, como la de movimiento, pueden necesitar muchas iteraciones para ser completadas.

También disponemos de otra categoría: las acciones que están compuestas a su vez de otras. Diferenciamos la acción de procesamiento secuencial y la de procesamiento en paralelo. Cada una gestiona de manera distinta la propagación de las peticiones de procesamiento y la existencia de

ambas es lo que permite gestionar los comportamientos más complejos. Es posible, por ejemplo, crear un hechizo en el que un proyectil se divida en varios y cada uno golpee a un enemigo, le empuje hacia atrás y luego tenga una probabilidad de poner a ese agente bajo el control del jugador.

Las acciones serían comparables a las cartas de muchos juegos de mesa. Son las que definen los efectos que deben ser aplicados y, tras resolverse, son descartadas. Numerosos juegos permiten que varias cartas sean jugadas al mismo tiempo, estableciendo reglas para su resolución conjunta. La principal diferencia es que en este caso las acciones se resuelven a sí mismas, ya que no solo contienen los datos sobre lo que debe suceder, sino que disponen de la lógica para aplicar dichos efectos sobre el tablero y los agentes y entidades.

Módulo Game.Controllers

Ya que los agentes no disponen de la lógica necesaria para resolver sus propios turnos, utilizamos a los controladores para generar las acciones correspondientes en su nombre. Existen dos controladores distintos: el que gestiona los agentes bajo el control del jugador y el que se encarga de los demás. Al controlador del jugador se le inyecta la entrada del usuario y, cuando se le solicita resolver el turno de un agente, utiliza esa información para determinar la acción que se crea. El controlador de los NPCs (Non-Player Characters) analiza las estadísticas del agente, así como su entorno, para elegir la acción que mejor represente el comportamiento y características de éste.

Esta arquitectura facilita enormemente la gestión de la entrada del jugador, ya que es mucho más sencillo determinar a qué agente va dirigida en cada momento y aplicarla de manera adecuada sin causar problemas en la sincronización y coordinación de los diferentes sistemas del juego.

Los controladores representan a los jugadores de un juego de mesa, aunque en este caso solo uno sería humano, ya que son quienes toman las decisiones que determinan la evolución de la partida. El controlador que analiza las características de un agente representa en realidad a un número indeterminado de jugadores, ya que si los agentes pertenecen a facciones enfrentadas defenderá en cada turno intereses distintos, como un jugador de ajedrez controlando las piezas de ambos colores.

Módulo Game.Core

Para combinar todos los módulos anteriores y crear un sistema de juego coherente solo nos falta un elemento central que los coordine. Este módulo únicamente contiene un gestor central que define un bucle de juego. El bucle de juego realiza los siguientes pasos:

1. Comprueba si la acción actual ha finalizado. Si no lo ha hecho:
 - 1.1. Procesa la acción, pudiendo ésta finalizar en ese momento o no.
 - 1.2. Da por finalizado el bucle actual.
2. Crea una nueva acción en paralelo.

3. Hasta que se repita el mismo dos veces, para al agente actual del gestor de turnos:
 - 3.1. Comprueba si el controlador, de requerir una entrada, la ha recibido. Si no:
 - 3.1.1. Salta al paso 4.
 - 3.2. Solicita una acción al controlador correspondiente.
 - 3.3. Finaliza el turno del agente actual, haciendo avanzar al gestor de turnos.
 - 3.4. Procesa la acción generada por el controlador.
 - 3.5. Comprueba si la acción del agente ha finalizado. Si no:
 - 3.5.1. La añade a la acción en paralelo.
 - 3.6. Comprueba si la acción permite continuar la parallelización. Si no:
 - 3.6.1. Salta al paso 4.
4. Establece la acción en paralelo generada como la acción actual.

Si tuviésemos que finalizar la metáfora de los juegos de mesa, este módulo cumpliría el papel del maestro de juego o gamemaster. En algunos juegos es necesaria esta figura para coordinar al resto de elementos e indicar a los jugadores los momentos en los que pueden intervenir.

Módulo Game

Pese a que el módulo Game no contiene más elementos que sus submódulos, es interesante analizar qué contienen todos esos submódulos en su conjunto y por qué se ha creado esta barrera lógica. Comprobamos que con los elementos definidos hasta ahora ya podemos representar toda la lógica de una partida completa, con todos los sistemas que la componen. Lo que nos falta es convertir esta representación en un videojuego, ya que actualmente solo existe como estructuras de datos accesibles solo mediante la depuración del código.

Esto lo conseguiremos por un lado construyendo los módulos ajenos a Game: coordinarán el bucle de juego definido con el otorgado por Unity, capturarán e injectarán la entrada del usuario en el controlador correspondiente y permitirán explorar y mostrar la información contenida en una interfaz de usuario. Por otro lado, dentro del entorno de desarrollo de Unity, conectaremos los módulos existentes con assets que los representarán, ya sea en 2D o en 3D. Esto permitirá visualizar el tablero, los agentes y entidades y las acciones.

Módulo Input

Para capturar la entrada del usuario y procesarla de manera adecuada se utiliza el módulo Input. Solo dispone de un script, el cual define una serie de callbacks que son ejecutados en respuesta a diferentes comandos de entrada. El grueso de la configuración se realiza dentro del editor de Unity, a través de unos ficheros en los que se establecen acciones y los diferentes controles que las ejecutan.

De esta manera es posible asignar cómodamente tanto las flechas del teclado como la cruceta de un mando al control de, por ejemplo, el movimiento. Los callbacks definidos se comunican directamente con el controlador de los agentes del jugador, inyectando al inicio del bucle la entrada de ese fotograma y haciendo una limpieza al final.

Actualmente los controles están centrados en el teclado y ratón, permitiendo el movimiento, el ataque y el uso del ratón como puntero. Sin embargo, resulta extremadamente simple añadir nuevos controles, ya que solo es necesario definir una nueva función y vincularla a una entrada, sin necesidad de modificar o estudiar el resto del código.

Para facilitar el funcionamiento de la arquitectura y evitar problemas de sincronización en el bucle de juego centralizado, se ha optado por indicar a Unity que la captura de la entrada de usuario se realiza de manera explícita desde código. De esta manera tenemos el control necesario para encajar esta etapa para que se ejecute en el momento exacto que definimos en el diseño.

Módulo Core

Finalmente, para que todos los módulos anteriores funcionen dentro del motor de Unity debemos vincular el bucle de Unity y sus fotogramas con el definido en el módulo Game. Para ello utilizamos las funciones Start y Update, que son ejecutadas por Unity como parte del ciclo de vida de todas las clases que heredan de MonoBehaviour, una clase provista por el motor.

Siguiendo el diseño, en la función Update se captura la entrada del usuario para después delegar la lógica del bucle de juego en el módulo Game.Core y su gestor. La aparente dualidad de bucles de juego entre los módulos Game.Core y Core proviene de la diferenciación entre la lógica pura de juego, contenida en el primero, y la necesidad de encajar dentro del entorno de Unity, su arquitectura y sus métodos de captura de entrada de usuario, resuelta por el segundo.

Con este módulo completamos la lógica necesaria para la ejecución de un prototipo completo y, a partir de aquí, podemos aprovechar el orden en el desarrollo para explorar la experiencia de añadir nuevos módulos a esta arquitectura.

Módulo Game.Generation

Para dar soporte a una importante característica del roguelike, la generación procedural, se ha implementado el módulo Game.Generation. Su funcionamiento aprovecha la existencia de definiciones de tablero en el módulo Board. Teniendo en cuenta las características especificadas, el generador crea un conjunto de salas superpuestas en el centro del mapa, que luego se repelen entre sí para quedar separadas a una distancia razonable. Después, se calcula el árbol recubridor mínimo para garantizar la conexividad del mapa y se conectan las salas combinando pasillos rectos y en esquina. Se pueden visualizar los pasos del algoritmo en el apéndice B, a partir de la figura B.11.

Módulo UI

También habíamos dejado pendiente de implementar un módulo externo a la lógica de Game. En este caso, el módulo UI debía otorgar herramientas que permitiesen al jugador explorar las características de diferentes elementos del juego.

