Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Herramienta para el diseño y generación de niveles procedurales en 2D para Unity

Estudiante: Alejandro Silva Durán

Dirección: Enrique Fernández Blanco

A Coruña, 13 de Noviembre de 2023.

A todos aquellos que me han construido como persona. Estoy en el buen camino.

Agradecimientos

A mi familia, por haberme apoyado siempre.
A mi pareja, por haberme soportado tanto.
A mis amigos de HS Nicaragua, sois lo mejor que me ha dado la carrera.
A Isidro y Dani, sin vosotros nada de esto existiría.

Resumen

La generación procedimental es una técnica de programación que cada vez goza de más éxito en el ámbito de los videojuegos. Continuamente se presentan productos que implementan esta técnica para diferentes aspectos de su construcción con rotundo éxito entre sus usuarios. La mayor ventaja de esta técnica es que se puede utilizar para prácticamente cualquier aspecto de un videojuego. Otro aspecto muy aclamado por la crítica del mundo de los videojuegos es la capacidad de sorprender al usuario, incluso jugando al mismo juego varias veces. Esto se consigue en mayor medida gestionando una generación de contenido de manera impredecible, o pseudoaleatoria.
Este proyecto desarrolla un sistema basado en estos dos conceptos, sumados a la recolección de datos por medio de plantillas de texto. Se implementará una herramienta capaz de generar un juego con los elementos básicos, fundamentado en una fuerte generación procedimental, que dé resultados satisfactorios y sobre todo, diferentes.

Abstract

Procedural generation is a programming technique that is increasingly successful in the field of video games. Products that implement this technique for different aspects of their construction, are often presented with overwhelming success among their users. The biggest advantage of this technique is that it can be used for most aspects of a video game. Another highly rated aspect by reviews, on the field of video games, is the ability to surprise the user, even by playing the same game several times. This is archieved mostly by managing content generation in an unpredictable way, or pseudorandom.
This project develops a system based on these two concepts, with the adition of data managment through text templates. The tool that will be implemented, is going to be capable of generating a game with the basic elements, based on a strong procedural generation, that gives satisfactory and, above all, different results.

Palabras clave:

- Videojuego
- Procedimental
- XML
- Pseudoaleatoriedad
- Unity
- Mazmorra

Keywords:

- Videogame
- Procedural
- XML
- Pseudorandom
- Unity
- Dungeon

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este capítulo expone la motivación y los objetivos de este proyecto.

1.1 Motivación

El sector del entretenimiento digital es uno de los sectores que más ha crecido en los últimos años. Como recoge la última edición del Libro Blanco del desarrollo Español de videojuegos [1, 2], el mercado de videojuegos facturó en 2021 unos 175 mil millones de dólares, siendo esta cifra superior al volumen de negocio de la industria musical y cinematográfica combinadas. A su vez, se estima que en 2022 se han facturado cerca de los 230 mil millones de dólares [3].

Dentro del desarrollo de videojuegos, se deben tomar una serie de decisiones que definen el resultado, tanto referidas a la parte artística del videojuego, como la ambientación, la historia, los personajes, etc.; como referidas a la parte técnica fuertemente influenciada por las decisiones artísticas. Algunas decisiones artísticas son: el entorno gráfico, es decir, las dimensiones, Tridimensional (3D) o Bidimensional (2D), el tipo de cámara, las mecánicas o las físicas, entre otros.

Cada decisión que se toma acarrea un cierto grado de complejidad para el desarrollador, que debe diseñar e implementar cada aspecto del videojuego por separado, para después ensamblarlo todo. Algunas de estas tareas serán: el desarrollo de los escenarios donde ocurre el juego, la composición de la música, las mecánicas del juego a implementar, etc. La Programación procedimental (PP) [4] es un paradigma de programación cuyo objetivo es agrupar y automatizar la creación de algunos de estos aspectos, para reducir la carga de trabajo al programador. Esta técnica se basa en crear algoritmos para el tratamiento y transformación de los datos de entrada, independientemente de cuales sean estos, siempre que cumplan una serie de precondiciones. Para poder aplicar esta técnica, se deben definir una serie de procedimientos que lleven a cabo tareas recurrentes dentro del proyecto. Por ejemplo, la creación

de habitaciones en un videojuego de mazmorras, es una tarea que, predeciblemente, tendrá que realizarse un número elevado de veces. Por tanto, esta tarea es adecuada para definir un procedimiento que la automatice.

El uso de esta técnica en mecanismos que realicen un gran número de actividades recurrentes, flexibiliza en gran medida el trabajo del usuario final de la herramienta.

Aplicada a los videojuegos, la PP puede añadir un componente pseudoaleatorio; este componente puede, desde indicar al programa una serie de posibles configuraciones y que el programa aplique una aleatoriamente, hasta darle una serie reducida de elementos y que el propio programa los construya de una manera impredecible (pseudoaleatoria). Esto implica la definición de una probabilidad ínfima de repetición de una determinada partida y los componentes que hay en ella, incrementando la rejugabilidad y la individualidad de la experiencia.

Cada vez más videojuegos hacen uso de la programación procedimental, algunos de ellos sumamente exitosos como No man's Sky [5] o The binding of Isaac [6]. Por todo lo expuesto en este punto, que se ha decidido que la implementación, la base y la característica principal de la herramienta que se expone en este proyecto, sea la aplicación de PP.

1.2 Objetivos

Tal y como se describió en el Anteproyecto, y atendiendo a las necesidades y ventajas descritas en el anterior punto, el objetivo principal del presente Trabajo de Fin de Grado es desarrollar una herramienta basada en la PP para la generación de niveles 2D para el framework Unity [7]. Para ello, se deben satisfacer una serie de subobjetivos:

- Recoger datos sobre el videojuego mediante plantillas redactadas en ficheros XML. Los datos más importantes que deben aportar estas plantillas a la herramienta son el diseño de habitaciones, la distribución de niveles y la información relativa a sprites.
- Gestionar los recursos necesarios para el ensamblado del videojuego, tales como pistas de audio, hojas de sprites relativas al entorno o a los personajes que haya dentro del videojuego, así como las relativas a los diseños de las diferentes habitaciones.
- Configurar un espacio donde introducir y conectar habitaciones, donde haya una habitación de inicio y otra de final, con todas las salidas conectadas, cumpliendo ciertos requisitos globales, como garantizar un mínimo de generadores de entidades o el libre movimiento del usuario por toda la superficie del suelo.

Asimismo, para ilustrar el alcance y la capacidad de la herramienta, se configurará un pequeño ejemplo utilizando recursos gratuitos.

1.3 Estructura de la memoria

La memoria está dividida en los siguientes apartados:

1. **Introducción**: El presente capítulo, que contextualiza el proyecto y plantea los objetivos del mismo.
2. **Fundamentos Tecnológicos**: Se introduce de manera técnica la programación procedimental. Para ello se expone la estructura y tecnologías que se irán utilizando a lo largo del proyecto. También se hace un estudio de mercado para comparar la herramienta propuesta con otras existentes, dentro y fuera del mercado de Unity.
3. **Metodología y Planificación**: Se exponen las etapas del proyecto en base a la metodología elegida. Se realiza la toma de requisitos y se estiman los recursos necesarios para el proyecto. También se encontrará en este apartado el resultado del seguimiento, en un afán por agrupar esta información.
4. **Desarrollo**: Análisis, diseño, implementación y pruebas de cada una de las iteraciones o etapas, marcadas por la planificación.
5. **Conclusiones**: Planteamiento de las lecciones aprendidas y el análisis final del proyecto, teniendo en cuenta duración, incidencias, aprendizaje y resultado final.
6. **Trabajo Futuro**: Explora posibles mejoras, funcionalidades que se podrían añadir o adaptaciones de la herramienta.

Capítulo 2

Fundamentos Tecnológicos

Este capítulo presenta los fundamentos tecnológicos que se han considerado para el desarrollo del proyecto. Así pues, se realizará una introducción al concepto de Programación procedimental (PP). Para ello, se analizarán ejemplos de uso de la PP en diferentes ámbitos como son el entretenimiento digital, cine y, finalmente, en el ámbito concreto del desarrollo de videojuegos. Se realizará un estudio de las herramientas que existen dentro del mercado de recursos de Unity [8] y se justificarán dos decisiones técnicas que se han tomado durante el desarrollo del proyecto.

2.1 Programación Procedimental

La PP parte del paradigma imperativo [9] y sus bases se fundamentan en llamadas a procedimientos. Entiéndase por procedimiento el conjunto de instrucciones que definen el algoritmo empleado para resolver una tarea específica, siendo estos un tipo de las conocidas como subrutinas. Aplicado a lenguajes de alto nivel este paradigma se le conoce como Programación Funcional.

El objetivo es que estos procedimientos realicen las diferentes operaciones específicas y comunicaciones necesarias para llevar a cabo las tareas a alto nivel que solicita el usuario u otros procedimientos. Así, se permite anidarlos para, en última instancia, presentar al usuario lo que solicita.

En el ámbito de los videojuegos, el programador define una serie de procedimientos que genera automáticamente el contenido del juego [10]. Esta técnica se puede usar para generar ciertos aspectos del juego como Minecraft [11], que usa la generación procedimental para crear el terreno y las entidades que hay en él. Esto llevado al extremo, permite también gene-

rar el juego entero de manera procedimental, como en The Binding of Isaac.

Dentro de la PP hay tres aspectos ya mencionados que son los más importantes, siendo estos:

- **Modularización**: La idea detrás de este principio es identificar las tareas específicas del programa donde se aplica el paradigma y construir un procedimiento que la realice. Por ejemplo, en un programa el cual construya un coche, las tareas específicas serían, por ejemplo, construir las ventanas, los neumáticos, el volante, etc. Estas son las tareas específicas que hay que automatizar, con lo cual, en vez de tener un código que realice todas las tareas él mismo, obtendría una función principal cuyo único objetivo es llamar a los procedimientos y ensamblar sus resultados para obtener el producto final. Esto permite, por otra parte, decidir que tareas se pretende realizar sobre el resultado final y cuales no de forma más ordenada.

- **Generalización**: Esto implica buscar tareas que compartan las suficientes similitudes como para poder agruparlas en el mismo procedimiento. La generalización permite, con el mismo procedimiento, obtener varios tipos de objetos. Siguiendo con el anterior ejemplo, imagínese el procedimiento que se encarga de construir el cristal de la ventana. Si se permite que recoja datos sobre forma, grosor y tamaño, se puede generar todas las lunas del coche con el mismo procedimiento. Este principio ayuda a tener un código más limpio y ordenado.

- **Anidamiento** : La última característica principal de este paradigma consiste en el anidamiento de procedimientos. Esto debe imaginarse en una estructura de árbol. La raíz es el procedimiento principal, el cual no es llamado por ningún otro procedimiento, y las hojas son aquellos procedimientos que no llaman a ningún otro. Los niveles intermedios son tareas no modulares que pueden expresarse como llamadas a varios procedimientos.

  Los pisos más bajos del árbol tendrán un alto grado de especificidad mientras que realizarán tareas menos complejas que los pisos superiores del árbol. Por contra, cuanto más se sube en la estructura del árbol, menos específicos serán los datos de entrada y las tareas a realizar por cada nodo.

  Un ejemplo para ilustrar esto, siguiendo con el programa que genera el coche, sería el expuesto a continuación: la raíz del árbol sería un procedimiento cuyo tipo de retorno será el coche y cuyos datos de entrada serán las características generales del coche: número de puertas, asientos, etc. Por tanto, el árbol se encargará de pedir y ensamblar

cada parte del coche. Los niveles intermedios serán procedimientos que realicen tareas no modulares como por ejemplo, montar una puerta, la cual requiere de un seguro para bloquearla, una cristal a modo de ventana, un soporte, etc. Cada nodo se encarga de pedir a sus nodos hijo los elementos que necesita y ensamblarlos. Por último, las hojas del árbol son procedimientos que realizan las tareas específicas y por tanto construyen los elementos más básicos de la estructura, como el cristal o el seguro ya mencionados.

2.1.1 Programación Procedimental y Aleatoriedad

En relación al último punto del apartado anterior, se puede introducir el concepto de pseudoaleatoriedad. Cuanta más abstracción se consiga de las tareas hoja, más factores se pueden generar de manera pseudoaleatoria. Aplicar correctamente el anidamiento e introducir algún tipo de generación pseudoaleatoria de datos permite conseguir una herramienta que, con introducir un pequeño grupo de entradas, construya el objeto deseado de una manera pseudoaleatoria. Esto garantiza que se obtiene el tipo de objeto deseado, por ejemplo, una mazmorra, pero siendo imposible de predecir la manera en la que se ha construido y cómo se presentará el producto. Para este caso, podría generarse pseudoaleatoriamente la distribución de habitaciones para la mazmorra.

Una ventaja derivada de introducir este factor es la alta capacidad de reutilización que aporta la PP pseudoaleatoria, ya que se pueden generar con la misma herramienta una elevada cantidad de productos iguales en su forma, pero diferentes en su composición.

Retomando el anterior ejemplo del programa que construye un coche; una manera de introducir un componente aleatorio al programa sería, por ejemplo, permitir que los datos que no se han introducido en la llamada al procedimiento principal, se generen de forma pseudoaleatoria, introduciendo un mínimo de datos estructurales. En este caso, se podría indicar que el coche deseado debe tener 4 ruedas, 7 asientos, 8 faros delante y 8 detrás. La información que debe generarse pseudoaleatoriamente para este ejemplo es el color del coche, su forma o el tipo de motor, entre otros.

2.1.2 Programación Procedimental en el Entretenimiento Digital

La PP se aplica en diferentes disciplinas del entretenimiento digital, sin duda la más prolífica son los videojuegos, pero también en los últimos tiempos tiene relevancia en el cine, más concretamente en el ámbito de la animación 3D. Algunos ejemplos son:

- **MASSIVE** [12]: Se trata de un software de animación que hace uso de Inteligencia Arti-

ficial (IA) desarrollado específicamente para generar miles de soldados automáticamente en las escenas de batallas en la saga del Señor de los Anillos de Peter Jackson. Utiliza fuzzy logic para dar independencia a cada individuo, aportándole comportamientos y capacidad de responder a su entorno. Aparte de este aspecto, la herramienta también sirve para generar conjuntos de partículas. Su popularidad ha aumentado tanto que se ha utilizado en películas tan importantes como Avatar (2009), Rise of the Planet of the Apes (2011), Avengers: Endgame (2019), Happy Feet (2006), WALL-E (2008) o Yo, Robot (2004). Películas donde además el aspecto visual resulta una de sus características más reseñables.

- **HOUDINI** [13]: Su nombre técnico es SideFX Houdini FX y se trata de un software de animación 3D, adaptado de la herramienta PRISMS, para la generación procedimental de imágenes. Entre las funcionalidades más destacadas que posee Houdini, se pueden nombrar:

  - Creación de escenarios y entornos
  - Rigging
  - Render

  Houdini se ha utilizado en producciones como Frozen (2013), Zootopia (2016), Raya y el último dragón (2021) o Contact (1997).

2.1.3 Programación Procedimental en los videojuegos

El ámbito donde brilla esta técnica, la Programación procedimental, en el entretenimiento digital es en los videojuegos. Como se ha avanzado en el capítulo de introducción, la PP aplicada al ámbito de los videojuegos define procedimientos que realicen las tareas más recurrentes del videojuego. Normalmente, este tipo de programación contribuye al encapsulamiento del código por lo que, cuanto más general sea el procedimiento, más tareas con similitudes se podrán agrupar dentro de él.

Un ejemplo de aplicación de esto, podría ser, la generación procedimental de flora y fauna. Imagínese un procedimiento, cuya tarea es crear flores, donde su entrada es, únicamente el tipo de flor a crear. Si a este procedimiento se le permite, en vez de recibir un tipo de flor, recibir información sobre tamaño y grosor del tallo, así como la forma de la corona o la textura que utiliza. Ahora, con la información adecuada, este procedimiento puede crear árboles, columnas, caminos, etc. En definitiva, cuanta más información se permita indicar sobre la tarea a realizar, más tareas que guardan similitudes con la tarea original del procedimiento podrán

agruparse.

Una de las características más deseables al aplicar este tipo de paradigma al ámbito de los videojuegos es el componente pseudoaleatorio, que está relacionado con el principio de anidamiento, ya mencionado al inicio de este capítulo. Cuanta menos información se le aporte al programa en capas superiores, más información tendrá que ir generando en capas inferiores, lo cual hace que el resultado sea casi impredecible, y al usuario le parezca aleatorio. Entiéndase por capas superiores y capas inferiores, los niveles de la estructura en forma de árbol referida en la sección de Anidamiento (página 5).

En general, este paradigma se utiliza para aumentar el rendimiento y el grado de rejugabilidad, y simplificar el tratamiento de datos. Además, los códigos que implementan estas técnicas son en su mayoría reutilizables para otros proyectos.

Algunas de las tareas concretas más habituales para las que se utiliza este paradigma en el ámbito de los videojuegos son:

- Generación aleatoria de niveles.
- Generación de terreno.
- Generación de diálogos.
- Generación de entidades del juego, como enemigos.

El género Rogue-Like

Existe una larga lista de obras que usan estas técnicas para generar contenido del videojuego, sin embargo, el genero de videojuegos que más se nutre de esta técnica es el *Rogue-Like*.

Los videojuegos *Rogue-Like* [14] son un género de videojuegos cuyo mapa se genera de manera aleatoria y, cada mapa, constituye una Dungeon o mazmorra. Estas dungeons suelen ser muy cortas y con un ritmo acelerado, ya que el punto fuerte de este tipo de juegos es la rejugabilidad.

Una característica de este género es que, normalmente, no impone al jugador una manera de jugar. Esto está asociado a la PP. En la mayoría de los juegos de este género, la disposición de objetos y la identidad de los mismos es aleatoria también, por lo que se abren muchas posibili-

dades para encaminar la partida y el jugador puede elegir cualquiera de ellas sin restricciones.