La implementación ha requerido modificar el módulo Input para que los callbacks también incluyan llamadas al gestor del módulo UI, así como integrarlo dentro de la lógica de la función Update del módulos Core, convirtiéndose en el último paso realizado. Se han creado diversas herramientas para explorar el movimiento por las casillas y las estadísticas de los agentes, indicar el agente bajo control en cada momento e incluso para controlar la cámara y que ésta siga a los agentes. Crear nuevos elementos para la interfaz es muy sencillo: se realiza el diseño de la interfaz en el entorno de Unity, se crea un nuevo script en el paquete con la lógica asociada y se integran en el gestor de UI las llamadas a este nuevo script, facilitando la información que sea necesaria sobre la posición del ratón u otros tipos de interacción del jugador.

Módulo Utils

El módulo Utils se ha desarrollado en paralelo al resto de módulos y ha servido para incorporar nuevos tipos a los que Unity no da soporte por defecto, como son las casillas o las dimensiones. También se han definido clases para simplificar la gestión de errores y warnings, así como para extender el funcionamiento del generador de números aleatorios.

Este módulo es accedido por la mayoría del resto de módulos y, en este caso, no se ha buscado lograr ningún tipo de desacople. Esto se debe a que su filosofía de diseño es extender las facilidades que provee Unity y utilizarlas como si hubiesen formado parte de la versión oficial del motor.

4.3. Empaquetamiento y registro de la extensión

El proceso para empaquetar la extensión es trivial. La incorporación de una extensión a Unity es, en definitiva, copiar una carpeta dentro de la raíz del proyecto, aunque con las comodidades de hacerlo a través de un gestor en lugar de manualmente. Una vez comprobado el correcto funcionamiento del software implementado, éste ha sido estructurado siguiendo las indicaciones provistas por Unity, con un directorio con el nombre del creador conteniendo otro con el nombre de la extensión en sí.

Para comprobar que se cumplen los criterios básicos para la publicación de la extensión es posible descargar un paquete en el editor que hace de validador automático. Utilizando esta herramienta comprobamos que se cumplen todos los requisitos. Los dos avisos existentes se derivan del uso del paquete de Unity de gestión de assets en tablero, que se encuentra todavía en acceso anticipado.

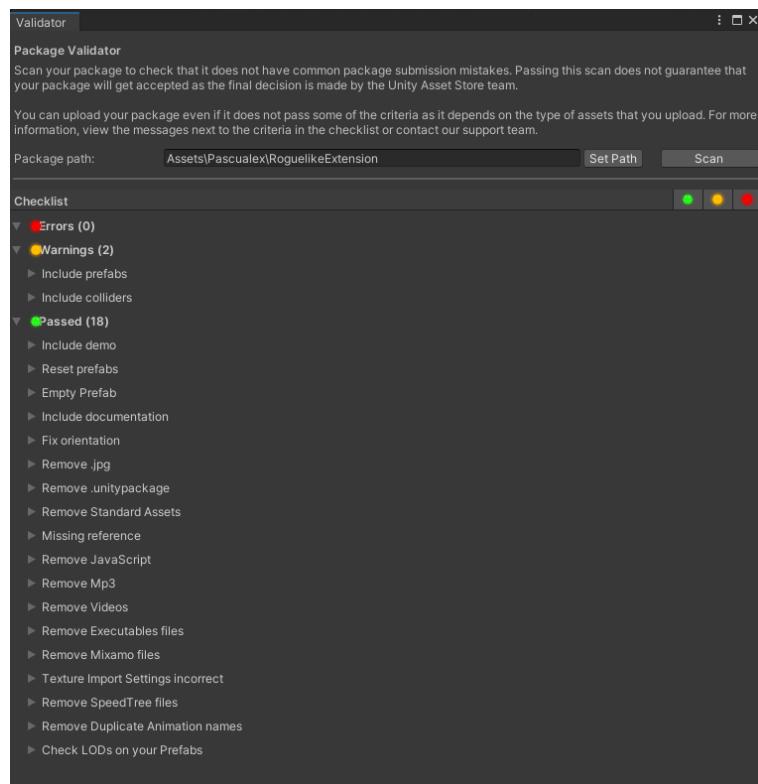


Figura 4.1: Validador de extensiones de Unity.

Una vez el paquete está validado, es necesario crear una cuenta en el Asset Store Publisher de Unity [16]. Con la cuenta creada podemos subir el paquete, aportar todos los metadatos necesarios e indicar si se trata de un paquete gratuito o de pago.

Package name ↑	Status	Version	Price	Average rating	Size
Roguelike tools	Draft	1.0	\$4.99	0/5 (0 ratings)	6.1 MB
Uploaded packages					
Editor version					
2020.3.10f1					

Figura 4.2: Borrador de la extensión en Unity Asset Store Publisher.

El último paso restante es solicitar su aprobación a Unity, lo que requiere esperar a su revisión manual, llevar a cabo los cambios solicitados y aportar métodos de contacto para el soporte y mantenimiento de la extensión.

PRUEBAS

Con la implementación terminada, se vuelve necesaria la creación y el análisis de las pruebas. Éstas cumplen esencialmente dos propósitos: otorgarnos mayores garantías del correcto funcionamiento de la extensión y facilitarnos su expansión futura tanto “oficial” como por parte de cada desarrollador. Permiten comprobar fácilmente que las modificaciones no han interferido con la funcionalidad previa y proporcionan interesantes ejemplos en los que basarse para lograr integrar los distintos sistemas.

5.1. Tests unitarios

Unity integra una solución propia como framework de tests y es la que hemos utilizado para desarrollar las pruebas unitarias. En cada uno de los paquetes ponemos a prueba sus diferentes funcionalidades de la manera más modular y aislada del resto posible. Para ello definimos scripts dedicados exclusivamente a esta tarea, que podemos ejecutar cómodamente desde el editor de Unity. Estos tests servirán para alertar siempre que la modificación o expansión de un módulo incorpore algún cambio o error que interfiera con la funcionalidad existente previamente.

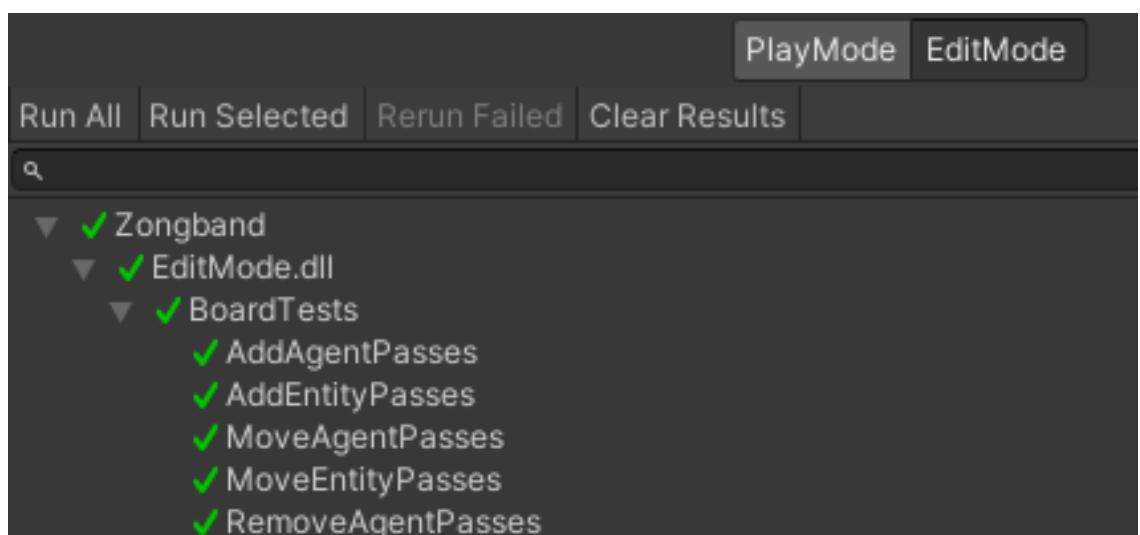


Figura 5.1: Tests del módulo Game.Borads ejecutándose en el framework de Unity.