Otra característica distintiva de este género es que cada vez que el personaje muere debe empezar de nuevo desde el principio, a esto se le llama muerte permanente, y también está asociada directamente a la PP. El uso de la PP en un videojuego, normalmente implica regenerar el nivel entero una vez muere el personaje. Cuando esto sucede, el personaje debe comenzar de nuevo el nivel sin ningún objeto o mejora que haya adquirido en el transcurso del mismo. Esta característica consigue hacer sentir al jugador el valor real, tanto del progreso que lleva en la mazmorra, como del hecho de tomar una decisión arriesgada que pueda implicar morir.

A nivel narrativo, estos juegos suelen aportar justificaciones para comenzar la dungeon una y otra vez. A su vez, a medida que se avanza en el juego, se desbloquean habilidades o nuevas rutas que permiten avanzar y seguir construyendo la historia. Cada recorrido de la mazmorra se denomina Run, y estas runs son lo que aporta el dinamismo a la experiencia.

Ejemplos de Programación Procedimental en los videojuegos

Algunos de los ejemplos más relevantes para la historia de los videojuegos que hagan uso de la técnica de programación y generación procedimental son:

- **Rogue (1980)** [15] : Este juego se considera el padre de los *Rogue-Like*, el Layout del mapa, las entidades y objetos se generaban de forma procedimental. La pretensión del grupo que lo desarrolló era conseguir un juego que maximizase la relación contenido-recursos, por lo tanto, los gráficos eran caracteres ASCII.

  Para maximizar el aspecto de contenido, utilizaron la técnica de generación procedimental, cada nivel comenzaba con la referencia del tablero de 3 en raya con cada habitación de tamaño diferente ocupando un espacio en el tablero, para después conectarlas mediante pasillos que se añadían al diseño de la Dungeon.

- **The Binding of Isaac (2011)**: Se trata de probablemente uno de los mayores exponentes del genero *Rogue-Like*, que también genera todo su contenido de manera procedimental.

  Está inspirado en Zelda y también utiliza el concepto de Dungeon, pero avanzando ha-

cia abajo, en vez de ser un layout completo generado de forma procedimental.

La partida se compone de 5 pisos, cada uno dividido en dos, cuyos layouts son los que se generan de manera procedimental. Existen maneras de saltar pisos, repetir pisos o ignorar enemigos o habitaciones.

Es un juego diseñado minuciosamente para promover el uso de técnicas de MiniMaxing a muy alto nivel, todo debido al componente aleatorio que permite la redistribución de una serie de reglas conocidas por el jugador.

- **Minecraft (2011)**: Calificado como el mejor juego del siglo 21 por *The Guardian* en 2019, galardonado en 2016 con el sexto puesto en el ranking de *New York Times "The 50 Best Video Games of All time List"* [16], en este juego el mundo se genera y existe de manera simultanea y aleatoria.

  Tiene unas reglas que el usuario debe ir descubriendo y adaptándose a ellas, y además, está construido de tal manera que, como jugador, puedas hacer prácticamente cualquier cosa.

  El terreno, entidades, estructuras y objetos se generan de manera procedimental, se utiliza una semilla que define el mundo, de tal manera que cada vez que un mundo se cree con esa semilla, será el mismo, pero cada diferente semilla genera un mundo completamente diferente.

- **No Man's Sky (2016)**: El primer videojuego capaz de anidar generaciones procedimentales, no solo en el mundo, sino incluso en el terreno, fauna y flora, de forma interna:

  Cada modelo de, por ejemplo, plantas, puede generarse con cierta probabilidad en cada planeta y cada planta tiene cierta probabilidad de generarse de alguna manera específica, también de forma procedimental.

  Esto aplica también a planetas y galaxias, ya que es un juego de exploración espacial, cada galaxia se genera de manera procedimental con ciertas probabilidades ligadas y cada sistema dentro de esa galaxia también lo hace de esta manera. El videojuego está construido de forma que continúa anidando hasta estrellas, planetas e incluso zonas planetarias, todo generado de manera procedimental.

2.2 Consideraciones Técnicas Previas

En este apartado se analizan dos cuestiones importantes que afectan a la realización del proyecto:

2.2.1 ¿Por qué 2D?

La elección de desarrollar la herramienta exclusivamente para 2D se debe a ciertos factores, entre otros:

- Es un tipo de programación gráfica que no requiere de unos recursos muy potentes para poder renderizar su contenido, por tanto, muchos más usuarios podrán acceder al mismo.
- Se trata del tipo de animación más utilizado en videojuegos para móvil.
- La curva de aprendizaje de este tipo de animación es mucho más liviana que de tratarse de 3D al prescindir de la dimensión de profundidad.
- Al tener una programación más simple, el desarrollador puede poner más énfasis a otros aspectos tales como el artístico o el narrativo, logrando, por lo general, juegos que, aunque más simples, están más cuidados y son más eficientes.
- Otra razón es el aumento de la popularidad en los últimos años del contenido en 2D en relación a la cantidad de herramientas específicas que existen.

2.2.2 ¿Por qué Unity?

Unity [17] es un motor gráfico multiplataforma 2D y 3D el cual soporta el desarrollo sobre varios dispositivos. Se trata de una herramienta intuitiva y que permite trabajar en más de 18 plataformas diferentes, según datos de la propia compañía, pudiendo incluso desarrollar proyectos con realidad virtual.

Por encima de todo esto, es una herramienta gratuita que permite no solo el desarrollo, sino la publicación gratuita de las obras, siempre que facturen menos de 100 mil dólares.

En oposición a Unity, está el motor Unreal Engine, ampliamente extendido. Sin embargo, en la tabla comparativa 2.1 (página 16) se pueden apreciar una serie de desventajas en relación al motor Unity.

Los datos más importantes en comparación a Unreal Engine son la gratuidad de su licencia y su diferencia de curva de aprendizaje, lo cual es suficiente para justificar su uso, sumado a lo anteriormente mencionado.

Cabe considerar que, en el campo de la creación de videojuegos, según datos de la propia compañía, el 70% de los 1000 principales juegos móviles se crearon con unity y tiene una participación activa mensual de 1.8 millones de creadores.

Algunos ejemplos de videojuegos famosos en 2D desarrollados con Unity son, Hollow Knight (2017), Cuphead (2017), la bilogía Ori (2015 y 2020), Celeste (2018) o Valheim (2021).

2.3 Estudio de Mercado

A continuación se expone un estudio sobre las posibles ventajas, competencia, diferencias con los productos existentes, para aportar contexto de negocio al proyecto.

2.3.1 Herramientas Similares en Unity Asset Store

La empresa Unity Technologies, responsable del Motor Unity, pone a disposición de sus usuarios un portal mediante el cual pueden comerciar con sus propias herramientas, desarrolladas por ellos mismos. Se trata de la Unity Asset Store. Al realizar la búsqueda "*Procedural*" en la Unity Asset Store, se obtienen 831 resultados. La gran mayoría de estas herramientas solo generan terrenos, entidades, objetos como árboles, grids, formas, User Interface o estructuras. Al modificar la búsqueda a "*Procedural level*", ahora se obtienen solo 139 resultados, de los cuales ahora se expondrán las diferencias de algunos de los más completos con el proyecto propuesto:

- **Roguelike Generator Pro - Level & Dungeon Procedural Generator** by nappin [18], figura 2.1 (página 13): La principal diferencia de esta herramienta es que no es gratuita. Se trata de un generador de layouts, no genera entidades, por contra, permite diseños en 3D, 2.5D y 2D. Esta herramienta no facilita una manera de manejar el flujo del juego, siendo esto la transición de niveles y el uso de interfaz de usuario.
- **Edgar Pro - Procedural Level Generator** by OndrejNepozitek [19], figura 2.2 (página 13): Podría ser la versión más parecida al proyecto planteado, permite definir la mayoría de variables, desde el diseño de habitaciones mediante editor, hasta los posibles layouts. También incluye niebla de guerra y puertas. Esta herramienta, sin embar-

Figura 2.1: Roguelike Generator Pro

go, tampoco facilita una manera de manejar el flujo del juego. Pero la diferencia más significativa es que es una herramienta de pago.

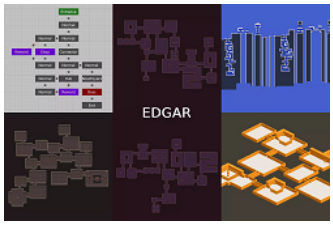


Figura 2.2: Edgar Pro - Procedural Level Generator

- **Procedural Level Generator** by Juan Rodriguez [20], figura 2.3 (página 13): Esta herramienta es gratis, pero no permite colocar jugador, solo genera el nivel con enemigos, la herramienta propuesta permitirá gestionar también este aspecto aparte del flujo del juego.

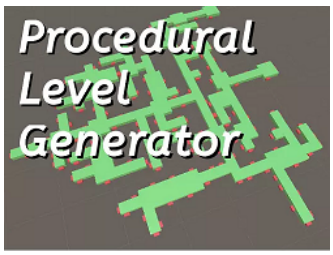


Figura 2.3: Procedural Level Generator

- **Advanced Dungeon Generator - LITE** by Oscar Vezz [21], figura 2.4 (página 14): Esta herramienta, aparte de ser gratis, es muy completa, pero carece de plantillas para adscribir mecánicas, armas y herramientas al jugador, cosa que la herramienta propuesta sí incluye.
- **New Dungeon Pack Modular Low Poly (Free Version)** by Daniel Miranda [22], figura 2.5 (página 14): Esta herramienta es gratis, pero utiliza diseños internos, con lo

Figura 2.4: Advanced Dungeon Generator - LITE

que no permite actualizar texturas, añadir usuario o entidades, solo generar el nivel. Además la herramienta es para programación 3D.

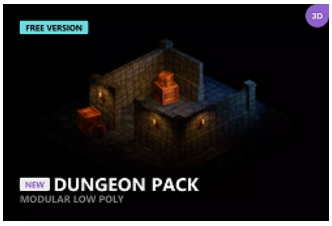


Figura 2.5: New Dungeon Pack Modular Low Poly

Después de presentar estas herramientas que utilizan PP para realizar diferentes tareas, se puede llegar a la conclusión de que la herramienta que se desarrollará a lo largo de este proyecto debe basarse en los siguientes aspectos:

- **Generalización**: La herramienta debe permitir el aporte por parte del usuario de texturas, recursos visuales o de audio, para poder generar la mayor cantidad de componentes del videojuego posibles, al contrario de, por ejemplo, la herramienta *New Dungeon Pack Modular Low Poly (Free Version)*.
- **Completitud**: Al contrario que sucede en la mayoría de herramientas expuestas, se pretende que la herramienta aporte al usuario maneras de construir, desde los niveles por separado, a ensamblar el juego entero. Esto quiere decir que la herramienta debe aportar también una manera de manejar el flujo del juego. Es uno de los aspectos diferenciantes y más significativos de la herramienta.
- **Animación 2D**: El entorno gráfico que más puede aprovechar la técnica de generación procedimental es sin duda el entorno 2D, como se puede observar en la herramienta *Edgar Pro - Procedural Level Generator*, que resulta ser la más completa. Lo cual reafirma la anterior decisión de escoger el entorno 2D como entorno de la herramienta.

- **Precio**: Ligado a los anteriores puntos, como se puede apreciar también en la herramienta *Edgar Pro - Procedural Level Generator*, cuanto más completa sea la herramienta, más se eleva su precio. Por ello un pilar fundamental de este proyecto será que la herramienta será gratuita.

2.3.2 Análisis DAFO

El análisis Debilidades-Amenazas-Fortalezas-Oportunidades refuerza la viabilidad del proyecto como se puede apreciar en la tabla 2.2 (página 16).

Como se puede ver en la tabla 2.2, las debilidades y amenazas señalizan una herramienta que, a priori, puede resultar tediosa de usar y, en general, puede tener una curva de aprendizaje empinada. Sin embargo, también expone una herramienta útil, flexible y con unas bases lo suficientemente extendidas, como el motor o el entorno gráfico. Estas características son suficientes como para generar interés y confianza en el producto.

Unity	Unreal Engine
Licencia gratuita para pequeños proyectos	Licencia de pago
Uso de C#	Uso de C++
Bajos requisitos en cuanto a recursos	Orientado a altos recursos y calidad gráfica
Formato e interfaz amigables y sencillos	Curva de aprendizaje pronunciada

Cuadro 2.1: Comparativa entre Unity y Unreal Engine

Debilidades	**Amenazas**
- Proceso de depuración de código - Alta complejidad de configuración para el usuario	- Aparición de otras herramientas durante el desarrollo - Proyecto sujeto a disponibilidad de recursos
Fortalezas	**Oportunidades**
- Utilidad para la comunidad - Mercado en crecimiento constante - Creciente demanda de contenido 2D - Experiencias variadas y únicas a bajo coste	- Adaptabilidad frente a nuevas tecnologías - Motor base ampliamente extendido - Gran mercado potencial de estudios y desarrolladores

Cuadro 2.2: Tabla Debilidades-Amenazas-Fortalezas-Oportunidades del proyecto

Capítulo 3

Metodología y Planificación

3.1 Metodología de desarrollo

La metodología que se utilizará en el proyecto será **Scrumban** [23]. Se trata de una planificación ágil que adopta características de Scrum y Kanban, dos metodologías diferentes.

Las características de **Scrum** [23] presentes en **Scrumban** son, entre otras, que la prioridad de trabajo se establece mediante un balance entre complejidad de la tarea y demanda del producto, así como marcar específicamente y con consenso el final de cada tarea. También es significativo que las iteraciones se producen en intervalos constantes, normalmente al final de cada iteración tras reuniones para la revisión de tal iteración. Las características de **Kanban** [23] presentes en **Scrumban** son la limitación de tareas simultáneas, para controlar las cargas de trabajo, o la especificación de una serie de tareas que componen el núcleo del trabajo que deben ser realizadas para que el trabajo se considere terminado.

3.1.1 Adaptación de la metodología al proyecto

Se indica que la metodología **Scrumban** aplica a grupos de varios integrantes que se sincronizan a través de reuniones como el daily Scrum [24], sin embargo en este caso se ajustará el método, siendo necesario que el único integrante adapte múltiples roles, incluidos el de asegurador de calidad o el "Scrum Master" [25].

La planificación de las iteraciones será más flexible, es decir, el individuo analiza las prioridades y necesidades del proyecto y fórmula las diferentes iteraciones para sí mismo, y, en favor, el flujo de trabajo se simplifica ya que no es necesaria una sincronización constante, se puede continuar con seguimiento simple de flujo KanBan, utilizando un tablero, con columnas

como “To do”, “Doing” y “Done”.

El proyecto se planteará como una sucesión de iteraciones, donde cada iteración estará dividida en tareas modulares que son las que se introducirán en el tablero. El seguimiento se realizará cada vez que todas las tareas relativas a una iteración estén en la columna “Done”.

Una vez expuestas la metodología y sus adaptaciones, es importante remarcar que cada iteración estará dividida en los siguientes ciclos:

- **Análisis**: Se busca entender y recoger las funcionalidades que se desean aportar con la iteración concreta. Se deben definir y detallar los requisitos que debe cumplir la misma.
- **Diseño**: Se preparan los diagramas para poder recoger los distintos componentes y contexto de la posterior implementación.
- **Implementación**: Se codifican las funcionalidades especificadas en el análisis y diseño.
- **Pruebas**: Se comprueba que las funcionalidades implementadas funcionen correctamente.

3.2 Recursos

Este proyecto se ha llevado a cabo con los recursos humanos indicados en la figura 3.1 (página 18)

Tipo de Recurso	Cantidad
Analista/Diseñador/Desarrollador	**1**(Alumno)
Jefe de proyecto	**1**(Tutor)

Cuadro 3.1: Tabla de Recursos Humanos

Las responsabilidades que se esperan del alumno son diseñar e implementar la herramienta, y documentar todo el proceso. Las responsabilidades del tutor son la supervisión del proyecto, las correcciones pertinentes en las diferentes iteraciones que así lo requieran.

3.2.1 Recursos Técnicos

- **Recursos Hardware**: Se limitan a la máquina en la que se ha desarrollado el proyecto, en este caso, un ordenador de sobremesa con las siguientes características: procesador

AMD Ryzen 7 2700X Eight-Core Processor 3.70 GHz, GPU NVIDIA GeForce GTX 1660 SUPER, RAM de 8,00 GB y un sistema operativo de 64 bits.

- **Recursos Software**: Para desarrollar este proyecto se han utilizado las siguientes tecnologías:
  - **Unity** [26]: El motor gráfico Unity con los siguientes módulos principales; **CoreModule**, el módulo central de Unity que contiene la clase **MonoBehaviour**; UnityEditor, API que permite trabajar con el Unity Inspector y el propio Unity Inspector; y como módulo externo, el paquete **Cinemachine**, que facilita el control de la cámara.
  - **Visual Studio** [27]: IDE que utilizado conjuntamente con C#, ayuda a realizar las tareas de codificación. También se utilizó para construir los diagramas de clases.

  Por último, hacer alusión al uso del sistema operativo *Windows*.

3.3 Toma de requisitos y casos de uso

En esta sección se definen tanto requisitos funcionales, no funcionales y Casos de uso del proyecto.

Requisitos funcionales

En base a los objetivos presentados en el apartado 1.2, se ha elaborado una tabla de requisitos funcionales, presentada en el **Apéndice B**, tabla B.1. Gracias a esto, se consigue aportar contexto a las próximas implementaciones y desarrollos de los apartados siguientes. Cada requisito funcional se presenta como una entrada con una breve descripción, su nombre e ID.

Requisitos no funcionales

Aparte de los requisitos funcionales, elaborados en la tabla referenciada en el subapartado anterior; se deben elaborar los requisitos no funcionales. Estos enmarcarán los requerimientos en cuanto a estándares de calidad del software se refiere. Los requisitos no funcionales se han desarrollado en el **Apéndice B**, tabla B.2.

Casos de uso

Una vez elaborados los requisitos, a fin de especificar las funcionalidades concretas que se desarrollarán en las iteraciones venideras, se definirán unos casos de uso iniciales, basados en los requisitos anteriormente mencionados y los objetivos del apartado 1.2. Se presentan los

casos de uso, con su correspondiente descripción y actores que participan de la interacción que describe el caso de uso. La tabla que contiene los casos de uso se encuentra en el **Apéndice B**, tabla B.3.

3.4 Iteraciones

A continuación, se expondrán las iteraciones en que se estructura la planificación de este trabajo:

- **Iteración 0**: En esta iteración se analizarán sistemas parecidos al proyecto planteado para extraer las características más relevantes, analizar sus diferencias y formar una idea de la estructura que va a seguir la herramienta.
- **Iteración 1**: Fase preparatoria donde se analizan los casos de uso, requisitos funcionales y no funcionales. Esto se hace para añadir más contexto y corregir o expandir algunos de los casos de uso que tengan varias tareas internas.
- **Iteración 2**: Comienzan a definirse los elementos complementarios a las dos operaciones principales. En esta iteración se desarrollará el personaje principal, definiendo su movimiento y características de manera general, para luego poder ajustarlas según sea necesario en posteriores iteraciones a su entorno.
- **Iteración 3**: Se definirá e implementará la configuración y operaciones relativas a la cámara.
- **Iteración 4**: Esta iteración define e implementa las operaciones y estructuras sobre las que se construirán los niveles. Entre las estructuras que se van a utilizar la más relevante es la que albergará la mazmorra. Esta estructura será de tipo matricial en la que cada posición contendrá una habitación.
- **Iteración 5**: La primera tarea central del proyecto es definir la creación de las habitaciones en base a los datos recogidos, principalmente en las dos anteriores iteraciones, y su agrupación por configuraciones similares. Para esta iteración se definirán las plantillas XML relativas a las habitaciones y tiles. También se definen los requisitos, los cuales deben cumplir las habitaciones, para ser utilizables por el algoritmo que se construirá en la siguiente iteración. En esta iteración, así como en la siguiente, es donde deben ajustarse los parámetros de los elementos creados en las iteraciones 2, 3 y 4.
- **Iteración 6**: La segunda de las tareas centrales del proyecto es definir el algoritmo que se encarga de crear una mazmorra y las operaciones básicas relacionadas con su trata-

miento y su información. En esta iteración se define la construcción de una mazmorra en base a una lista de posibles habitaciones.

- **Iteración 7**: En esta iteración se introducen los algoritmos que crean y controlan la apariencia y el comportamiento de las entidades que estarán en la mazmorra. También se crearán los objetos que instanciarán las entidades dentro del juego.
- **Iteración 8**: Se creará el algoritmo que indicará el flujo del juego. Se creará una Máquina de estados que controlará el flujo. En esta iteración se definirán los hitos generales del juego como el inicio, los puntos que controlarán los saltos de nivel y el final del juego.
- **Iteración 9**: En esta iteración se crean las clases que gestionan los elementos User Interface (UI). Se definirán las clases externas para funcionalidades como el control por teclado o la asignación de elementos UI.
- **Iteración 10**: Se pondrán a prueba las anteriores iteraciones creando diferentes ensamblados y asegurando el correcto funcionamiento y flujo del juego. Es en esta iteración donde se crea el ejemplo de referencia para otras implementaciones, ya mencionado en los objetivos del proyecto.

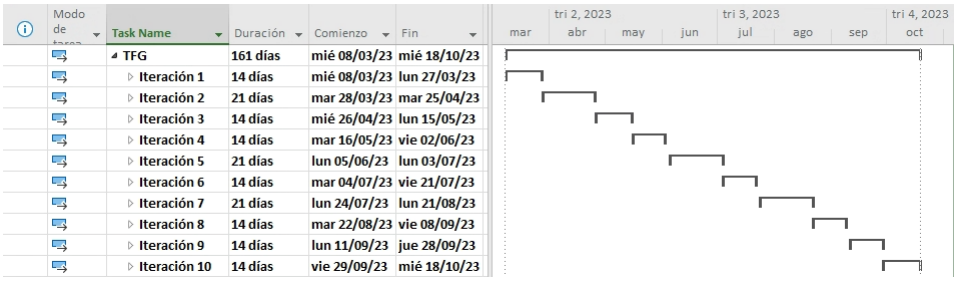
Cabe mencionar que a través de estas iteraciones se irá documentando todo el proceso para conformar la presente memoria.

3.5 Planificación inicial

La planificación inicial para este proyecto consta de las anteriormente mencionadas 10 iteraciones, las cuales llevarán a cabo una o varias tareas relacionadas con la herramienta. La metodología elegida, **Scrumban**, plantea iteraciones de duración de entre dos semanas y tres semanas, sumado a una jornada de revisión con el "Scrum Master" o jefe del proyecto.

Para la planificación inicial se han aproximado las iteraciones que se considera que serán las que requerirán 3 semanas, y cuales necesitarán 2, esto en base a los casos de uso y requisitos seleccionados en el apartado 3.3. Esto se puede ver reflejado en la primera aproximación del *diagrama de Gantt* del proyecto. En esta aproximación solo se han mostrado las iteraciones y sus duraciones como se puede ver en la figura 3.1 (página 22).

A continuación se muestra, basadas en las temporalidades calculadas en figura 3.1 (página 22), un despliegue de los casos de uso asignados a cada iteración y sus temporalidades. Esto corresponde a las figuras A.1 (página 83) y A.2 (página 84), reflejadas en el apéndice A.



Modo de tarea	Task Name	Duración	Comienzo	Fin
	TFG	161 días	mié 08/03/23	mié 18/10/23
	Iteración 1	14 días	mié 08/03/23	lun 27/03/23
	Iteración 2	21 días	mar 28/03/23	mar 25/04/23
	Iteración 3	14 días	mié 26/04/23	lun 15/05/23
	Iteración 4	14 días	mar 16/05/23	vie 02/06/23
	Iteración 5	21 días	lun 05/06/23	lun 03/07/23
	Iteración 6	14 días	mar 04/07/23	vie 21/07/23
	Iteración 7	21 días	lun 24/07/23	lun 21/08/23
	Iteración 8	14 días	mar 22/08/23	vie 08/09/23
	Iteración 9	14 días	lun 11/09/23	jue 28/09/23
	Iteración 10	14 días	vie 29/09/23	mié 18/10/23

Figura 3.1: Diagrama de Gantt Inicial

En estas figuras se aprecia como, debido a la metodología empleada, se pueden solapar fases internas de las iteraciones, como los análisis.

Como se puede apreciar en el diagrama A.1 (página 83), hay 2 iteraciones, aparte de la primera ya superada, que solo constan de sus tareas básicas. Esto se debe a que para poder implementar funcionalidades, deben crearse ciertos módulos y tareas que ameritan una iteración por sí mismos, debido a la relevancia que tienen para la herramienta, aunque el usuario no intervenga.

3.5.1 Estimación de Costes

En este apartado se estimarán los costes, tanto materiales como humanos, y se expondrá una visión general de cuanto podría costar este proyecto.

Para estimar los costes humanos se han tenido en cuenta los salarios medios de los puestos asumidos por los actores del proyecto, que son el tutor [28] y el alumno. Estos datos se pueden ver en la tabla 3.2 (página 22):

Tipo de Recurso	Salario medio
Analista/Desarrollador	13,48 €/h
Jefe de proyecto	19,23 €/h

Cuadro 3.2: Tabla de los salarios medios de los recursos humanos

El valor de salario medio para el recurso que corresponde al alumno, se calcula como una media aritmética de los salarios medios para los puestos de analista [29] y desarrollador [30] en un proyecto de software. Estos valores son 16,19 €/h y 10,77 €/h, respectivamente.

Por último, para calcular el horario laboral, no se han contemplado horarios exactos, ya que la disponibilidad del alumno se ve afectada por factores externos, así como la disponi-

bilidad del jefe de proyecto, el director. Sin embargo, a efectos de seleccionar un estándar a través del cuál poder calcular el coste, se establecerá una media de 4 horas de trabajo diarias. Por contra, se contarán los fines de semana como días laborables, ya que el alumno también trabajó en el proyecto en ellos. Los días que se calcula que el jefe de proyecto trabaje, serán 10 revisiones, una por iteración, donde se le aplicará la misma media de horas estimada.

La tabla 3.3 (página 23) estima el coste total de los recursos humanos del proyecto. Este coste se expresará como el producto de la media de horas elegida y el número de días que marca el *diagrama de Gantt*, figura 3.1 (página 22), multiplicado por el salario medio de la tabla 3.2 (página 22).

Tipo de Recurso	**Horas**	**Coste**
Analista/Desarrollador	644	8.681,12 €
Jefe de proyecto	40	769,20 €
TOTAL:	684	9.450,33 €

Cuadro 3.3: Tabla de los costes estimados de los recursos humanos(en €)

En cuanto a los costes materiales, debido a que las licencias necesarias para llevar a cabo el proyecto son gratuitas, solo se cuenta el ordenador personal del alumno, cuyo precio aproximado es de 700 €. Esto, asumiendo que la vida útil del ordenador ronda las 10.000 horas, y teniendo en cuenta las 644 horas que se estiman de trabajo para el alumno; nos da un coste material de 45,08 €.

En conclusión, los gastos totales ascienden a los valores expresados en la tabla 3.4 (página 23).

Costes Humanos	9.450,33 €
Costes Materiales	45,08 €
TOTAL:	9.495,41 €

Cuadro 3.4: Tabla de los costes totales estimados del proyecto

3.6 Seguimiento y resultado final

En este apartado, se presentará el seguimiento y resultado final del proyecto y se abordarán los problemas ocurridos en su transcurso y las razones de los retrasos ocurridos. En definitiva, se tratará de obtener una visión real del desarrollo del proyecto para su posterior comparación y con lo inicialmente planeado.

3.6.1 Seguimiento

La mayor incidencia registrada en el proyecto fue un parón obligado en el mes de junio, debido a que el alumno atendía asuntos externos relacionados con otras materias. Se registró la incidencia en una última actualización del *diagrama de gantt*, donde el inicio de la iteración 5 se vio desplazado un periodo de 15 días. Esto se puede apreciar en la figura 3.2 (página 24).

Figura 3.2: Incidencia registrada en el diagrama de Gantt

3.6.2 Resultado Final

Salvando el imprevisto sucedido, reflejado en el seguimiento, no ha habido más desplazamientos, ni en coste ni en tiempo.
Por lo tanto, el *diagrama de Gantt* resulta igual que el expuesto en las figuras A.1 (página 83) y A.2 (página 84), con la salvedad de añadir el hito reflejado en la figura 3.2 (página 24).

La desviación en coste se realiza de la siguiente manera:

- **Coste Humano**: Se contabilizan los 15 días, con el horario asignado de 4 horas diarias y el salario calculado de 13,48 €/h para el alumno. En cuanto al director, al haber sido el parón justo entre dos iteraciones, es necesario contabilizar una jornada de *Sprint Review*, por la que se sumarán una jornada a mayores, con salario medio 19,23 €/h. Lo cual añade 808,8 € por parte del alumno, y 76,92 € por parte del director.
- **Coste Material**: Se contabiliza el valor perdido en los 15 días de parón. Esto refleja una perdida de valor aproximada de 4,20 € a añadir al precio estimado. Este cálculo se realiza en base a los 15 días, con las respectivas 4 horas de trabajo diarias del alumno, y los valores de vida útil y precio anteriormente expuestos.

El análisis de desviación en coste y tiempo, contando los 15 días que se han perdido refleja las cuestiones que se exponen en las tablas 3.5 (página 25) y 3.6 (página 25).

Desviación	Fecha inicio	Fecha final
Inicial	Miércoles 08/03/23	Miércoles 18/10/23
Real	Miércoles 08/03/23	Miércoles 08/11/23

Cuadro 3.5: Tabla de desviación en tiempo

Desviación	Costes Humanos	Costes Materiales	**Total**
Inicial	9.450,33 €	45,08 €	9.495,41 €
Real	10.336,05 €	49,28 €	10.385,33 €

Cuadro 3.6: Tabla de desviación en coste

Capítulo 4
Desarrollo

Este capítulo presenta el desarrollo del trabajo, elaborado a lo largo de las diferentes iteraciones expuestas en el capítulo 3.

4.1 Iteración 1

En esta primera iteración se analizarán y extenderán, en el caso de ser necesario, los requisitos, tanto funcionales como no funcionales, así como los casos de uso . Estos últimos dividen los requisitos abordados en cada una de las iteraciones, para definir su comportamiento final.

4.1.1 Análisis

Esta iteración está centrada en analizar los requisitos y casos de uso que se desarrollarán en las próximas iteraciones. Para los casos de uso, se han identificado 16, los cuales se basan en una serie de interacciones entre el usuario y la herramienta. A su vez, la herramienta se encargará de transformar los elementos de entrada del usuario, a través de una serie de plantillas de texto, en objetos dentro del interior del motor Unity. El diagrama de caso de uso general es el reflejado en la figura D.2 (página 104). Salvo que se especifique explícitamente, para lo cual se aportará la debida justificación, este será el diagrama que se utilizará durante todo el proyecto.

El objetivo más importante relacionado con el desarrollo de estos casos de uso, es minimizar y centralizar la interacción del usuario, único actor del sistema, con el motor gráfico Unity. En este sentido, los casos de uso aseguran que la mayor parte de la interacción entre el usuario y Unity, se produzca a través de las plantillas de texto y la estructura de archivos que utiliza la propia herramienta.

Se ha identificado, como ya se ha comentado, un único actor que se relaciona con el sistema. Este actor representa al usuario que utiliza la herramienta, no necesariamente el desti-

natario del producto de la herramienta, sino el desarrollador que introduzca la configuración del producto a recibir. La representación del actor se puede ver en la figura 4.1 (página 27)

Figura 4.1: Actor Usuario

En cuanto a la expansión o modificación de casos de uso, se ha considerado detallar, a efectos de contexto, los casos de uso CU-10 y CU-16, cuyos diagramas de caso de uso se encuentran en la figura 4.2 (página 27).

- **CU-10: Generar una mazmorra** : Que se dividirá en los siguientes subapartados.
  - Generar aleatoriamente la mazmorra
  - Activar la mazmorra
- **CU-16: Gestionar el flujo del juego**: Que se dividirá en los siguientes subapartados.
  - Activar/Desactivar flujo del juego
  - Generar los elementos complementarios al flujo del juego, como los elementos UI
  - Crear la instancia del juego

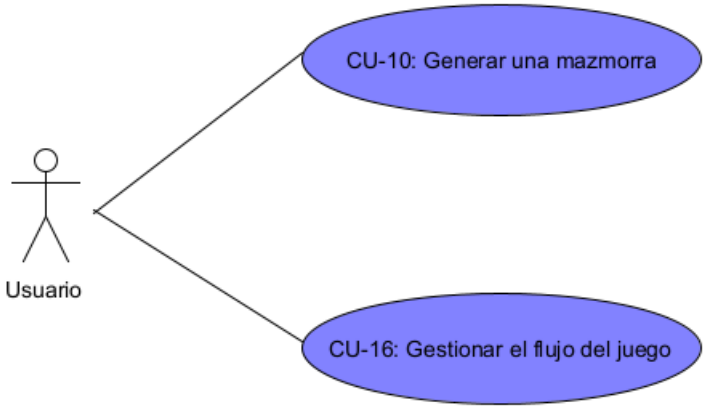


Figura 4.2: Diagramas de caso de uso CU-10 y CU-16

4.1.2 Diseño

En este sprint no se ha desarrollado ningún caso de uso, por lo que no hay diseño.

4.1.3 Implementación

En este sprint no se ha desarrollado ningún caso de uso, por lo que no hay implementación.

4.1.4 Pruebas

En este sprint no se ha desarrollado ningún caso de uso ni implementación, por lo que no hay que aplicar ninguna fase de pruebas.

4.2 Iteración 2

Esta iteración comprende la realización de los casos de uso vinculados a la misma, en este caso CU-1, CU-2, CU-3, CU-4, CU-5, recogidos en el apéndice B, tabla B.3.

4.2.1 Análisis

A continuación se presenta una asociación entre requisitos funcionales y casos de uso para el desarrollo de esta iteración:

- **CU-1**, Crear objeto de audio:
  - Req-1 : Gestionar recursos multimedia (tabla B.1)
- **CU-2**, Activar o Desactivar clips de audio:
  - Req-1 : Gestionar recursos multimedia (tabla B.1)
- **CU-3**, Especificar configuraciones y diseños:
  - Req-2 : Manejar datos de plantillas de texto (tabla B.1)
- **CU-4**, Crear Personaje:
  - Req-3 : Gestionar el personaje (tabla B.1)
- **CU-5**, Asignar comportamiento del personaje:
  - Req-3 : Gestionar el personaje (tabla B.1)

4.2.2 Diseño

Para esta iteración, teniendo en cuenta los requisitos anteriormente mencionados, se empieza a definir el marco donde se generará el producto.

Crear, Activar o Desactivar objetos de audio

Para resolver los dos primeros casos de uso CU-1 y CU-2, detallados en la tabla B.3, los cuales corresponden al mismo requisito, se desarrolla la clase *AudioController*, junto con la clase *AudioConf*. Ambas clases conforman el sistema para importar a la herramienta recursos de audio.

- *AudioConf*: Contiene la información relativa a un archivo de audio, así como el propio archivo, ya transformado en un tipo de recurso utilizable por Unity.
- *AudioController*: Es el controlador global, al que cualquier sección de código accederá, si desea reproducir algún clip de audio. Esta clase se compone de un método *PlayAudio*, derivado a su vez por otros dos métodos privados, que recibirá una instancia de la clase *AudioConf*, y reproducirá su contenido.

Es importante remarcar que, cumpliendo el requisito RNF-7, detallado en la tabla B.2, que indica seguir los principios SOLID [31] de programación; esta clase es fácilmente extensible sin modificar su comportamiento. Si se quisiera añadir más tipos de recursos de audio que los implementados actualmente, bastaría con especificar el tipo de recurso y añadir su caso a la función *PlayAudio*. Por otra parte, se respeta la responsabilidad única, ya que todo el control de audio estará centralizado en esta clase.