5.2. Tests de integración

Utilizando el mismo framework hemos definido también tests de integración. Estos tests, a diferencia de los unitarios, comprueban la funcionalidad conjunta de varios módulos al mismo tiempo. Con ellos es algo más complicado llegar a la fuente exacta de un error, pero permiten probar lógicas más complejas que también son indispensables para el correcto funcionamiento de la extensión.

La herramienta de tests de Unity diferencia entre dos tipos fundamentales de pruebas: aquellas que se deben realizar en una escena en ejecución (EditMode) y aquellas que no (PlayMode). Por ejemplo, aquellos elementos que utilicen las funciones internas Start y Update necesitan probarse en PlayMode para que dichas funciones sean ejecutadas correctamente. Por su naturaleza más cercana al producto final, los tests de integración generalmente necesitan ejecutarse en PlayMode.

5.3. Videojuego implementado con la extensión

Más allá de las pruebas más formales y automatizadas, es de gran interés para este proyecto realizar una prueba manual que nos permita ponernos en el papel de un usuario de esta extensión. De esta manera, no solo podremos comprobar la calidad de la experiencia del desarrollo, sino que también estudiaremos la correcta integración de todos los sistemas proporcionados.

En el apéndice B encontramos las capturas de este pequeño proyecto, mientras que esta sección se centra en la descripción de las etapas de su desarrollo.

Estructuración de los gestores y los controladores

El primer paso necesario para poner en funcionamiento los principales sistemas de la extensión es crear dentro del editor de Unity la estructura de gameobjects necesaria. Éstos tienen asociados los diferentes scripts de los gestores y los controladores y pueden acceder a las funciones especiales de Unity que son llamadas como parte de la arquitectura de éste.

Aunque esta estructura puede ser creada manualmente, hemos optado por importarla desde la extensión, lo que facilita el trabajo y ayuda a reducir errores.

Configuración del tablero

El siguiente paso que hemos seguido es el de configurar el tablero de juego. Para ello, debemos crear un scriptable object de tipo tablero y especificar sus dimensiones. También es necesario especificar diferentes tipos de terreno con los que construir el tablero. En este caso debemos crear tanto un scriptable object de terreno como un objeto nativo de Unity para el uso de casillas.

En nuestro caso hemos diseñado algunos modelos 3D en MagicaVoxel, después los hemos configurado en el editor de Unity para formar una casilla con modelos específicos para esquinas abiertas y cerradas y por últimos hemos creado el scriptable object incorporando todos los elementos anteriores.

Creación de los agentes y las entidades

Con los sistemas de juego en funcionamiento y el tablero configurado, el único elemento básico restante son los agentes y las entidades. La complejidad de este paso dependerá enormemente de la variedad que busquemos, por lo que nosotros nos limitaremos a media docena de elementos.

Hemos utilizado las formas primitivas que otorga Unity para su representación y definido sus estadísticas y comportamiento en sendos scriptable objects de agentes y entidades.

Diseño de mapas personalizados

Con todos los elementos necesarios ya disponibles, solo necesitamos instanciarlos. Para la realización de pruebas nos interesa tener un control absoluto, por lo que por el momento no utilizaremos la generación procedural de contenido. En su lugar, definimos funciones para crear niveles de ejemplo, utilizando las interfaces de los diferentes módulos para diseñar el mapa, instanciar a los agentes y a las entidades, especificar cuáles se encuentran bajo el control del jugador y ejecutar las acciones adicionales que la prueba requiera.

Ejecución de casos de ejemplo

Algunos de los casos comprobados satisfactoriamente mediante este sistema han sido:

- El movimiento de los agentes por las casillas.
- La imposibilidad de movimiento a una casilla ocupada.
- El intercalado de turnos entre un agente del jugador y un NPC.
- El ajuste a diferentes velocidades de turno entre agentes.
- El ataque a enemigos adyacentes por parte del jugador y de los NPCs.
- La mecánica de muerte de los agentes y el funcionamiento sin agentes del jugador.
- La generación procedural de una mazmorra con assets propios.

Creación procedural de contenido

Finalmente, se ha utilizado la generación procedural de contenido para estudiar un producto más cercano a los resultados que se buscan alcanzar con la extensión. Se ha experimentado con diferentes números de habitaciones, distintos tamaños máximos y mínimos de las salas y varias dimensiones del mapa.

CONCLUSIONES Y TRABAJO FUTURO

Tras el trabajo realizado, analizaremos brevemente los resultados para extraer conclusiones y determinar la dirección en la que tendría que continuar el desarrollo de trabajo futuro.

6.1. Conclusiones

La principal pregunta que debemos hacernos es si hemos cumplido con los objetivos que marcamos al inicio del proyecto. El diseño y la implementación nos han revelado que no es trivial llegar a una arquitectura óptima para la estructura del género roguelike, principalmente por sus grandes diferencias con el enfoque proporcionado por Unity. Pese a ello, actualmente los distintos sistemas de la arquitectura ofrecen una solución sencilla y funcional que da soporte a las principales características del género. Las buenas prácticas seguidas han resultado en unas bases robustas y flexibles sobre las que poder desarrollar con éxito videojuegos roguelike. Por su parte, Unity ha demostrado ser la plataforma correcta para el desarrollo. En la prueba ya hemos experimentando con algunas de sus herramientas modernas avanzadas y el sistema de extensiones ha funcionado sin contratiempos.

Sin embargo, nos estaríamos engañando si pensásemos que con los sistemas presentes actualmente es posible desarrollar un videojuego completo. Las mecánicas y herramientas proporcionadas por la extensión deben ser combinadas con un extenso trabajo por parte de los desarrolladores para incorporar sus propios sistemas y peculiaridades al videojuego. Por suerte, el desacoplamiento entre los módulos ha logrado su objetivo de facilitar la implementación de nuevas funcionalidades, lo cual ha sido puesto a prueba desarrollando dos de los módulos en iteraciones posteriores a la puesta en marcha de la arquitectura y los sistemas más esenciales.

También podemos comprobar que se trata de una extensión apta solo para perfiles técnicos con conocimientos de programación. Al no proporcionarse una interfaz gráfica, la única manera de acceder a la funcionalidad es a través del uso de los paquetes de código proporcionados. Esto estaba previsto en el diseño, pero podría suponer una barrera para algunos desarrolladores.

En definitiva, la extensión cumple correctamente con los objetivos marcados al inicio del proyecto, pero éstos dejan margen para su mejora y expansión.

6.2. Trabajo futuro

Dada la naturaleza modular de la extensión, lo más interesante es definir el trabajo futuro como potenciales módulos que se podrían implementar dando soporte a nuevos sistemas.

Módulo de habilidades

El sistema actual de acciones permite realizar acciones compuestas tan complejas como desee el desarrollador. Sin embargo, resultaría mucho más conveniente poder definir directamente conjuntos de estas acciones mediante ficheros de scriptable objects. Éstos conformarían las habilidades o hechizos de los agentes, con sus correspondientes costes y requisitos asociados.

Módulo de inventario y equipamiento

Muchos juegos del género incorporan la gestión del inventario y el equipamiento, por lo que sería de gran utilidad que la extensión los soportase oficialmente. El inventario debería permitir definir la capacidad de almacenamiento de entidades y agentes y el traspaso de objetos de unos a otros. Al equiparse, se tendrían que modificar de manera apropiada las estadísticas de los agentes.

Módulo de diálogos

La capacidad de interactuar y conversar con otros agentes para recibir misiones, recabar información, comerciar o desescalar un conflicto, añadiría muchísima variedad y riqueza a los títulos desarrollados. Debería funcionar mediante una máquina de estados y soportar la serialización.

Módulo de gestión de niveles

Pese a que se da soporte al tablero y a la generación procedimental, una capa adicional para el movimiento entre distintos niveles y partes del mapa facilitaría enormemente la gestión del juego. Soportar la serialización del estado actual de un nivel haría mucho más sencillo guardar y cargar partidas, lo que resulta fundamental en cualquier juego donde éstas sean de larga duración.

Interfaz gráfica

Además de la incorporación de nuevos módulos, se podría plantear dar soporte a otras maneras de interactuar con las funcionalidades ya existentes, ofreciendo una interfaz gráfica como alternativa al uso directo de los paquetes de código. Por ejemplo, poder diseñar de manera visual un nivel hecho a mano, determinando las casillas de su tablero y la disposición de sus agentes y entidades, agilizaría en gran medida este trabajo.