El diagrama de clases correspondiente a estos caso de uso se encuentra en la figura 4.3, página 29.

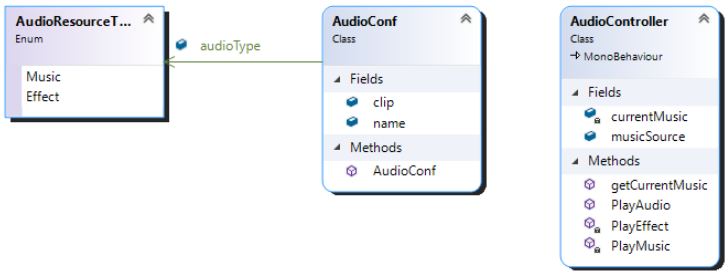


Figura 4.3: Diagrama de clases CU-1 y CU-2

Especificar configuraciones y diseños

Esta es una de las cuestiones más importantes de esta iteración, ya que es este caso de uso el que define los cimientos para las siguientes iteraciones. Uno de los objetivos del proyecto indica que, para especificar los diseños de los elementos del juego, se hará a través de plantillas especificadas en archivos de texto de tipo XML. Para poder asegurar esto, se ha seguido la estrategia detallada a continuación:

Debe definirse una clase *Desearializer*, que contendrá los métodos relacionados con la deserialización de las plantillas, cumpliendo así el principio SOLID de responsabilidad única [32], así como el de Abierto/Cerrado [33]. Estos métodos transformarán las plantillas XML en objetos dentro de Unity, que principalmente serán instancias de almacenamiento de datos que contendrán la información que se especificó en las plantillas. Es importante remarcar que estas clases son simples contenedores, no poseen métodos o constructores, más allá de heredar la clase principal *Desearializer* con su correspondiente método de deserialización, el cual se extenderá en el apartado de implementación.

Este caso de uso forma parte del RNF-4, detallado en la tabla B.2, el cual indica que los elementos del juego deben ser personalizables por el usuario. Este caso de uso representa la puerta por la cual toda la personalización del usuario fluctuará entre él mismo y Unity.

En el diagrama de clases correspondiente a este caso de uso se puede apreciar una extensión de múltiples objetos de la clase ***Deserializer***. Figura 4.4, página 30.

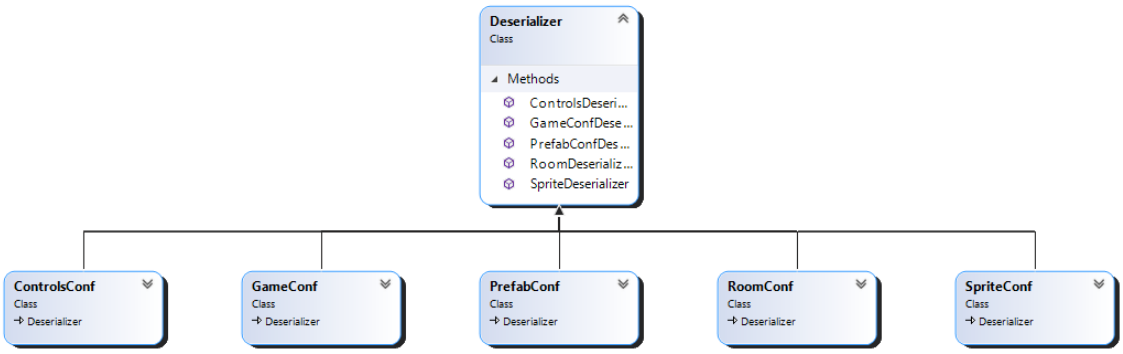


Figura 4.4: Diagrama de clases CU-3

Crear Personaje y Asignar su comportamiento

Para el diseño de las soluciones a los casos de uso CU-3 y CU-4, detallados en la tabla B.3, se han tenido en cuenta las siguientes cuestiones:

- El personaje debe ser un objeto móvil y controlado por el usuario
- El personaje debe tener un comportamiento básico que transforme la entrada del usuario en eventos dentro del juego
- El comportamiento del personaje debe adaptarse para cumplir el RNF-4 y RNF-5, y RNF-6, detallados en la tabla B.2.

En base a esto, se han creado una serie de clases que cumplimentan lo anteriormente mencionado:

- ***BehaviourConf*** y su extensión ***PlayerBehaviour***: Se introduce la clase *BehaviourConf*. Se trata de una clase que actúa de interfaz, la cual proporciona una serie de métodos a implementar por sus hijas. Estos métodos, que representan acciones básicas de una entidad, definirán el comportamiento de todas aquellas entidades que lo utilicen. *PlayerBehaviour* no es más que la implementación concreta de la clase *BehaviourConf*, para servir al propósito de controlar de manera básica al personaje. Este comportamiento se ampliará según las necesidades de la herramienta en posteriores iteraciones.
- ***Player***: Se trata del controlador de personaje. Su función será transformar el control de usuario a acciones dentro del juego. Lo hará manipulando una instancia de comportamiento, detallada en el anterior punto. Aparte de esto, el controlador se encarga de transmitir información del usuario relacionada con el entorno donde se encuentra, como su posición, o si ha alcanzado algún punto específico de su entorno. Esta última característica se expandirá en profundidad en próximas iteraciones.
- ***Input***: Esta clase recogerá la entrada del usuario y la transformará en acciones dentro del juego. ***Input*** relacionará la información recogida por entrada de teclado y la transformará en las mociones anteriormente mencionadas. Esta clase hará las correspondientes llamadas, con la acción a realizar, al controlador de personaje, detallado en el punto anterior. Esta será una clase que se ampliará en una iteración posterior.

Lo anteriormente mencionado se presenta en el diagrama de clases correspondiente a este caso de uso, que se encuentra en la figura 4.5, página 32.

4.2.3 Implementación

La implementación se realizó por separado para cada caso de uso. Sin embargo, para aplicar una noción general al sistema de archivos que se utilizará en este proyecto, Unity proporciona al desarrollador un sistema de recolección de recursos. Esto se realiza a través de

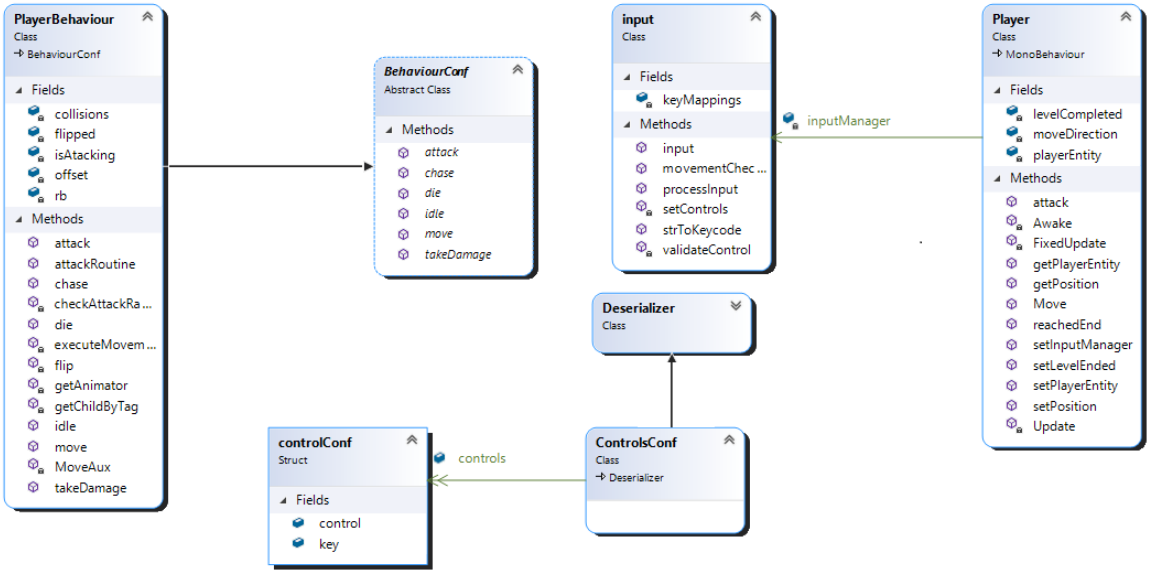


Figura 4.5: Diagrama de clases CU-4 y CU-5

la clase Resources [34], la cual permite gestionar archivos en tiempo de compilación y ejecución , antes y después de construir el producto final. Un ejemplo de lo mencionado se puede apreciar en la figura 4.6, página 32.

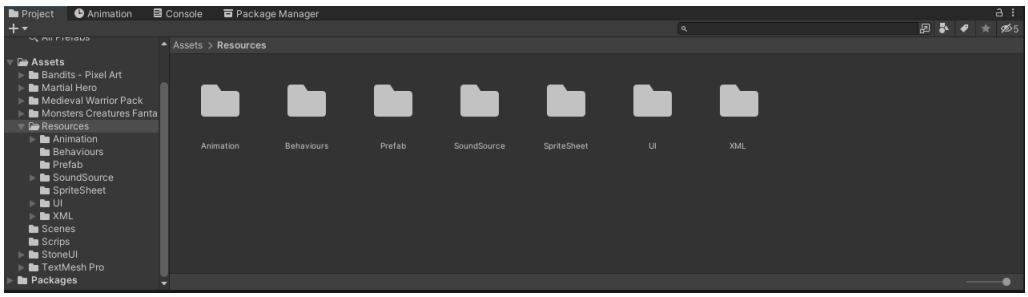


Figura 4.6: Ejemplo de estructura de archivos en la carpeta resources

- ***AudioController* y *AudioConf***: *AudioConf* es una clase que únicamente contiene un método Constructor. Su único propósito es almacenar información relativa a un clip de audio. *AudioController* en cambio, contiene 3 métodos, aparte de un método Get, de la música que actualmente se está reproduciendo.

  AudioController se basa en dos cuestiones principales:

  - Para reproducir temas de audio, utiliza un objeto global que reproduce el archivo de audio

- Para reproducir efectos de audio, crea y destruye un objeto de audio que reproduce el efecto de audio una vez

Para funcionar, la función *PlayAudio* recoge un objeto tipo *AudioConf* y lo traspasa a la función que lo procesará.

El objeto de audio es de tipo AudioSource [35], que es el tipo de objeto de audio que Unity utiliza para reproducir audio. Para poder almacenar uno y crear otros, además de ser un objeto público, *AudioController* hereda de ***MonoBehaviour*** [36], una de las clases principales de Unity.

Las características más destacables de *Monobehaviour* son que define un componente para un objeto dentro del juego, y que proporciona una serie de métodos para actuar en marcos de tiempo que utiliza el juego, los Frames.

- ***Deserializer***: El reto principal de implementación para esta clase era definir una función que deserialice los archivos XML. Finalmente se llegó a un pseudocódigo de la siguiente forma:

```
1        Deserializar(cadena) -> TipoObjeto
2            if nombre != vacío entonces
3                Xml <- cargarContenidoDelArchivo(nombre)
4                deserializador <- crearDeserializador(TipoObjeto)
5                stream <- abrirFlujo(Xml)
6                TipoObjeto <- deserializarContenido(TipoObjeto)
7                cerrarFlujo(stream)
8                devolver TipoObjeto
9            else Error
10
```

La clase que se utiliza para llevar esta tarea a cabo es XMLSerializer [37], la cual pertenece a C#. Utilizando esta clase y el sistema de recursos, los datos almacenados en las plantillas se transforman en información manejable por Unity.

- **Crear Personaje y Asignar su comportamiento**:
  - ***BehaviourConf*** y su extensión ***PlayerBehaviour***: La clase *BehaviourConf*, a pesar de que se expandirá en profundidad más adelante, se puede comentar que es una clase abstracta que define las acciones básicas, que son las siguientes:

    - Mover
    - Inactivo
    - Perseguir
    - Recibir Daño
    - Atacar
    - Morir

    La clase *PlayerBehaviour* implementa la ya mencionada *BehaviourConf*. Es importante remarcar de esta clase, así como de las de los demás comportamientos que se formalicen, que deben manejar tanto la lógica de la entidad, así como cuestiones físicas concretas que sucedan con la entidad y manejar estructuras como el Animator [38] de Unity.
    Ejemplos de esto pueden ser cálculo de colisiones de maneras concretas o asignación de estadísticas.

- ***Player***: Esta clase implementa funciones que se encargan de manejar el objeto que se sitúa en el juego. Por tanto, es una clase que extiende a *MonoBehaviour*.
  Sus dos responsabilidades, relacionadas con el objeto Jugador, son manejar las propiedades físicas del jugador en el juego, y llamar a las funciones correspondientes del comportamiento, según sean invocadas por los controles. Es importante recordar que esta clase sirve de conexión entre las expuestas en el punto anterior y en el siguiente.
- ***Input***: Almacena en una estructura, la relación entre las acciones y las teclas, por medio de la clase Input [39] de Unity.
  Esta clase proporciona un procedimiento que se encarga de comprobar qué teclas se pulsan sobre el teclado y las traduce a llamadas a acciones cuyo intermediario es la clase anteriormente explicada. Como se ha explicado en el diseño, esta clase se ampliará en posteriores iteraciones.

4.2.4 Pruebas

Para la fase de test, se han confeccionado una serie de pruebas que pueden apreciarse en la tabla 4.1, página 35.

Cuadro 4.1: Tabla de pruebas segunda Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-1 y CU-2	**Crear y reproducir archivo de audio**	Crear un script de forma manual que instancie, destruya y reproduzca varios efectos y temas de sonido.
CU-3	**Deserialización de tipos básicos**	Deserializar un archivo con tipos básicos, como enteros o cadenas, y transformarlos en objetos legibles por Unity.
CU-3	**Deserialización de tipos complejos**	Deserializar un archivo con tipos complejos, como vectores, listas y estructuras, y transformarlos en objetos legibles por Unity.
CU-4	**Comprobar entrada de teclado**	Crear un script de prueba para comprobar que al pulsar una tecla sucede algo, en este caso escribir un mensaje en la consola de Unity.
CU-4	**Crear elemento player con su controlador**	Crear un elemento player y vincularlo con su respectivo controlador, comprobar a traves del Unity Inspector que los valores están correctamente asignados.

. . . . . *(continúa en la siguiente página)* . . . . . .

Cuadro 4.1 – *(viene de la página anterior)*

Caso de Uso	Nombre de prueba	Descripción breve
CU-5	**Vincular elemento player con su comportamiento**	Vincular elemento player, creado en la anterior prueba, con su respectivo comportamiento, así como un objeto de controles. Se comprueba que el controlador llama al método correcto del comportamiento, como por ejemplo, comprobar que el elemento player se mueve.

4.3 Iteración 3

En esta iteración se desarrolla un caso de uso especial. Esto es debido a que se abordará un elemento central de la herramienta, pero que no corresponde como tal a ninguna funcionalidad. Se trata del objeto cámara, el cual es necesario para la creación del juego y para garantizar el resto de funcionalidades.

4.3.1 Análisis

Como se ha mencionado anteriormente, este elemento está fuertemente relacionado con otros casos de uso. Con el fin de aportar contexto y justificación a la presente iteración, a continuación se listan aquellos casos de uso en los que más relevancia tiene este elemento:

- ***CU-7***, Generar al personaje en el mundo
- ***CU-10***, Generar una mazmorra
- ***CU-16***, Gestionar el flujo del juego

Estos casos de uso están directamente relacionados con el resultado de esta iteración. Esto quiere decir que harán uso de operaciones que el objeto creado pondrá a su disposición.

Debido a que resulta un elemento central del proyecto, se creará un caso de uso especial, que se comunicará a través de relaciones “include” con los ya mencionados casos de uso. Se

le denominará *CU-17*, o CU: Manejo de cámara, y estará relacionado con los 3 casos de uso anteriores, como puede apreciarse en el diagrama de casos de uso, en la figura 4.7, página 37.

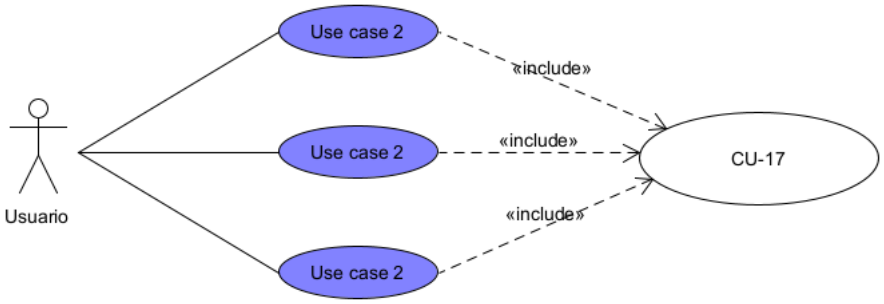


Figura 4.7: Diagrama de caso de uso para *CU-17*

4.3.2 Diseño

Codificar la lógica de una cámara puede resultar algo muy complejo. Prestando atención a algunos ejemplos de funcionalidades sencillas para cámara [40] [41], puede apreciarse como los scripts tienden a alargarse, aún añadiendo una única funcionalidad. A pesar de que el control de cámara es un elemento central de cualquier proyecto que involucre videojuegos, el propósito central de esta herramienta es proporcionar un entorno centrado en la generación de niveles.

Debido a estas causas, se ha considerado la opción de utilizar un paquete extendido y avalado por Unity. Se trata de **Cinemachine** [42], paquete el cual, aún no siendo un paquete oficial de Unity Technologies, ha sido galardonado con el premio "*Technology and Engineering Emmy Award*" [43], y es un paquete bien valorado por los usuarios, tanto de Unity [44] como externos [45].

Para aportar más contexto a la decisión de utilizar este paquete, es necesario exponer y comprender los requerimientos técnicos mínimos que debe cumplir el objeto cámara, los cuales se exponen a continuación:

- **Requerimiento de tamaño**: El tamaño de la lente de la cámara debe poder adaptarse a las necesidades de cada momento.
- **Requerimiento de vinculación**: La cámara debe estar vinculada al personaje, esto quiere decir que debe moverse con el personaje cuando este se mueva.

Para cumplir estos requerimientos, se exponen en la tabla 4.2, página 38, las soluciones que aportan tanto la cámara básica de Unity, como **Cinemachine**. Aquí se pueden apreciar las ventajas de utilizar este paquete sobre la cámara básica de Unity.

Requerimiento	Solución Cinemachine	Solución Cámara Unity
Tamaño	Modificar parámetro *Follow*	No proporciona solución directa
Vinculación	Modificar parámetro *m_Lens.OrthographicSize*	Modificar parámetros *orthographicSize* y *aspect*

Cuadro 4.2: Requerimientos y soluciones para la cámara

Ahora que se ha contextualizado la decisión de utilizar el paquete **Cinemachine**, se expondrá su utilización. Se ha creado una clase, llamada *CameraSetUp*, la cual estará asociada a un objeto **Cinemachine**. Se pondrán en práctica las soluciones expuestas en la tabla 4.2, página 38, para hacer cumplir los requerimientos del objeto.

El diagrama de clases referido a esta iteración es el de la figura 4.8, página 38.

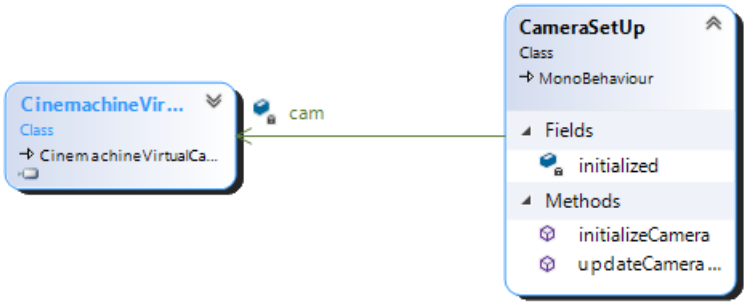


Figura 4.8: Diagrama de clases de la iteración 3

4.3.3 Implementación

Como ya se ha explicado en el diseño, el uso de el paquete **Cinemachine** facilita la implementación.

Una vez aplicadas las soluciones mencionadas en la tabla 4.2, página 38, la clase *CameraSetUp* tiene las siguientes características:

- Debe estar asociada a un objeto de juego, por lo que extiende a **MonoBehaviour**.
- Su inicialización es obligatoria, debido a que no puede tener constructor ya que **Mo-**

noBehaviour no lo permite. Por tanto, su inicialización vincula el objeto al elemento *Player*, asignando el parámetro *Cinemachine.Follow*.

- Proporciona una operación para cambiar el tamaño de la cámara recibiendo la altura.

4.3.4 Pruebas

Se describen a continuación las pruebas que se han aplicado, aparte de las aplicadas en la iteración anterior, tabla 4.3, 39. Los resultados de estas pruebas se verifican en el editor de Unity.

Cuadro 4.3: Tabla de pruebas Tercera Iteración

Caso de Uso	**Nombre de prueba**	**Descripción breve**
CU-17	**Crear cámara**	Comprobar que la cámara se crea correctamente.
CU-17	**Vincular Cámara**	Comprobar que la cámara se mueve con el personaje una vez inicializada.
CU-17	**Actualizar Cámara**	Comprobar que se actualiza el tamaño de la lente de la cámara en tiempo de ejecución, mediante un script.

4.4 Iteración 4

En esta iteración se definen e implementan las operaciones y estructuras sobre las que se construirán los niveles. Se realizarán los casos de uso CU-6, y CU-7, recogidos en el apéndice B, tabla B.3.

4.4.1 Análisis

A continuación se exponen los requisitos funcionales correspondientes a casa caso de uso:

- **CU-6**, Generar espacio de contenido:

  - Req-5: Gestionar el espacio de contenido

- **CU-7**, Generar al personaje en el mundo:
  - Req-5: Gestionar el espacio de contenido

4.4.2 Diseño

Teniendo en cuenta las relaciones anteriormente mencionadas, en esta iteración se desarrollarán dos clases, *GridAux* y *CellGrid*. Estas clases conjuntamente construyen una estructura de tipo matricial, la cual será la que contenga el contenido generado en las siguientes iteraciones.

GridAux

GridAux es una clase cuya responsabilidad es manejar un conjunto de celdas o unidades, que son una serie de espacios de información ordenados. Será una estructura basada en coordenadas que debe permitir acceso a cada unidad que contenga. Debe proporcionar maneras de eliminar o añadir contenido, así como de acceder a información relativa a las unidades que lo conforman. También podrá permitirá acceder a zonas de la estructura, es decir, conjuntos de unidades que compartan una configuración concreta especificada por el usuario o la propia herramienta. Esto se puede apreciar en el diagrama 4.9, página 41.

CellGrid

La clase *CellGrid* será la clase que representará una unidad o celda en el grid. Debe permitir almacenar la información que interese y, acceder a ella cuando sea necesario. También almacenará información relativa a sí misma, como su posición en el grid. En el diagrama 4.9, página 41, se puede apreciar como la clase ***GridAux*** almacena una colección de datos tipo ***CellGrid***.

Nótese que *CellGrid* contiene un atributo **Spawner** de tipo *RoomSpawner*. Este atributo, cuya clase se expandirá en futuras iteraciones, representa la información almacenada en la unidad. Sin embargo es importante recalcar que para la realización de esta iteración, y sobre todo para las pruebas, la información que se almacenó fue en forma de tipos básicos, como cadenas o enteros.

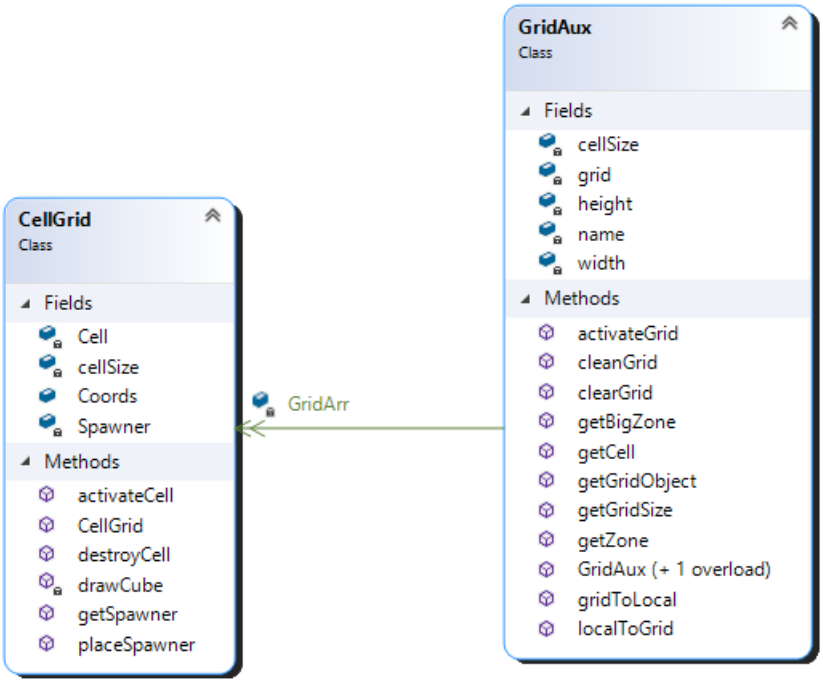


Figura 4.9: Diagrama de clases de CU-6 y CU-7

4.4.3 Implementación

GridAux

Este grid será la base de todas las operaciones relacionadas con la mazmorra. En esta estructura se colocarán todos los elementos relativos a un nivel. Por ello es muy importante definir correctamente su construcción:

El grid será una matriz bidimensional en donde cada posición representa una celda. Es muy importante tener en cuenta que el sistema de coordenadas de Unity es cartesiano, y se traduce en el espacio de juego por medio de objetos **Transform** [46], por tanto al definir la matriz existen dos opciones:

- Construir la matriz siguiendo el sistema de coordenadas cartesiano que utiliza Unity.
- Convertir al sistema clásico matricial donde las coordenadas son [fila, columna].

La segunda opción implica tener que convertir cada valor de posición cuando se llame a las operaciones de posición. Si bien es cierto que el mayor impacto será al seleccionar zonas, la primera opción permite no tener que convertir los valores, tomando en cuenta que cuando seleccionen o se comprueben zonas debe hacerse recorriendo a la inversa.

Habiendo seleccionado entonces la primera opción, una instancia de *GridAux* se puede construir de 2 maneras: con un grid ya existente, lo cual crea una copia borrando el anterior; o introduciendo sus parámetros como tamaño del grid, tamaño de tile y tamaño de celda. Hay que tener en cuenta que las coordenadas de las celdas serán cartesianas ahora, es decir, la celda [0,0] estará en el origen de coordenadas, la [0,1] encima de la anterior, y la [1,0] a la derecha de la [0,0]. Esta será una característica que no afectará al usuario en modo alguno.

Como se ya se señaló en el anterior apartado, esta clase tiene métodos, tales como *localToGrid()* o *gridToLocal()*, que permiten obtener información posicional sobre el grid. Por otro lado también contiene métodos para eliminar todas, o una selección de unidades del grid.

En cuanto a la recolección de zonas, que se hará principalmente para colocar habitaciones que ocupen más de una unidad, recogen la zona indicada indicando el inicio de esta. Es necesario convertir esta coordenada cartesiana en una matricial, ya que el usuario diseñará la habitación de izquierda a derecha y de arriba a abajo, sin embargo la conversión es sencilla, basta con recorrer la zona por columnas en vez de por filas, e ir disminuyendo el valor de las columnas en vez de aumentarlo. Así se obtendrá la zona deseada para poder comparar con el diseño del usuario. En iteraciones posteriores se ampliará este concepto.

CellGrid

Esta clase contiene la información relativa a una celda del grid, tal como su posición o su tamaño. A mayores, tambien mantiene una referencia a sí misma en el entorno de juego de Unity y la información que almacena. De esta forma, proporciona métodos para recuperar la información que contiene, destruirla o activarla.

4.4.4 Pruebas

Para la fase de test, además de las pruebas anteriormente realizadas, se han confeccionado una serie de pruebas que pueden apreciarse en la tabla 4.4, página 43.

Cuadro 4.4: Tabla de pruebas Cuarta Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-6	**Crear un grid**	Crear un grid, utilizar funciones de la clase *debug* [47] para trazar los límites correspondientes.
CU-6	**Visualizar la información de una celda del grid**	Asignar una cadena a la información contenida en una celda, recuperarla y visualizarla.
CU-6	**Visualizar la información de una zona del grid**	Asignar cadenas a la información contenida en una zona del grid, recuperarla y visualizarla.
CU-6	**Comprobar las posiciones de las celdas**	Utilizar las funciones de posición para visualizarla en la consola de Unity y comprobar que las celdas siguen el orden establecido.
CU-6	**Borrar el grid**	Utilizar las funciones para destruir el grid y comprobar que ya no es accesible.
CU-7	**Colocar al jugador en una celda**	Asignar un valor Booleano a la información de las celdas del grid, e instanciar al jugador en la celda cuyo valor sea Verdadero.

4.5 Iteración 5

En esta iteración se definirá la creación y agrupación de habitaciones. Esta es una de las tareas centrales del proyecto. Se realizarán los casos de uso CU-8, y CU-9, recogidos en el apéndice B, tabla B.3.

4.5.1 Análisis

A continuación se relacionarán los casos de uso con sus respectivos requisitos funcionales:

- **CU-8**, Crear una habitación:
  - *Req-6*: Gestionar una selección de habitaciones
- **CU-9**, Crear una selección de habitaciones:
  - *Req-6*: Gestionar una selección de habitaciones

4.5.2 Diseño

El diseño de esta iteración consta de dos partes principales:

- Crear la estructura que almacenará la habitación junto con su configuración.
- Crear la estructura que se encargará de crear conjuntos de habitaciones y almacenarlas.

Crear la estructura que almacenará la habitación junto con su configuración

Una habitación de una mazmorra, es un espacio limitado por paredes, que a su vez pueden estar cerradas o abiertas. Si están cerradas significará que hay un **Muro**, si están abiertas significará que hay una **Puerta**. Una vez definido esto, hay que definir la habitación, tal y como la especificará el usuario, y tal como la utilizará Unity.

Desde el punto de vista del usuario, especificará el diseño de la habitación por medio de una plantilla XML. En ella debe incluir una serie de datos teniendo en cuenta las siguientes restricciones:

- La plantilla de una habitación puede referenciar un recurso de audio que se reproducirá cuando se active la habitación.
- El diseño de las habitaciones estará conformado por capas, que se especificarán en orden de profundidad descendente en el parámetro **gameWorld**. Se trata de una lista que contiene los identificadores que, posteriormente, se traducirán en las tiles.

  El elemento de *gameWorld*, representará la capa de mayor profundidad y las siguientes serán capas ordenadas de mayor a menor profundidad.

  Un ejemplo visible de esto se encuentra en la figura D.1, página 102, en ella se aprecia cómo el primer elemento, etiquetado como *<ArrayOfString>*, representa la capa más

profunda de la visualización de la habitación. Típicamente esta capa será el suelo del diseño. Los siguientes elementos, etiquetados como <*ArrayOfString*>, representan capas intermedias, ordenadas de más a menos profundas, hasta llegar al último elemento. Este representa la capa de menor profundidad de la visualización de la habitación.

- Cada capa estará representada por un conjunto de caracteres , que actuarán como identificadores. Cada identificador representará una Tile u objeto. Entiéndase por objeto una tile de tamaño irregular que debe indicarse aparte. Estos identificadores se comparan con la plantilla XML que contendrá los descriptores de tiles y objetos para obtener sus traducciones.

- Es posible indicar que la habitación corresponde a un final de nivel, mediante el parámetro *levelEnd*. Si existe, significará que la habitación actual es final de nivel. Para que una habitación pueda ser final, debe tener un objeto o tile que **NO** pertenezca a los límites de la habitación, y es recomendable que tampoco pertenezca a la capa de suelo, cuyas coordenadas se especificarán en el parámetro *levelEnd*, junto a la capa a la que pertenece.

- El tamaño es un vector de la forma [Base, Altura] que debe relacionarse al tamaño de unidad especificado en la configuración de la selección.

- El parámetro *gridUnits* representa la cantidad de unidades del grid que ocupa la habitación. Las habitaciones que ocupen más de una celda deben ser de forma cuadrada.

  El parámetro *gridUnits* es una referencia que debe respetarse, tanto a la hora de cubrir el apartado *Doors*, como el de *gameWorld*.

  Una forma fácil de comprobar lo anterior es hacer la raíz cuadrada de *gridUnits*, la cual debe resultar en un valor entero e igual al número de celdas que ocupa un lado de la habitación. Un ejemplo visible de esto se encuentra en la figura 4.10, página 46, donde se aprecia la correspondencia.

- El parámetro *Doors* es una lista de vectores que representan las puertas o muros. Cada elemento de la lista describe una unidad de la habitación sobre el espacio de contenido. Posteriormente este parámetro será transformado en un objeto máscara, de la clase ***Mask***, que será el objeto con el que Unity calculará los límites de la habitación.

- El número de elementos del parámetro *Doors* debe ser igual al número de *gridUnits* especificado. Cada elemento representará los límites de unidad o celda, que tendrá tamaño

Figura 4.10: Ejemplo de correspondencia de unidades

cellSize, vector que se especifica en la configuración de la selección.

Para representar las unidades de los elementos de *Doors* se utilizará el parámetro *gameWorld*, expuesto en puntos anteriores. Esto quiere decir que debe de existir una correspondencia y un orden entre ambos parámetros.

El orden de representación los elementos del parámetro *Doors* sobre el parámetro *gameWorld* será el siguiente; el primer elemento corresponde a la celda que ocupe la esquina superior izquierda, y se continuará describiendo los límites de las celdas o unidades de izquierda a derecha y de arriba a abajo.

Un ejemplo visible de esto se encuentra en la figura 4.11, página 47. Esta figura representa una habitación de 2 unidades por 2 unidades sobre el espacio de contenido, por lo que el parámetro *gridUnits* asociado a esta habitación tendrá valor 4. En ella, el parámetro *gameWorld* cuenta con dos elementos, etiquetados como *<ArrayOfString>*, esto quiere decir que tiene dos capas de profundidad. Por otro lado, el parámetro *Doors* cuenta con

4 elementos, ya que debe describir 4 unidades del espacio. La correspondencia entre los elementos y sus unidades queda descrita con el código de colores.

Por ejemplo, el primer elemento, enmarcado en rojo, hace referencia a la unidad superior izquierda en cada capa, el segundo, enmarcado en verde, a la unidad superior derecha en cada capa, y así sucesivamente.

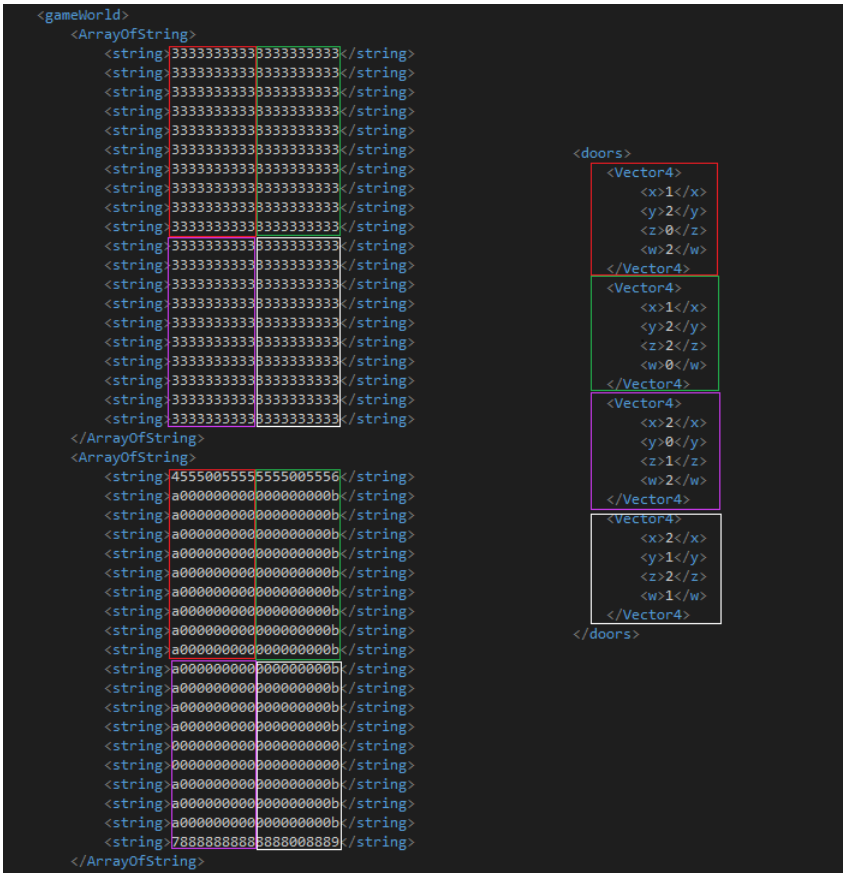


Figura 4.11: Correspondencia **gameWorld - Doors**

- Cada elemento del parámetro *Doors* es un Vector de 4 elementos con la siguiente correspondencia:
  - El primer elemento del vector se relaciona con la puerta o muro superior de la unidad a la que pertenece.
  - El segundo elemento del vector se relaciona con la puerta o muro inferior de la unidad a la que pertenece.
  - El tercer elemento del vector se relaciona con la puerta o muro izquierdo de la unidad a la que pertenece.