BIBLIOGRAFÍA

- [1] "RogueBasin, comunidad de desarrolladores de roguelikes." <http://www.roguebasin.com/>.
- [2] M. Mahardy, "Roguelikes: The Rebirth of the Counterculture," *IGN*, 2014.
<https://www.ign.com/articles/2014/07/04/roguelikes-the-rebirth-of-the-counterculture>.
- [3] "Listado de videojuegos desarrollados en Unreal Engine."
https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games. Accedido el 22/05/2021.
- [4] Epic Games, "Unreal Engine 5 early access presentation."
<https://www.unrealengine.com/en-US/unreal-engine-5>. Accedido el 06/06/2021.
- [5] Unity Technologies, *Documentación de Unity*. <https://docs.unity3d.com/>.
- [6] "Listado de videojuegos desarrollados en Unity."
https://en.wikipedia.org/wiki/List_of_Unity_games. Accedido el 23/05/2021.
- [7] Unity Technologies, "Unity Forum." <https://forum.unity.com/>. Accedido el 29/05/2021.
- [8] Unity Technologies, "Unity Learn." <https://learn.unity.com/>. Accedido el 29/05/2021.
- [9] Unity Technologies, "Made With Unity." <https://unity.com/madewith>. Accedido el 29/05/2021.
- [10] L. Bonin, "Unity Technologies acquires Bolt."
<https://ludiq.io/blog/unity-acquires-bolt>. Accedido el 30/05/2021.
- [11] J. Albahari and E. Johannsen, *C#8.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2020.
- [12] B. Wagner, "Nullable reference types," tech. rep., Microsoft Corporation, 2020.
<https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references>.
- [13] Microsoft Corporation, "Code style rule options."
<https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/code-style-rule-options>.
- [14] B. Nystrom, "A turn-based game loop," *stuffwithstuff*, 2014.
<https://journal.stuffwithstuff.com/2014/07/15/a-turn-based-game-loop/>.
- [15] B. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.
- [16] Unity Technologies, "Unity Asset Store Publisher."
<https://unity3d.com/asset-store/sell-assets>. Accedido el 10/06/2021.

APÉNDICES

FRAGMENTOS DE INTERÉS DEL CÓDIGO

A continuación, se muestran los fragmentos de código críticos para el funcionamiento del bucle de juego. Al ser la capa más superior de la lógica, se pueden comprender aún sin su contexto completo.

Código A.1: Función Update de la clase CoreManager del módulo Core, encargada de sincronizar la arquitectura de Unity con los módulos de la extensión.

```
1 // Unity Method
2 private void Update()
3 {
4     InputManager.ProcessInput();
5     GameManager.GameLoop();
6     UIManager.Refresh();
7     InputManager.ClearInput();
8 }
```

Código A.2: Función OnMouseLeftClick de la clase InputManager del módulo Input. Ejemplifica un callback a la captura de la entrada y la propagación de la información.

```
1 // Unity Method
2 private void OnMouseLeftClick()
3 {
4     UIManager.HandleMouseLeftClick();
5     PlayerController.PlayerInput = new PlayerInput(MouseTile, false, true);
6 }
```

Código A.3: Función GameLoop de la clase GameManager del módulo Game.Core, gestora del procesamiento de la acción actual y de la petición de creación de nuevas acciones.

```
1 public void GameLoop()
2 {
3     if (CurrentAction.IsCompleted) CurrentAction = ProcessTurns();
4     else CurrentAction.Process();
5 }
```

Código A.4: Función ProcessTurns de la clase GameManager del módulo Game.Core. Solicita a los controladores las acciones del agente correspondiente en cada momento, construyendo la acción en paralelo que se ejecutará en las siguientes iteraciones del bucle de juego.

```

1 private Action ProcessTurns()
2 {
3     var newParallelAction = new ParallelAction();
4     var processedAgents = new HashSet<Agent>();
5     var ctx = new Action.Context(TurnManager, Board, AgentPrefab, EntityPrefab);
6
7     Agent? agent;
8     while (((agent = TurnManager.GetCurrent()) != null) && !processedAgents.Contains(agent))
9     {
10        Action? agentAction;
11        if (agent.IsPlayer) agentAction = PlayerController.ProduceAction(agent, ctx);
12        else agentAction = AIController.ProduceAction(agent, ctx);
13        if (agentAction == null) break;
14
15        agentAction.Process();
16        if (!agentAction.IsCompleted) newParallelAction.Add(agentAction);
17
18        processedAgents.Add(agent);
19        TurnManager.Next();
20        if (!(agentAction is MovementAction) && !(agentAction is NullAction)) break;
21    }
22
23    return newParallelAction;
24 }
```

Código A.5: Funciones Refresh y HandleCtrlMouseLeftClick de la clase UIManager del módulo UI. Ambas propagan la lógica al resto de componentes de la interfaz: la primera su actualización y la segunda un ejemplo de entrada del usuario.

```

1 public void Refresh()
2 {
3     TileHighlighter.Refresh();
4     PlayerHighlighter.Refresh();
5     AgentInspector.Refresh();
6
7     CameraController.Refresh();
8     UpdateMouseTile();
9 }
10
11 public void HandleCtrlMouseLeftClick()
12 {
13     AgentInspector.LockAgent();
14 }
```

Código A.6: Función Process de la clase Action del módulo Game.Actions. Divide la lógica en ProcessStart y ProcessUpdate, que deben ser implementadas por herencia.

```
1 public void Process()
2 {
3     if (IsCompleted) return;
4     if (!HasStarted)
5     {
6         IsCompleted = ProcessStart();
7         HasStarted = true;
8     }
9     else IsCompleted = ProcessUpdate();
10}
11
12 protected virtual bool ProcessStart()
13 {
14     return false;
15 }
16
17 protected virtual bool ProcessUpdate()
18 {
19     return true;
20 }
```


CAPTURAS DEL VIDEOJUEGO

IMPLEMENTADO CON LA EXTENSIÓN

En este apéndice se muestran las capturas del pequeño videojuego realizado con las herramientas desarrolladas. Se ilustran así algunos de los sistemas y mecánicas implementados en la extensión.

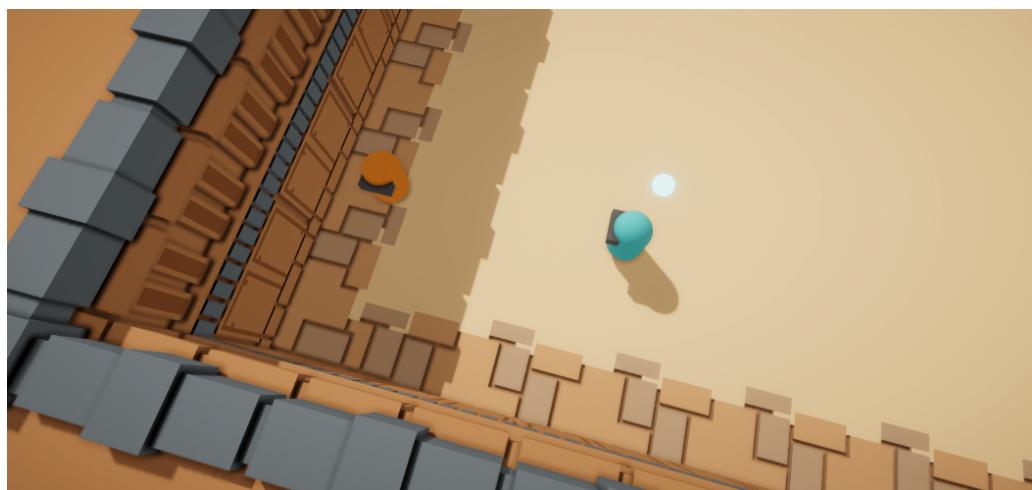


Figura B.1: Intercalado de turnos 1. El jugador (cápsula azul) ve a un enemigo (cápsula naranja) y decide huir de él para evitar el combate.