  - El cuarto elemento del vector se relaciona con la puerta o muro derecho de la unidad a la que pertenece.

- Para cada habitación debe indicarse las entidades que contiene y su posición. Esta cuestión pertenece a una iteración posterior, pero en esta se añade el contenedor que más adelante se llenará con los datos correspondientes. No se expandirá este asunto en esta iteración.

Es necesario que cada elemento del parámetro *Doors* simbolice la situación del lugar al que referencia. Es decir, que indique si la pared tiene una puerta, si tiene un muro o si pertenece al interior de una habitación de múltiples unidades. Continuando con el ejemplo de la figura 4.11, página 47, el primer elemento de *Doors* enmarcado en rojo, contiene una serie de valores.

En este caso, está indicando que la pared superior tiene una puerta, que la pared izquierda tiene un muro, y que las paredes derecha y abajo pertenecen al interior de una habitación. Esto se verifica en la unidad enmarcada en rojo del segundo elemento etiquetado como *<ArrayOfString>*, capa la cuál, en este caso, representa el diseño de las paredes. En la fila superior de la unidad hay dos identificadores diferentes, lo cual indica que eso es una puerta, al contrario que su pared izquierda, cuyos identificadores son todos iguales.

Desde el punto de vista de Unity, una habitación será una instancia de la clase ***RoomBuilder***, la cual tendrá las siguientes características y reglas:

- Los límites de la habitación se representarán mediante un objeto máscara, de la clase ***Mask***, traducción directa del parámetro *Doors* de la plantilla XML. Este concepto se expandirá en iteraciones posteriores.
- Si la habitación ocupa más de una celda de grid, es decir, *Doors* tiene varios elementos, cada uno se traducirá en una máscara.
- Guardará referencias a todos los generadores de entidades que la habitación tenga. Esta cuestión pertenece a una iteración posterior, pero en esta se añade el contenedor que más adelante se llenará con los datos correspondientes.

Crear la estructura que se encargará de crear conjuntos de habitaciones y almacenarlas

Una vez definidas las dos maneras de interpretar las habitaciones, el módulo que se encargará de conectar y traducir ambas será la clase ***Pool***. Se trata de una clase que tiene los siguientes objetivos o responsabilidades:

- Recibir una plantilla XML y transformar sus datos deserializados en un objeto ***RoomBuilder***.
- Almacenar una serie de habitaciones que pertenecerán a una selección.

Es importante recalcar que para que una habitación exista y sea utilizable por la herramienta, debe pertenecer a una selección. Una selección de habitaciones también será definible mediante plantillas XML, con las siguientes características:

- Debe indicarse el tamaño de celda o unidad, esta será la referencia para todas las habitaciones que pertenezcan a la selección. Se permiten celdas de base y altura desiguales.
- Se indicará el tamaño de tile y donde se encuentra la plantilla que contiene los descriptores de las mismas.
- Se indicará qué habitación será la inicial y cual será la final de la mazmorra, también la distancia mínima entre ambas.

Finalmente, la apariencia de una habitación en Unity parte de un conjunto de tiles, que también deben especificarse a través de una plantilla XML. Para cada tile debe especificarse el código, el nombre y si es o no sólido. En caso de ser un objeto también se especificará el tamaño.

Con todos estos datos, la clase ***Pool*** puede construir las habitaciones, utilizando un algoritmo que se describirá en el apartado de implementación.

El diagrama de clases correspondiente a esta iteración se encuentra en la figura D.10, página 110.

4.5.3 Implementación

Con lo descrito en el apartado anterior, las clases que se encargan de la recolección y de datos son:

- ***RoomConf***: Clase derivada de *Deserializer* que contiene la información relativa a una habitación.
- ***SpriteConf***: Clase derivada de *Deserializer* que contiene los descriptores de tiles y objetos.
- ***PoolConf***: Clase derivada de *Deserializer* que contiene la información relativa a una selección de habitaciones.

Previamente a la construcción de la selección, debe traducirse la plantilla de descriptores de tiles y objetos, en estructuras de tipo Item, que disponen tal información a la herramienta y a Unity.

La clase ***Pool*** define finalmente la forma de traducir todos estos datos en un objeto ***RoomBuilder***, el cuál representará a la habitación dentro del juego. El algoritmo se compone de los siguientes pasos:

- **Crear un nuevo *Grid* en la escena**: Para crear un sprite es necesario añadir al objeto un componente grid definido por Unity [48].
- **Crear las referencias a las puertas**: Se transforman los elementos del parámetro ***Doors*** en objetos dentro de Unity, que serán hijos del objeto habitación, y tendrán asociada una etiqueta identificativa. La clase ***RoomBuilder*** accederá a estos objetos cuando sea necesario para crear su propia máscara.
- **Crear el envoltorio de las capas**: Cada capa tendrá su referencia y se procesará por separado. Esto se hace para poder activarlas en orden una vez ensambladas, y que no haya errores de renderizado.
- **Crear los Tilemap**: Un sprite en Unity se puede representar de varias maneras, pero la más sencilla y acertada es por medio de la clase **Tilemap** de Unity [49].
- **Transformar el parámetro *gameWorld***: Se asociará cada carácter del parámetro *gameWorld* con su descriptor homólogo.
- **Gestionar colisiones**: Si la tile u objeto a instanciar es sólido, deben añadirse sus respectivos componentes **RigidBody** [50] y **Collider** [51].
- **Gestionar fin de nivel**: Si la tile u objeto a instanciar es el de final de nivel, debe etiquetarse como tal.
- **Instanciar las tile**: Se instancia la tile en el *Tilemap*
- **Ocultar las capas**: Se ocultan las capas, cambiando el parámetro *active* de su objeto, para poder activarlas en orden más adelante.
- **Situar los *Tilemaps***: Se modifica la posición local de los *Tilemaps* en el objeto, para que su referencia apunte al centro de la habitación. En caso de ser una habitación que ocupe más de una unidad del espacio de contenido, la referencia apuntará al centro de la unidad superior izquierda.

Después de esto, lo único que queda es construir la instancia de ***RoomBuilder*** con el objeto recién creado.

En las figuras 4.12, página 51 y D.8, página 108, se muestran dos ejemplos de generación de habitaciones de una y de múltiples unidades de tamaño.

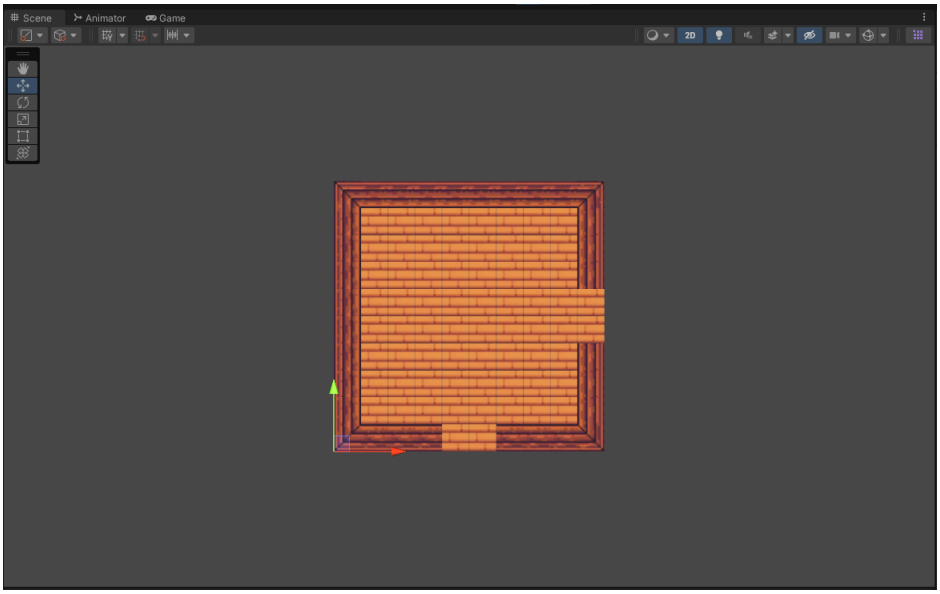


Figura 4.12: Habitación de una unidad de tamaño

4.5.4 Pruebas

Para la fase de test, además de las pruebas anteriormente realizadas, se han confeccionado una serie de pruebas que pueden apreciarse en la tabla 4.5, página 52.

Cuadro 4.5: Tabla de pruebas Quinta Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-8	**Crear y visualizar una habitación**	Comprobar que se crea una habitación correctamente. Para ello se crea un objeto ***RoomBuilder*** mediante un script separado, y se comprueba que su apariencia y la información sobre sus límites es correcta.
CU-9	**Crear y comprobar una selección**	Comprobar que se crea una selección correctamente. Para ello se crea un objeto ***Pool*** y se comprueba que la información sobre sus habitaciones es correcta.

4.6 Iteración 6

En esta iteración se desarrollará el caso de uso CU-10, recogido en el apéndice B, tabla B.3.

4.6.1 Análisis

A continuación se relaciona el caso de uso con su respectivo requisito funcional:

- **CU-10**, Generar una mazmorra:
  - *Req-6*: Generar pseudoaleatoriamente una mazmorra sobre el espacio de contenido

4.6.2 Diseño

En esta iteración el objetivo es definir un algoritmo que, sobre el espacio de contenido construido en la iteración 4, genere de manera pseudoaleatoria un laberinto de habitaciones el cual conforma una mazmorra.

Existen ciertos requisitos para generar la mazmorra:

- Debe tener un inicio y un final.
- No debe tener habitaciones superpuestas.
- Cada habitación debe tener todas las salidas conectadas, es decir, ninguna salida de la habitación debe dar a la nada o a alguna pared.
- Las habitaciones deben generarse de manera pseudoaleatoria.
- Desde cualquier habitación del espacio de contenido, debe ser posible llegar a cualquier otra habitación.

Para lograr lo descrito anteriormente, se hará uso de dos clases; ***RoomSpawner***, que se encargará de generar o expandir la habitación, y de activarla en la posición en la que se encuentre; y ***Dungeon***, que implementará el algoritmo de generación.

RoomSpawner

Este será el objeto que almacenarán las instancias de clase ***cellGrid*** dentro del grid, expuesta en la iteración 4. Su diagrama se encuentra en la figura 4.9, página 41.

La principal responsabilidad de esta clase es generar una habitación. Cada objeto de tipo ***RoomSpawner***, a los que en adelante se les referirá como *Generadores*, será una referencia a un objeto dentro del juego, en el cual se generará una habitación de manera pseudoaleatoria.

En el contexto de la mazmorra, cada generador debe poder almacenar información de sus alrededores. Para lograr esto, cada generador tendrá una máscara, es decir, una instancia de la clase ***Mask***. Una máscara contiene información sobre los límites de una habitación, se trata de una traducción directa del parámetro *Doors* de las plantillas XML, e indica si en una pared de una habitación hay un muro, una puerta, o es una pared interior.

Sin embargo, un generador utilizará la máscara, no para describirse a sí mismo, sino para describir la situación de los muros vecinos. Hay ciertas peculiaridades con el uso de la máscara con generadores, respecto de usarla con habitaciones. Por ejemplo, un generador no expandido no podrá indicar en ningún caso una pared interior, ya que significaría que el generador pertenece al interior de una unidad ya expandida, y eso sería una contradicción.

Para ilustrar este caso, se presenta un ejemplo visual de como entender la máscara de un generador, en la figura 4.13, página 54. En él se aprecia un framento del espacio de contenido, el cual posee seis unidades con sus respectivos generadores. En la figura se está analizando el generador señalizado de color rojo, su máscara debe indicar la situación de los alrededores.

Hay situadas 3 habitaciones señalizadas con cuadros verdes, con sus respectivas puertas, señalizadas en amarillo.

Para formar la máscara del generador de este ejemplo, se analiza en orden el entorno, primero arriba, después abajo, luego su izquierda, y por último su derecha. Por tanto, la máscara indica que arriba tiene una puerta, a su izquierda un muro, y nada abajo y a su derecha. La habitación de la esquina inferior izquierda no conecta con ningún borde de la unidad analizada, por lo que no se guarda información sobre ella.

Así se construirán todas las máscaras de generadores en el espacio de contenido, y se irán actualizando a medida que cambie su entorno. De esta manera, cada generador podrá juzgar si una habitación de una selección es candidata para generarse sobre él, comparando las máscaras de la misma y el propio generador, y determinando si las dos máscaras son compatibles.

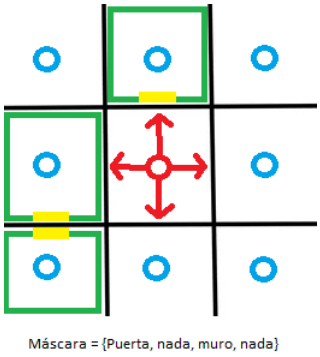


Figura 4.13: Ejemplo de construcción de máscara

Asignar a cada generador la responsabilidad de decidir que habitación se genera sobre él permite; por una parte centralizar la generación, ya que la clase ***Dungeon*** tendrá únicamente que procesar los generadores correspondientes; y encapsular el componente pseudoaleatorio, para el cual lo único que habrá que hacer es seguir los siguientes pasos:

- Comparar la máscara del generador con la de todas las habitaciones posibles.
- Seleccionar un conjunto de candidatas cuyas mascaras sean compatibles con la propia.
- Escoger de manera pseudoaleatoria una de ellas y generarla.
- En el caso de que una posible habitación ocupe más de una unidad en el espacio de contenido, deben compararse, en orden, todas las máscaras de la habitación con las máscaras de todas las combinaciones de zonas, de tamaño *gridUnits*, que es la cantidad de unidades que conforman la habitación, en las que el generador está contenido.

Para entender el último punto, en el ejemplo de la figura 4.14, página 55, se puede apreciar como está siendo procesado el generador rojo, en la coordenada (1, 1). Para saber si puede generar una habitación de 2 unidades por 2 unidades, debe comparar las posibles configuraciones de zonas que le contengan. Es decir, las combinaciones de generadores que aparecen debajo de la figura.

Las flechas rojas apuntan a todos los posibles inicios de zona que aseguren contener el generador analizado. Otro detalle importante es que hay una habitación, señalizada con el color verde, en la coordenada (2, 1). Esto implica que las configuraciones 2 y 4 ya no sean candidatas, ya que la habitación generada estaría encima de la ya existente.

Las configuraciones 1 y 3 podrán ser candidatas, dependiendo de si la habitación colocada en (2, 1) tiene o no una puerta en su muro izquierdo, y de si la habitación analizada tiene una puerta en el muro derecho de alguna de sus dos unidades situadas a la derecha.

Con todo lo explicado hasta el momento, ya se asegura:

- Que todas las habitaciones estarán conectadas entre sí
- Que las habitaciones se generarán de manera pseudoaleatoria

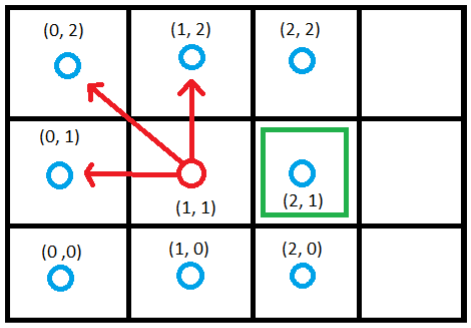


Figura 4.14: Ejemplo de comparación para habitaciones de múltiples unidades

Dungeon

Las responsabilidades de generación que la clase ***Dungeon*** tiene son reducidas gracias a la abstracción que proporciona el módulo ***RoomSpawner***. Su comportamiento se limita a

procesar en orden los generadores que haya colocados en el espacio de contenido, y actualizar los valores de los generadores vecinos, una vez que un generador haya sido expandido.

La ventaja que tiene la clase ***Dungeon***, es que conoce la situación del espacio de contenido en cada momento, por tanto puede utilizar esta información y transmitirla cuando sea necesario. El algoritmo debe cubrir los restantes requisitos de mazmorra:

- **Debe tener un inicio y un final**: El algoritmo seleccionará él mismo el generador donde colocará el inicio de la mazmorra, y colocará la salida cuando se haya expandido el suficiente terreno.
- **No debe haber habitaciones superpuestas**: Para conseguir esto, la implementación debe asegurar que ningún generador se active dos veces en la misma ejecución.
- **Desde cualquier habitación debe ser posible llegar a cualquier habitación**: Para que todas las habitaciones estén conectadas, es necesario que no se generen bloques de habitaciones separados, es decir, que la implementación debe asegurar que los generadores se procesan linealmente y en orden.

Por otro lado, esta clase también trabajará sobre la mazmorra generada, por ejemplo, situando al jugador en el inicio o activando recursos de audio o entidades de la misma.

El diagrama de clases correspondiente a esta iteración se encuentra en la figura D.9, página 109.

4.6.3 Implementación

RoomSpawner

Un generador para poder expandirse, debe primero seleccionar a los candidatos. Como se ha explicado anteriormente, deben compararse las máscaras de las posibles habitaciones con la propia. Para ello se hace uso del método *Fits* de la clase ***Mask***, que compara dos máscaras, o dos conjuntos de máscaras en orden. Si la habitación a analizar ocupa más de una unidad, debe seleccionarse una zona para compararla con la lógica que se ha explicado anteriormente.

Para conseguir la zona, se hace uso del método *getZone* de la clase ***GridAux*** que, dada una coordenada de inicio y un tamaño, devuelve un grupo de celdas correspondientes a esa zona.

Después de esto, se expandirá una habitación seleccionada pseudoaleatoriamente de la lista de candidatas, mediante *Random.Range* [52], función propia de Unity.

Para colocar esta habitación, se actualizará la máscara del generador y la referencia a su ***RoomBuilder***, con la máscara y ***RoomBuilder*** de la habitación seleccionada. En caso de que la habitación ocupe varias unidades, se actualizarán también todos los generadores que pertenezcan a la zona.

En caso de que la habitación expandida sea de múltiples unidades, se devolverá también el inicio de la zona que abarca.

Dungeon

El algoritmo de generación hace uso de un objeto Cola, implementado en la clase ***Queue***, con la peculiaridad de que manejará el tipo de dato ***RoomSpawner***. Gracias a este objeto se garantiza que todos los generadores que se introduzcan serán procesados en orden. Esto da lugar a un único bloque de habitaciones, por lo que todas ellas serán accesibles desde cualquier otra, habiendo asegurado ya, en la clase ***RoomSpawner***, que todas las habitaciones estarán interconectadas.

El algoritmo también hace uso de una lista donde almacena aquellos generadores que han sido procesados ya, previniendo así que se introduzcan generadores a la cola que ya hayan sido procesados. De esta manera se garantiza que ningún generador será procesado 2 veces y que no habrá habitaciones superpuestas.

Una vez expandido un generador, se procesarán sus vecinos. Un detalle de implementación importante es que se emplea vecindad 4, ya que las habitaciones son cuadradas. Haciendo uso de las máscaras, o zonas en caso de tener habitaciones de más de una unidad, se irán actualizando las máscaras de los generadores vecinos, si estos no han sido expandidos ya. En caso de detectar una puerta, aparte de modificarse la máscara vecina, se añadirá el generador correspondiente a la cola.

El algoritmo pedirá a los generadores que se expanda una habitación inicial en la casilla que seleccione, en este caso, el centro del espacio de contenido. También, si se ha expandido lo suficiente, y las máscaras lo permiten, el algoritmo pedirá que se expanda una habitación de final, y cuando haya sido expandida, no lo volverá a pedir. Si se completa la generación de una mazmorra sin haber colocado la habitación final, se desechará y se construirá otra, hasta ha-

berla generado con la salida. Con esto ya se han cubierto todas las necesidades de la mazmorra.

Por último, un objeto ***Dungeon*** tiene la capacidad de activar las celdas que contengan habitaciones, de limpiar el espacio de contenido de generadores que no hayan sido expandidos, de colocar y actualizar la cámara cuando cambien los tamaños de celda y de activar recursos de audio y entidades de las habitaciones cuando el jugador pase por ellas.

Se muestran varios ejemplos de generación en las figuras 4.15, página 58, 4.16, página 59 y D.11, página 111. La primera de ellas corresponde a una generación en la que no se han especificado habitaciones de múltiples unidades en el espacio de contenido, la segunda donde sí se han especificado y la última es una generación expandida, para mostrar la capacidad del sistema.

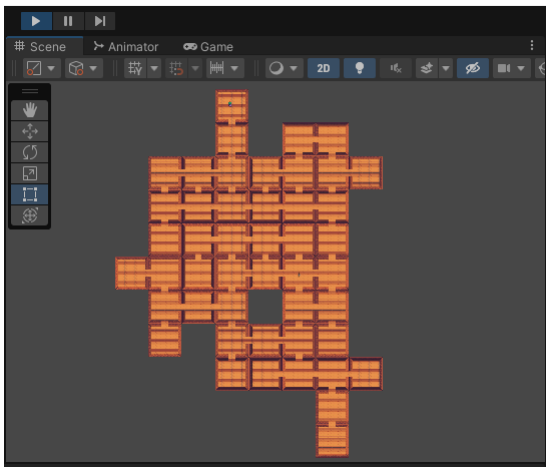


Figura 4.15: Generación sin habitaciones múltiples

4.6.4 Pruebas

Para la fase de test de la sexta iteración, se ha pasado la siguiente prueba de unidad, expuesta en la tabla 4.6, página 59.

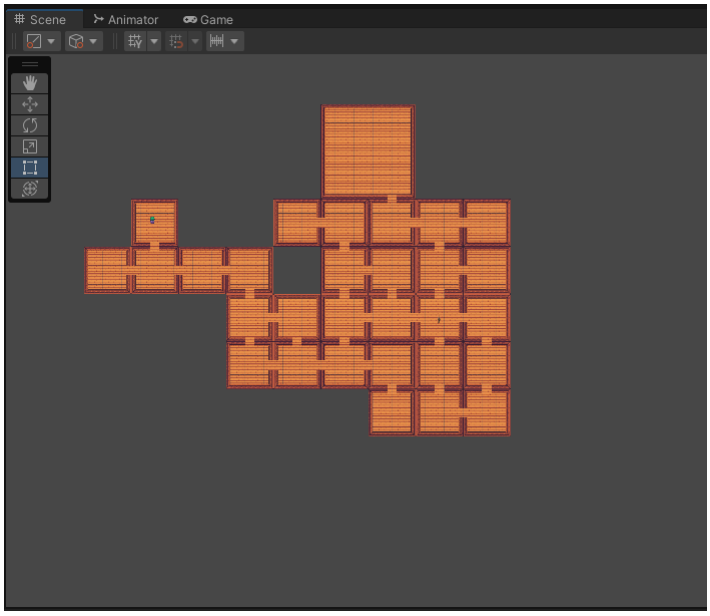


Figura 4.16: Generación con habitaciones múltiples

Cuadro 4.6: Tabla de pruebas Sexta Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-10	**Comprobar los estándares de mazmorra**	Se ha ejecutado el algoritmo de generación un elevado número de veces. Se ha comprobado que las mazmorras generadas cumplan con todos los mínimos requeridos para una instancia de mazmorra. Como se ha comentado ya, un ejemplo de generación puede apreciarse en la figura D.11, página 111.

4.7 Iteración 7

Para la séptima iteración del desarrollo, se crearán las estructuras responsables de gestionar los casos de uso CU-11, CU-12 y CU-13, recogidos en el apéndice B, tabla B.3.

4.7.1 Análisis

A continuación se exponen las asociaciones entre los casos de uso con sus respectivos requisitos funcionales:

- **CU-11**, Crear una entidad:
  - *Req-9*: Gestionar entidades
- **CU-12**, Asignar comportamiento de una entidad:
  - *Req-9*: Gestionar entidades
- **CU-13**, Generar entidades sobre el espacio de contenido:
  - *Req-8*: Generar entidades en la mazmorra

4.7.2 Diseño

Una entidad será un elemento que se generará sobre el espacio de contenido. Tal elemento corresponde a lo que en teoría de videojuegos se conoce como NPC [53]. Por tanto la tarea central será crear la lógica que manejará NPCs dentro del juego.

Para crear una entidad, se separarán y analizarán los componentes principales que, combinados, les darán vida dentro del juego:

- Su **controlador**: Este módulo es el que se comunicará con el objeto que generará la entidad sobre el espacio de contenido. Contendrá las directrices mediante las que *Unity* controlará la entidad.
- Sus **estadísticas**: Valores que representan propiedades jugabilísticas de la entidad, estos valores aportarán personalidad y diferenciación entre las entidades. También cumplen la función de abstraer la parte relacionada directamente con el juego, con el comportamiento físico que controlará Unity.
- Su **comportamiento**: Define las operaciones físicas que contienen los algoritmos que manejarán directamente objetos de Unity. Es en este módulo donde, por ejemplo, se manejarán las cuestiones relacionadas con el Animator.

Aparte de lo expuesto en estos puntos, se diseña un módulo que permitirá comunicar el controlador, que es la pieza de nivel más alto para la construcción de la entidad y la que se comunica con el entorno y espacio de contenido, con su comportamiento.

En la segunda iteración ya se pudo ver una aplicación de un comportamiento aplicado al jugador. La estructura será la misma. Esto se consigue gracias a que, tanto el controlador de entidades como el controlador del jugador, utilizarán instancias del módulo anteriormente mencionado.

Este módulo, llamado ***Entity***, servirá para abstraer los comportamientos concretos de cada entidad, de sus controladores, los cuales se comunican con los objetos que los generan, o, en el caso del jugador, con el módulo que controla la entrada por teclado.

Por tanto, el jugador será un tipo especial de entidad. Las entidades no controladas, los NPCs, podrán definir el algoritmo que los controla de manera centralizada, mientras que, para el jugador, se definirán sus métodos de control de manera modularizada, y serán invocados por su controlador cuando así lo indique la entrada de teclado.

Como ya se ha mencionado, para aportar personalidad y individualidad a las entidades, es necesario codificar los valores que representan las características de la entidad en el juego. Se han identificado las siguientes estadísticas:

- ***HP***: simboliza los puntos de vida de la entidad.
- ***Strength***: simboliza los puntos de vida que restará a su objetivo cuando ataque.
- ***Defense***: simboliza los puntos de vida que podrá mantener al recibir un ataque. Si la defensa es mayor que la fuerza del ataque recibido, no perderá ningún punto de vida.
- ***Speed***: Representa la velocidad con la que se moverá por el espacio de contenido.
- ***CoolDown***: representa el tiempo que tendrá que esperar entre lanzamientos consecutivos de ataques por la entidad.

Una vez diseñados los componentes que definirán a una entidad, es necesario especificar una manera de generarla sobre el espacio de contenido. Para ello se hará uso del módulo ***PrefabSpawner***, que tendrá las siguientes responsabilidades:

- Crear y almacenar la entidad.
- Almacenar la posición donde debe generarse la entidad.
- Generar o anular la entidad cuando sea necesario.

Los detalles de personalización, tales como la localización de la entidad, su apariencia o sus estadísticas, serán especificables a través de plantillas XML, tal como se hizo en iteraciones anteriores.

En la figura D.7, página 107, se presenta el diagrama de clases relativo a esta iteración.

4.7.3 Implementación

Para poder instanciar una entidad en el espacio de contenido, es necesario implementar los tres componentes principales de la misma, y los dos componentes externos.

Recolección de información

Se creará un nuevo objeto para almacenar la información de las plantillas XML. Este objeto, llamado ***BehaviourConf***, heredará de ***Deserializer*** y será una clase para consulta de datos. Se indirarán tanto sus estadísticas, como su diseño en este objeto.

Se recomienda encarecidamente, para que este diseño sea más sencillo, que exista un ***gameObject*** específico para señalizar el rango de ataque, por medio de un componente **Collider** de Unity. Debe estar etiquetado como "AttackRange" para que sea reconocido, tal y como se muestra en la figura 4.17, página 62.

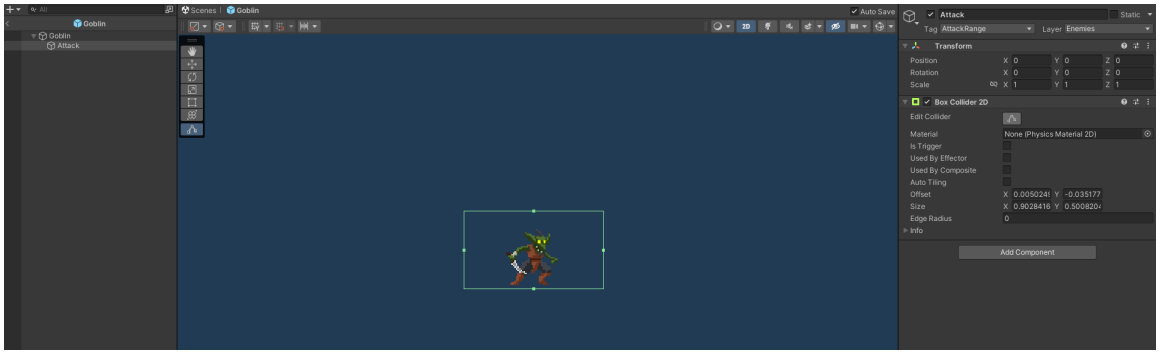


Figura 4.17: Objeto Range

Controlador

El controlador será muy parecido a la implementación de la clase ***Player***.

EntityController será la clase encargada de controlar a las entidades generadas. Proporcionará métodos para dirigir las peticiones de activación o desactivación de la entidad, y controlará que su comportamiento se ejecute en cada Frame del juego.

Al ser la clase referencia de las entidades, extenderá ***MonoBehaviour*** y será la encargada de controlar la posición inicial de la entidad.

Estadísticas

Es una clase simple que almacena los datos de las estadísticas recibidas a través de ***BehaviourConf***. Contiene operaciones para acceder de forma segura a los datos e implementa el algoritmo que se utilizará para restar vida a la entidad.

Comportamiento

La primera aproximación de esta clase, realizada en la iteración 2 para el control del comportamiento del jugador, estableció una interfaz llamada ***BehaviourConf***. De esta interfaz partirá el comportamiento de cualquier entidad dentro del juego.

Dicha interfaz contiene aquellos métodos referidos a las acciones básicas de una entidad, tales como moverse, recibir daño o morir.

Pero el objeto ***Player*** necesita una forma de control concreta que implica manejar independientemente cada una de sus acciones, desde el punto de vista de Unity, ya que el control será por teclado.

Una entidad, en cambio, no requiere tanta independencia como el jugador, ya que su comportamiento es diseñado y planificado por el desarrollador. Sus acciones están sujetas a condiciones que el desarrollador ya conoce con anterioridad, tales como moverse cuando el jugador entre en su zona de visión o estar inactivo si el jugador no está cerca.

Por ello, para encapsular el comportamiento de las entidades diferentes al jugador, se creará un una función llamada *npcBehave*, a la cual se llamará cuando una entidad se active, una vez por cada Frame.

Esta función será diseñada específicamente para implementar el algoritmo de comportamiento de la entidad y tendrá acceso al jugador y sus estadísticas, para poder modificarlas cuando sea necesario.

Este algoritmo no se diferencia demasiado de las acciones básicas diseñadas para el jugador, ya que el comportamiento del objeto dentro del juego y las físicas aplicadas, serán prácticamente las mismas, solo que deben ejecutarse de manera diferente. Para ejemplificar esto, basta con prestar atención a la función de moverse de ambos comportamientos implementados. De facto, las funciones son iguales, ambas comprueban las colisiones de su entorno antes de moverse, ambas giran su *sprite* cuando cambian de dirección y ambas activan fun-

ciones de su objeto Animator.

Sin embargo, la condición de movimiento de las entidades normales es que su objetivo esté en la misma habitación que ellas, ya que es cuando se activarán, y el objeto ***Player*** depende de llamadas externas.

Por último, para reproducir efectos asociados a las acciones es necesario crear un objeto *AudioConf*, ya explicado en la segunda iteración, con diagrama reflejado en la figura 4.3, página 29. Se obtendrá este objeto mediante *Resources.Load<AudioClip>*, función que realiza la recolección de recursos, asociado al tipo *AudioClip* que es el que representa efectos de audio.

A partir de este momento, cualquier comportamiento que desee añadirse, podrá ser utilizado por una entidad, siempre que herede de ***BehaviourConf***, implemente correctamente sus métodos, esté almacenado en su espacio correspondiente en el sistema de archivos y sea referenciado desde una plantilla XML.

Objetos

En el caso de quererse instanciar un objeto, podría hacerse de manera sencilla, creando un comportamiento, por medio de la función *npcBehave*, cuya algoritmo sea esperar a que el jugador entre en contacto con el objeto, modificar entonces sus estadísticas y ordenarle morir después. El diseño podría ser una tile estática, siempre que se referencie como un objeto **Prefab**.

Entity

Esta clase será el nexo de unión entre el comportamiento y el controlador. Cada vez que el controlador necesite acceder a alguna operación de comportamiento, se lo solicitará a esta clase.

Entity también maneja las estadísticas asociadas a la entidad, y almacena información sobre su estado, como si está muerto o si puede o no volver a atacar, en el caso del ***Player***. Para las entidades no jugables, también almacena una referencia al objetivo o *target* de la entidad, que en este caso será siempre el ***Player***, ya que es este quién las activa al entrar en las habitaciones.

Generación de la entidad

Por último, y siguiendo la referencia de los generadores de habitaciones, también se creará un ***PrefabSpawner***, solo que esta vez no será necesario que extienda ***MonoBehaviour***, ya que los generadores de entidades deben pertenecer a una habitación, por lo que pueden ser almacenados en el objeto ***RoomBuilder***, tal y como se ve en la figura D.7, página 107.

Por lo demás, la mayor responsabilidad de los generadores de entidades es construir el controlador de entidades, que al heredar de ***MonoBehaviour***, no permite construirse por medio de un constructor de clase, y debe construirse desde otro lugar, si se desea inyectarle datos externos.

4.7.4 Pruebas

Para esta iteración se han diseñado y pasado las pruebas especificadas en la tabla 4.7, página 65.

Cuadro 4.7: Tabla de pruebas Séptima Iteración

Caso de Uso	**Nombre de prueba**	**Descripción breve**
CU-11	**Crear y personalizar una entidad**	Definir una entidad por medio de ficheros XML, comprobar que su información se procesa correctamente.
CU-12	**Asignar comportamiento de una entidad**	Crear un entorno controlado, aparte del espacio de contenido, donde poder comprobar que las operaciones de la entidad se aplican correctamente.
CU-13	**Generación con entidades**	Crear diseños de habitaciones que contengan generadores de entidades. Comprobar que se generan correctamente sobre el espacio de contenido.