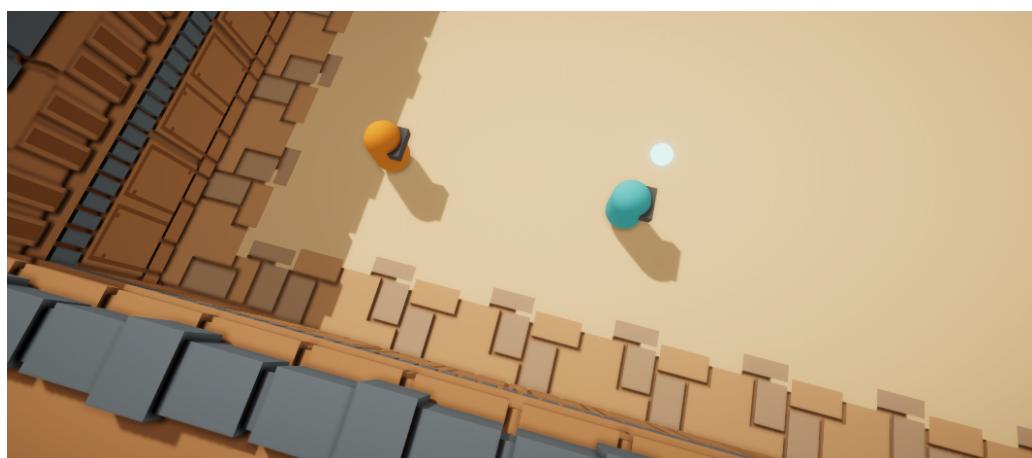


Figura B.2: Al moverse el jugador, el enemigo también obtiene un turno, que utiliza para perseguirle. Ambos movimientos son visualizados en paralelo y, al terminar, vuelve a ser el turno del jugador.



Figura B.3: Interfaz de usuario 1. El jugador (cápsula azul) está indicado por el PlayerHighlighter (esfera de luz). Un enemigo (cápsula morada) está siendo analizado por el AgentInspector (esquina superior izquierda). El TileHighlighter muestra que es posible moverse a la casilla entre ambos.



Figura B.4: Interfaz de usuario 2. El jugador posee al enemigo y es ahora capaz de controlarlo, por lo que todos los módulos de la interfaz son actualizados. Se muestran ahora las estadísticas del agente azul, que también ha sido dañado en el combate y ya no es hostil.

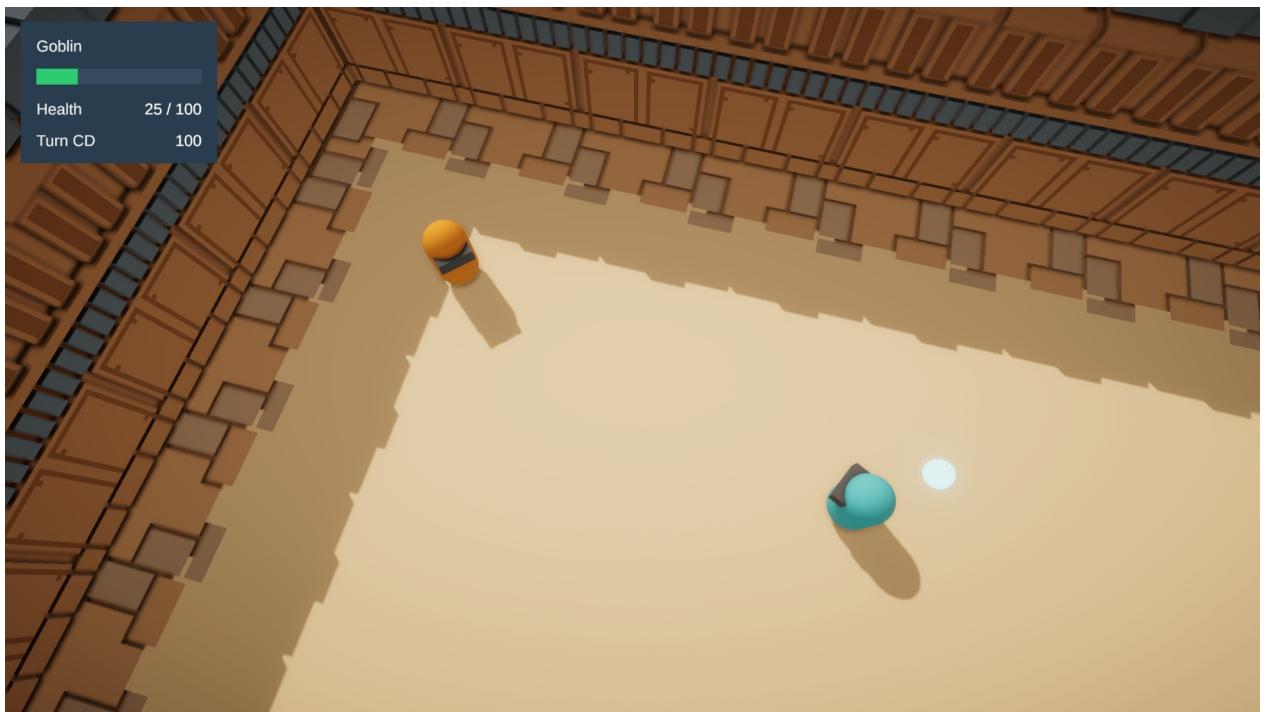


Figura B.5: Eliminación de un agente 1. Un enemigo bajo de vida (cápsula naranja) es acorralado por el jugador (cápsula azul).

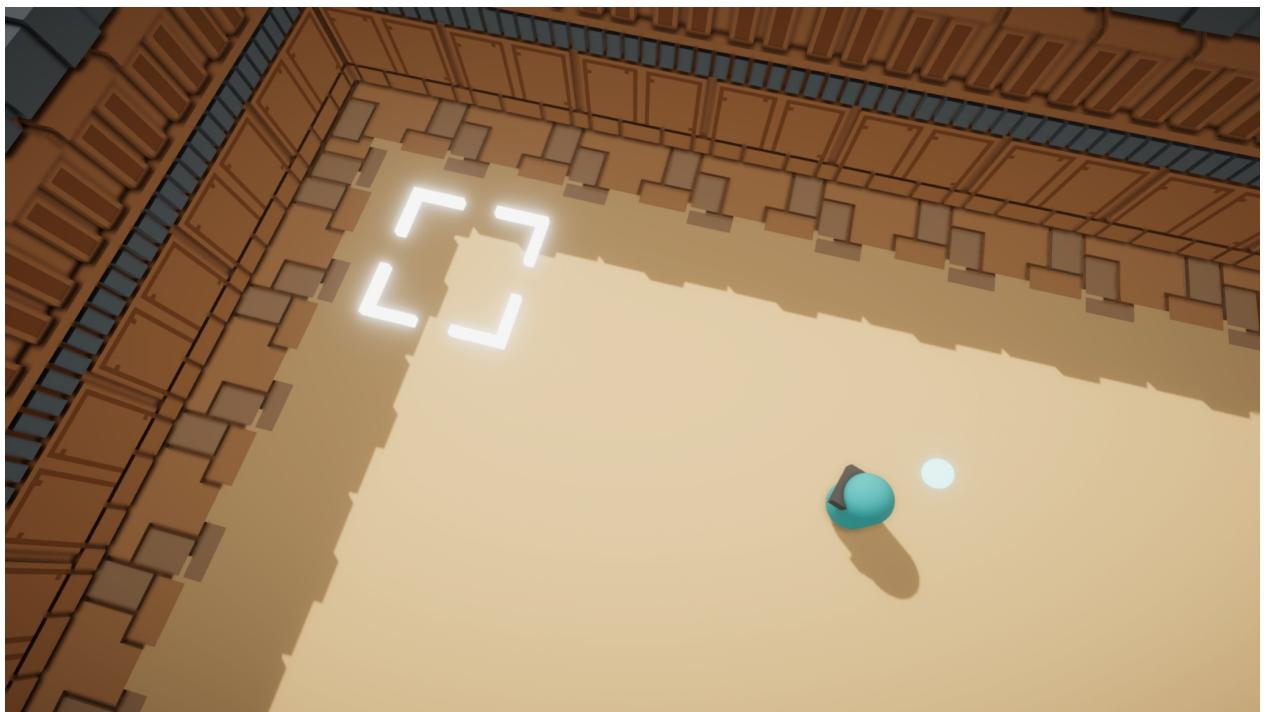


Figura B.6: Eliminación de un agente 2. Tras recibir un último disparo, el agente muere y es eliminado correctamente del tablero y el gestor de turnos, dejando su casilla libre.

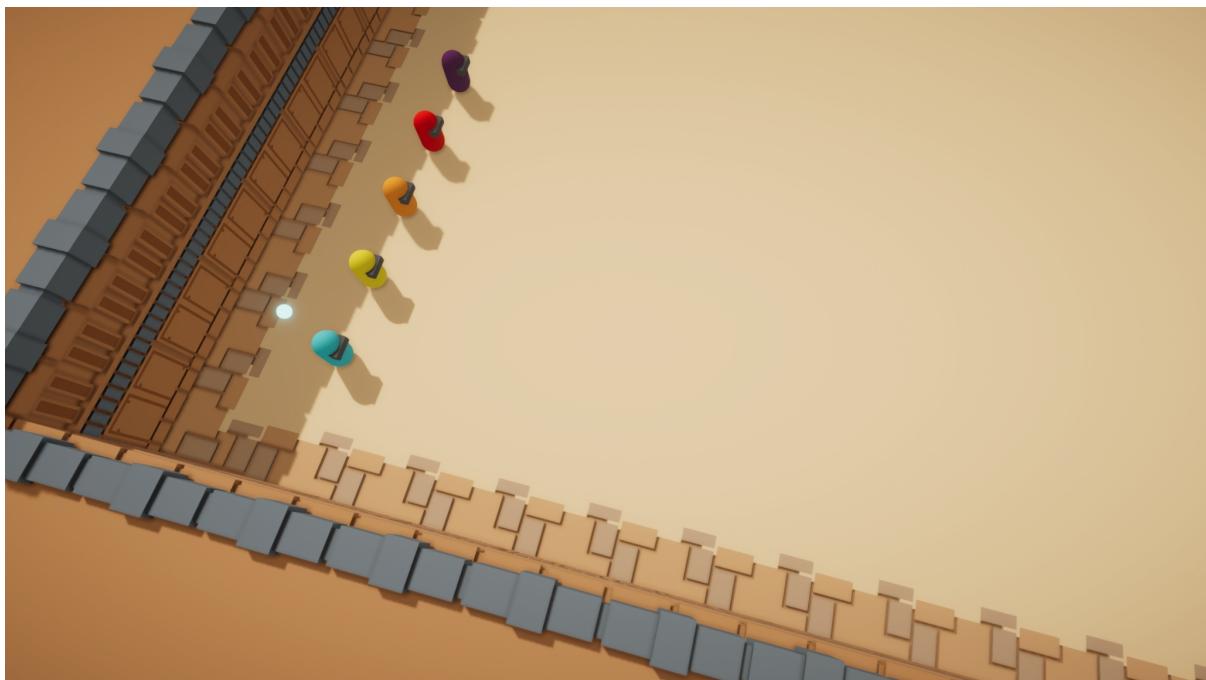


Figura B.7: Agentes de distintas velocidades 1. Se alinean agentes con diferentes estadísticas.

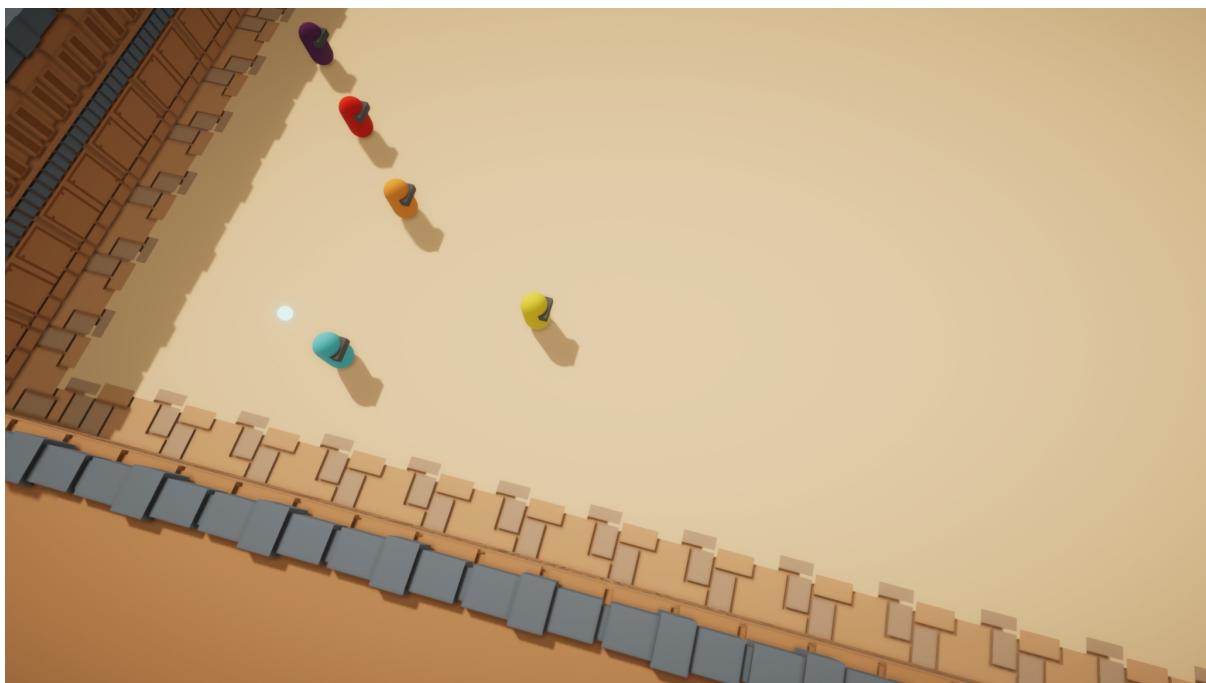
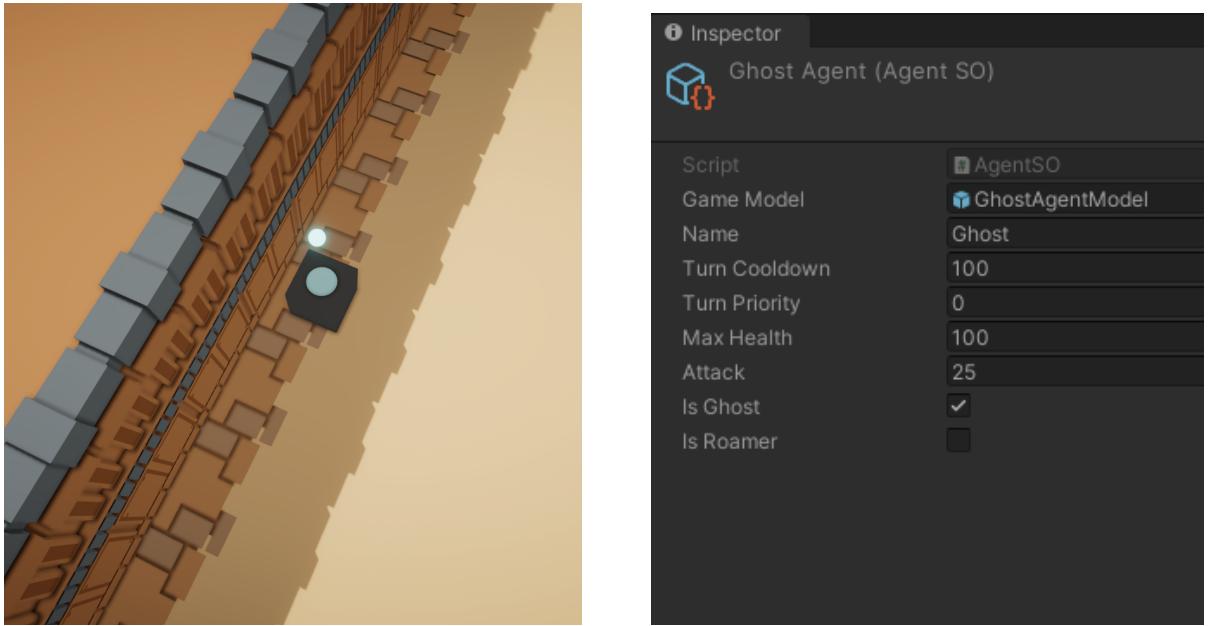


Figura B.8: Agentes de distintas velocidades 2. Observamos la situación después de que el jugador (cápsula azul), que tiene un retraso entre turnos de 100 ticks, se mueva dos veces. El agente amarillo tiene un retraso de 50, por lo que ha tenido cuatro turnos. El agente naranja también tiene 100 y se ha movido lo mismo que el jugador. El agente rojo tiene 200 ticks entre turnos, por lo que se ha movido solo una vez. El comportamiento del agente morado le hace estar quieto cuando no está en combate, aunque su retraso es de 100 ticks.



(a) Aplicación del scriptable object.

(b) Definición del scriptable object.

Figura B.9: A la izquierda, el jugador (cápsula azul pálido) se encuentra superpuesto con una entidad (caja negra) ya que cuenta con la propiedad *fantasma*. A la derecha, el fichero de datos (scriptable object de agente) que define sus estadísticas.

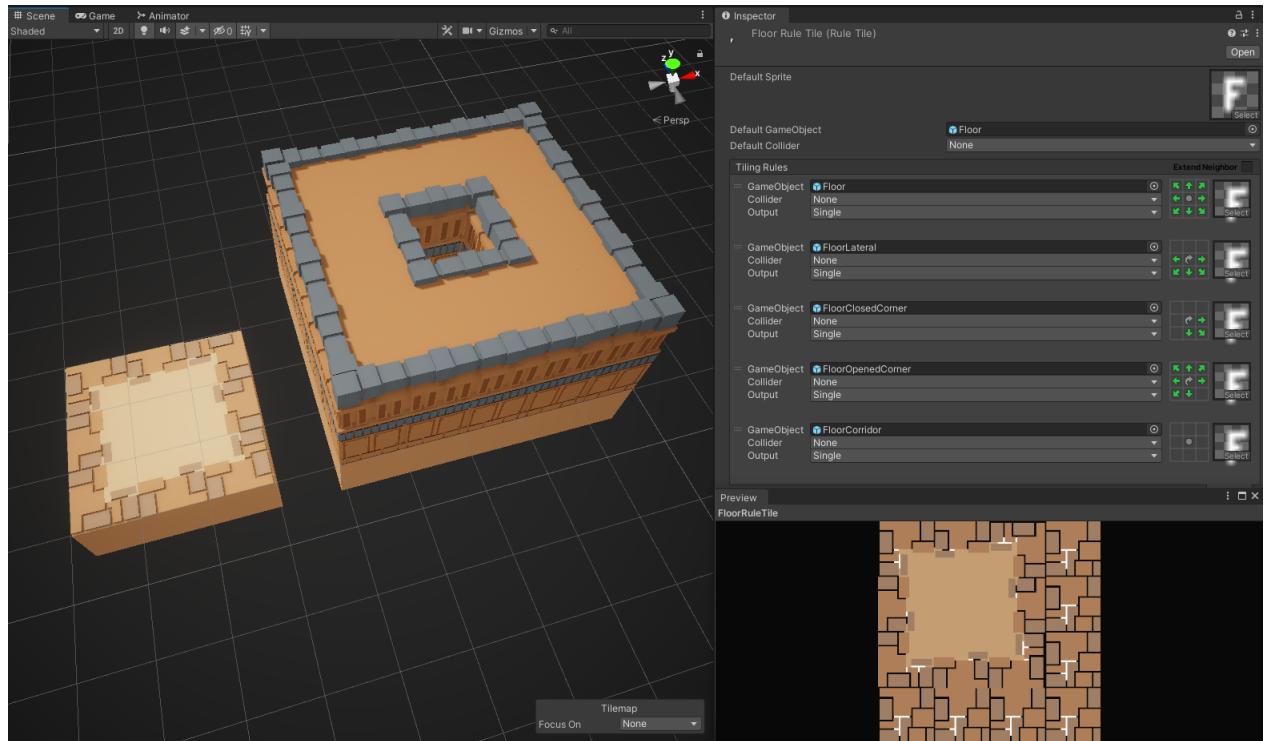


Figura B.10: Modelos 3D utilizados para las casillas del tablero, junto a la configuración del módulo de Unity que permite colocarlos y orientarlos automáticamente en base a reglas preestablecidas.

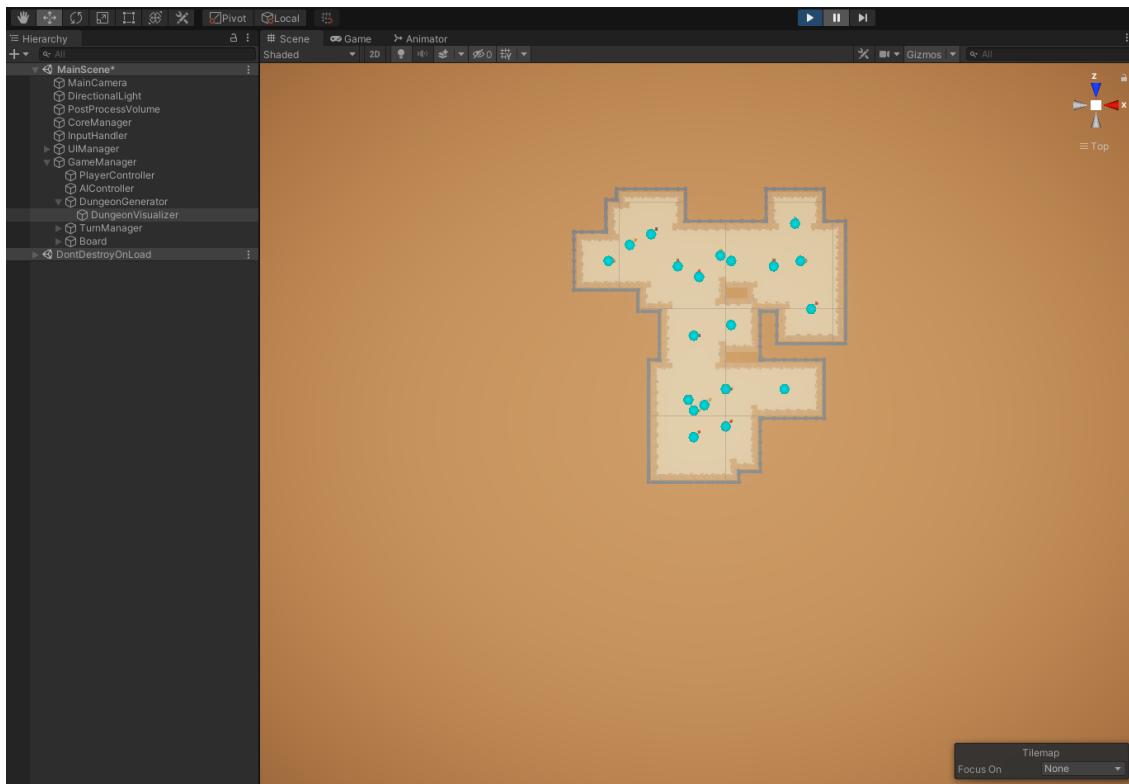


Figura B.11: Primer paso de la generación de mazmorras: se crean las salas en el centro del mapa.

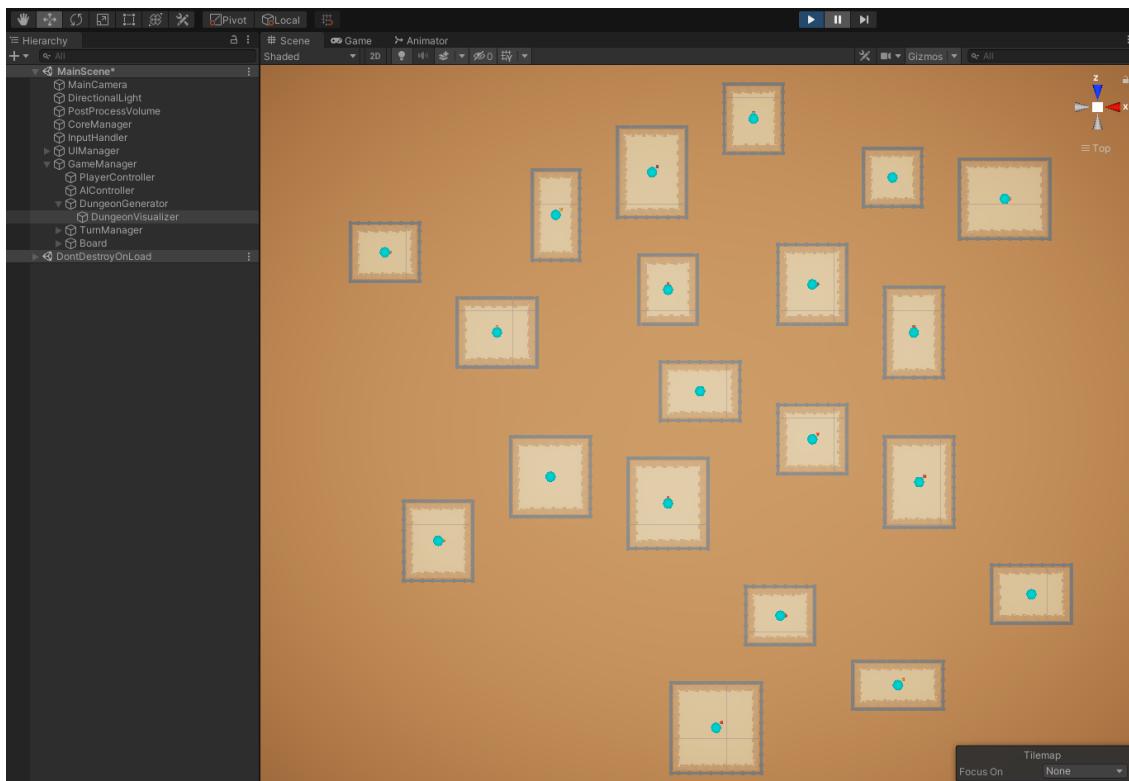


Figura B.12: Segundo paso de la generación de mazmorras: se repelen las salas para separarlas.

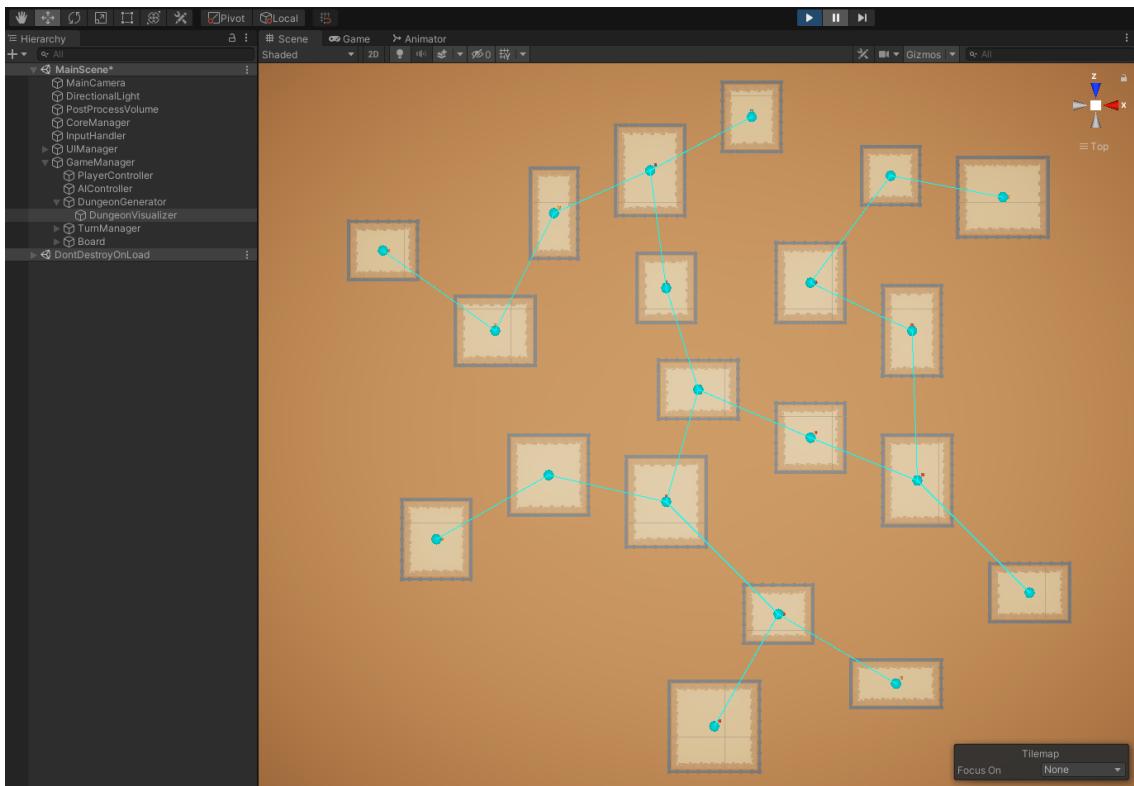


Figura B.13: Tercer paso de la generación de mazmorras: se calcula el árbol recubridor mínimo.

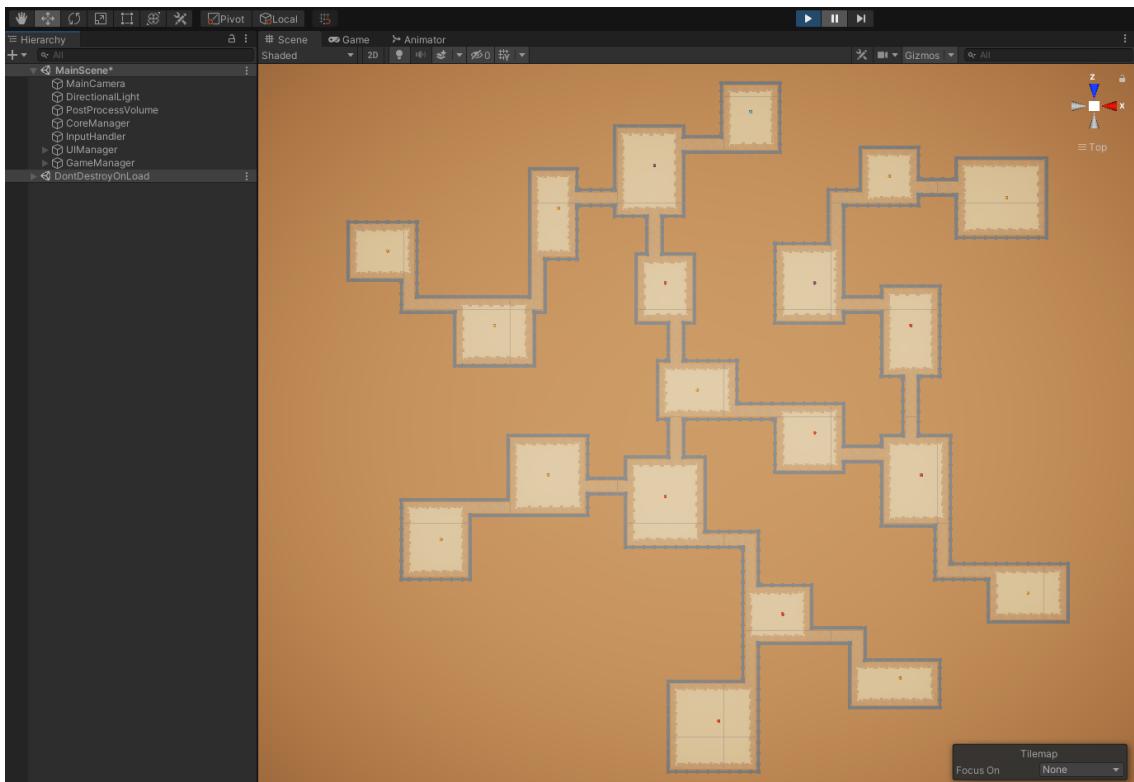


Figura B.14: Cuarto paso de la generación de mazmorras: se construyen los caminos entre salas.