4.8 Iteración 8

En esta iteración se desarrollarán los casos de uso CU-14, CU-15 y CU-16, recogidos en el apéndice B, tabla B.3.

4.8.1 Análisis

A continuación se muestra la asociación entre los casos de uso y sus respectivos requisitos funcionales:

- **CU-14**, Crear un nivel:
  - *Req-11*: Gestionar el flujo del juego
- **CU-15**, Crear flujo del juego:
  - *Req-11*: Gestionar el flujo del juego
- **CU-16**, Gestionar el flujo del juego:
  - *Req-11*: Gestionar el flujo del juego
  - *Req-12*: Gestionar elementos de Interfaz de Usuario
  - *Req-13*: Gestionar el juego

En esta iteración solo se cubrirán las partes relacionadas con el Req-11. En las dos iteraciones posteriores se cubrirán los demás requisitos.

4.8.2 Diseño

El término flujo del juego, hace referencia al transcurso de los niveles del juego, sus interrupciones, su inicio y su final. Un nivel es un ensamblado de una mazmorra, con los elementos que la conforman, un jugador, y un control de que indique que el nivel ha sido superado. Estas son las características básicas que un nivel debe poseer.

Para lograr esto, un nivel de la herramienta tendrá las siguientes responsabilidades:

- Recibir un fichero de configuración de una selección de habitaciones y construir la selección.
- Crear una mazmorra con la selección obtenida.
- Recibir un fichero de configuración de recursos de audio del nivel y construir tales objetos.

- Eliminar la selección una vez usada.
- Proporcionar métodos para activar y desactivar la mazmorra.

Para lograr que los niveles del juego se ejecuten sucesivamente, se construirá una máquina de estados que controlará el paso de niveles.

Un estado de una máquina de estados será suficiente para aportar el control de nivel superado. Para que la máquina de estados pueda ser reutilizable, se construirán los estados siguiendo el patrón *Factory* [54].

Se creará una interfaz que servirá de plantilla para los estados, así, diferentes implementaciones concretas podrán utilizarse con la misma estructura de la máquina.

Es necesario establecer como la máquina de estados avanzará sobre los estados, ya que la estructura solo proporciona los métodos de uso, para poder utilizarse de diferentes maneras y con diferentes criterios.

Para ello se creará una clase llamada ***GameFlow***, que definirá el algoritmo del paso de niveles, es decir, iterará sobre la máquina de estados. Aparte, esta clase tendrá la capacidad de parar el flujo del juego, según las necesidades de cada momento.

A través de las plantillas XML, el usuario podrá especificar todos los datos necesarios sobre la estructura de niveles y los propios niveles. El usuario debe aportar información sobre niveles, como su tamaño o qué selección de habitaciones usa. También es necesario que especifique las configuraciones de las mismas y cuales son las plantillas que contienen descriptores de entidades, controles, o elementos UI. A través de estos datos se configurará el entorno donde se ejecutará el juego, iterando sobre los niveles especificados.

El diagrama de clases para los casos de uso CU-14, CU-15 y CU-16, figura D.12, página 112, representa lo expuesto en este apartado.

4.8.3 Implementación

Para la implementación de esta iteración, lo más importante es recabar todos los datos aportados de las plantillas XML. Son clases contenedoras de datos, para su posterior tratamiento, las siguientes:

- ***PrefabConf***

- ***GameConf***

A través de estas clases, Unity podrá acceder a los datos de las plantillas.

Siguiendo el patrón factoría [54], se crea una clase que implementa ***State***, ***LevelState***, la cual tendrá las siguientes responsabilidades:

- Crear un nivel
- Gestionar cuándo el jugador ha llegado al final del nivel
- Destruir un nivel

Esta clase, través de sus métodos, gestionará la relación entre un estado y un nivel. Para ello, se implementan los métodos de la clase ***State*** de la siguiente manera:

- ***Activate***: Creará una nueva instancia de la clase nivel y la activará
- ***Deactivate***: Destruirá el nivel
- ***isEnd***: Comprobará si tiene o no referencia a un nivel siguiente
- ***isRunning***: Comprobará si el jugador ha alcanzado el final de un nivel
- ***getNext***: Devolverá su referencia al siguiente nivel

Una instancia de un nivel contendrá los correspondientes objetos ***AudioConf***, ***Pool*** y ***Dungeon***, ya explicados en iteraciones anteriores. La clase ***Level*** contendrá la lógica para ensamblar los elementos de los niveles.

Sus funciones son:

- Recibir la configuración de una selección de habitaciones y crear un objeto ***Pool*** con ello.
- Crear una ***Dungeon*** por medio del objeto ***Pool***.
- Asignarle su recurso de audio al objeto ***Dungeon***.

Por último, la clase ***GameFlow*** definirá el algoritmo de flujo del juego. El pseudocódigo utilizado para resolver las necesidades del flujo del juego, es el que se presenta a continuación:

```
Algoritmo
 Estado estadoActual = IniciarEjecución()
 Mientras(la máquina se esté ejecutando)
     Mientras (el estadoActual se esté ejecutando)
         No hacer nada

     Si el estado actual no es el último
         Cargar y avanzar al siguiente nivel
     Sino
         Finalizar la ejecución
```

Aparte de esto, la clase ***GameFlow*** también gestionará elementos externos al juego como: la creación del personaje en base a los criterios de las plantillas XML, la creación del objeto cámara y la construcción de las instancias de la clase ***State*** para su posterior uso sobre la máquina de estados.

4.8.4 Pruebas

Para esta iteración, se han pasado las pruebas expuestas en la tabla 4.8, página 69.

Cuadro 4.8: Tabla de pruebas Octava Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-14	**Crear un nivel**	Crear un nivel con inicio, final y recursos de audio.
CU-15	**Crear varios niveles y conformar un flujo de juego**	Crear un flujo de juego a partir de plantillas XML, comprobar que la información se almacena correctamente.
CU-16	**Comprobar el algoritmo de flujo**	Comprobar que el algoritmo decide correctamente cuando pasar de nivel, para esto se crea un flujo simple y se comprueba que se puede llegar al final de la ejecución.

4.9 Iteración 9

Esta iteración continúa definiendo el CU-16, recogido en el apéndice B, tabla B.3.

4.9.1 Análisis

Para poder aspirar a una experiencia de juego completa, es necesario que se gestionen ciertos elementos con el flujo de juego aparte del propio flujo. Para ello, se ampliará el CU-16, para que pueda cumplir también el Req-12, Gestionar elementos de Interfaz de Usuario. Aparte de esto, se definirán al completo funcionalidades como el control por teclado.

4.9.2 Diseño

El hecho de añadir elementos de interfaz de usuario, parte de permitir explotar la creatividad de un desarrollador. Para permitir que el usuario de esta herramienta pueda aplicar la máxima creatividad posible, se han asignado las funcionalidades básicas a cada tipo de elemento UI.

Los elementos UI se especificarán como ***Prefabs***, que es el nombre que Unity da a aquellos objetos de los que queda una referencia fuera de una escena. Así la herramienta podrá encargarse solo de activar tales elementos, permitiendo al usuario definir su comportamiento.

Se han identificado los siguientes tipos de elementos UI:

- **Pantalla de carga**
- **Pantalla de Victoria**
- **Pantalla de Derrota**
- **Menú principal**
- **Pantalla de opciones**
- **Barra de Vida**

Todas ellas serán gestionadas a través de ***Prefabs*** y su información aportada a través de plantillas XML.

Por otro lado, para activar ciertos elementos, es necesario disponer de más teclas, aparte de las básicas de movimiento. Por ello, es un buen momento para ampliar la relación entre

controles, y como el usuario podrá personalizar los controles de acciones básicas.

Por tanto, se utilizará la clase ***Input***, ya creada en la segunda iteración, encargada de realizar tales relaciones. Las acciones que manejará son las acciones básicas relacionadas con las funcionalidades implementadas:

- Atacar.
- Mover el personaje hacia arriba.
- Mover el personaje hacia abajo.
- Mover el personaje hacia su derecha.
- Mover el personaje hacia su izquierda.
- Pausar el juego.

El jugador podrá definir con qué teclas desea que se realicen estas acciones. La clase ***input*** las relacionará con los efectos dentro del juego.

El diagrama de clases relativo a lo anteriormente expuesto se encuentra en la figura D.6, página 106.

4.9.3 Implementación

Para gestionar correctamente los elementos UI, la clase ***GameFlow*** almacena un estructura donde guarda las referencias a objetos de Unity. ***GameFlow*** reconocerá estos objetos ya que deben implementar las clases básicas que controlarán los elementos UI.

Estas clases proporcionan métodos para visualizar y gestionar los elementos. A continuación se da una breve descripción de ellas:

- ***PauseMenu***: A través de la clase ***Input***, pausará y reanudará el juego. También implementa ciertos métodos que pueden utilizarse o no si se desea, como cerrar el juego modificar el volumen.
- ***LoadScreen***: Proporciona un método para superponer una pantalla de carga a la creación de un nivel.
- ***DeathScreen***: Es activada al morir el jugador a través de su método *Display*.
- ***VictoryScreen***: Igual que el anterior elemento.

- ***healthBar***: Gestionará la barra de vida en base a los puntos de salud que le queden al jugador.

A su vez, se proporciona una escena, denominada UI, donde poder generar el menú principal al gusto de cada usuario, basta con enlazar el botón de "Jugar", con el método de ***LoadScreen***, para cargar la escena de juego.

Por otro lado, la clase ***Input***, en vez de relacionar elementos con su significado, relacionará códigos de tecla con sus acciones.

Cabe destacar, que el ***Player*** almacenará la referencia única a este objeto, ya que el ***Player***, sumado a la interacción del usuario son las piezas centrales de la ejecución del juego.

El ***Player*** realizará comprobaciones a través de métodos de la clase ***Input***, como podría ser *movementChecker* y, a su vez, la clase ***Input*** llamará a métodos de ***Player*** cuando sucedan las diferentes interacciones.

4.9.4 Pruebas

Para esta iteración, se han confeccionado una serie de pruebas, expuestas en la tabla 4.9, página 72.

Cuadro 4.9: Tabla de pruebas Novena Iteración

Caso de Uso	**Nombre de prueba**	**Descripción breve**
CU-16	**Crear elementos UI**	Crear diferentes elementos UI y comprobar que se generan correctamente.
CU-16	**Comprobar la correspondencia de controles**	Cambiar varias veces el fichero de controles y comprobar que la herramienta responde satisfactoriamente.

4.10 Iteración 10

En esta última iteración se presentan los últimos retoques al CU-16 y se presenta un ejemplo de uso de la herramienta.

4.10.1 Análisis

El último paso de este proyecto será ampliar el CU-16 para poder crear el juego. Esto responde al Req-10, Gestionar el juego.

4.10.2 Diseño

Es necesario que la acción en Unity parta de un objeto que ya está creado en la escena. Por ello se creará el objeto **Director**. Este objeto contendrá el componente ***Game***, que iniciará toda la ejecución.

A este objeto el usuario debe proporcionarle el nombre de la plantilla que contiene la configuración general del juego.

El objeto director contendrá también el componente ***AudioController***, para gestionar y centralizar la referencia general al objeto de audio.

A continuación se presentan tanto el descriptor del objeto director en Unity, como la clase ***Game***. Figuras D.4, página 105, y D.5, página 105.

4.10.3 Implementación

La implementación de la clase ***Game*** es muy simple. Buscará el elemento *Canvas* [55] donde se construirán los elementos UI, deserializará la plantilla de configuración general del juego, e iniciará la ejecución del algoritmo de flujo de juego, definido en la iteración 8.

Por otro lado, se ha confeccionado un ejemplo de uso para la herramienta, que se puede encontrar en la siguiente referencia [Demo].

Para acceder al ejemplo, lo único que hay que hacer es descargar la carpeta que presenta el repositorio y ejecutar ***TFG.exe***.

Los controles para la Demo son los siguientes:

- **Atacar**: Q
- **Moverse Arriba**: Flecha arriba
- **Moverse Abajo**: Flecha abajo
- **Moverse Izquierda**: Flecha izquierda
- **Moverse Derecha**: Flecha derecha

- **Pausar el juego**: Tecla Escape

4.10.4 Pruebas

La única prueba que se ha llevado a cabo para esta iteración, es la presentada en la tabla 4.10, página 74.

Cuadro 4.10: Tabla de pruebas Décima Iteración

Caso de Uso	Nombre de prueba	Descripción breve
CU-16	**Crear el ejemplo**	Construir el ejemplo, comprobar que es pasable y comprobar que incorpora correctamente todas las funcionalidades desarrolladas.

4.11 Cierre del Desarrollo

Finalmente y de manera breve, se realiza un análisis del resultado técnico de las 10 iteraciones realizadas.

Durante el transcurso de este proyecto, se ha buscado la manera de proporcionar soluciones a los problemas planteados en la presentación del presente trabajo, y de diseñar e implementar tales soluciones en un marco de trabajo cumpliendo los estándares establecidos.

En este sentido, en el aspecto técnico, relacionado con los requisitos funcionales y casos de uso, expuestos en el apéndice B, tablas B.1 y B.3; se obtiene una herramienta capaz de:

- Gestionar elementos de personalización de juego externos a la generación y recoger tales elementos, detalles y configuraciones del juego a través de plantillas XML.
- Convertir la información en objetos del juego.
- Generar mazmorras de manera pseudoaleatoria.
- Gestionar un flujo de juego y su ciclo de vida y crear una instancia de un juego.

Por otro lado, a nivel de los estándares establecidos, relacionado con los requisitos no funcionales, expuestos en el apéndice B, tabla B.2, la herramienta ha resultado estar a la altura

de ellos, sobre todo en materia de personalización, generación procedimental y pseudoaleatoriedad, que son los tres pilares principales que sostienen este proyecto.

En conclusión, el desarrollo de la herramienta se ha centrado principalmente en resolver sus requerimientos centrales, tales como la generación y el diseño de elementos, y aporta soluciones válidas y flexibles para aquellas cuestiones no relacionadas directamente con estos requerimientos, tales como la generación de elementos UI.

Capítulo 5

Conclusiones

Durante el transcurso de este trabajo, se han presentado diferentes módulos que sirven al propósito de generar contenido de manera impredecible. Los objetivos del proyecto, marcados en el apartado 1.2, enmarcan una herramienta capaz de presentar resultados, tanto a nivel de componentes de un juego, tales como la creación automática de niveles para su posterior uso en otros ámbitos, como la creación de una instancia de un juego.

No existen muchas herramientas que abarquen los objetivos propuestos para ésta; de haberlas, serían un activo importantísimo para cualquier proyecto empresarial relacionado con videojuegos. Este proyecto se centra en presentar una base para lo que podría ser una herramienta **fabricante** de juegos. Una herramienta capaz de automatizar la mayoría de operaciones a un desarrollador, para que pueda centrarse en cuestiones de personalización.

El cumplimiento de los objetivos, ha condensado la mayor cantidad de esfuerzo mecánico y técnico que presenta la creación de un videojuego con las características expuestas en la introducción del presente TFG. Siendo consciente de que esta herramienta no es más que un comienzo en el camino que lleva a la automatización de los esfuerzos anteriormente mencionados, el sistema ha resultado ser un *software* sólido, con capacidad de expansión y que produce resultados satisfactorios para poder trabajar sobre ellos.

En cuanto al cumplimiento de los objetivos expuestos en el apartado 1.2, el desarrollo realizado y expuesto en el capítulo 4 cumple todo lo planteado al principio del presente trabajo. La síntesis del desarrollo, apartado 4.11, capítulo 4, expone de manera más detallada como el resultado final satisface los objetivos recogidos.

A nivel personal, el ámbito de los videojuegos ha estado presente durante toda mi vida, ha nutrido mi experiencia y contribuido a sostener mi desarrollo personal. Ello es la razón

principal por la que escogí explorar el campo de la informática y acabar eligiendo esta carrera como comienzo de mi camino laboral, y también forma parte de mis objetivos de vida el hecho de dedicarme al desarrollo de los videojuegos.

Amén de estas cuestiones, el desarrollo de este proyecto ha aportado diferentes beneficios y creado cierto impacto en mi perfil como desarrollador y programador. Comenzando por que es el primer proyecto real completado de principio a fin, íntegramente por mí, y no solo eso, sino que he enriquecido mi capacidad, tanto de programación, como de búsqueda de soluciones, como de organización para resolver los problemas.

Entre las nuevas habilidades adquiridas, la que más valor aporta, según mi parecer, es la capacidad de plantear problemas que, en mi experiencia como jugador, han aparecido alguna vez, y buscar la manera de solventarlos en mi propia herramienta. También es crucial destacar el profundo conocimiento y familiarización adquiridos sobre la herramienta Unity, la cual abre las puertas al mundo del desarrollo de videojuegos. No cabe duda de que mi perfil laboral se ve beneficiado con la adquisición de automatismos y recursos, sobre todo en materia de planificación de proyectos de cara a trabajar en empresas con equipos de múltiples personas.

No obstante, hay una cuestión más, la cuál es importante analizar antes de terminar.

Proyección comercial

Cumple esta herramienta con ciertos estándares suficientes para su comercialización?:

La respuesta a esta pregunta tiene matices. Por una parte, una de las bases de realización de este proyecto, expuestas en el apartado de análisis de mercado del capítulo 2, era asegurar que la herramienta sería puesta a disposición de los usuarios de manera gratuita, a través de la *Asset Store* de Unity.

Por otro lado, analizando el mercado gratuito, se aprecia como casi no hay herramientas con una lista de objetivos tan extensa como la presente, y, analizando el mercado de pago, se puede comprobar como ciertas herramientas mejor valoradas y mejor vendidas de la *Asset Store* en el momento de finalización de este trabajo, tales como **Male 4** [56], o **Huge Playground Post Apocalyptic** [57], son herramientas que aportan diseños de entorno o personajes, siendo este el punto más valorado en la *Asset Store* de Unity. Herramientas que, siendo de un refinamiento muy alto, no presentan el nivel de complejidad que se ha manejado en el trans-

curso del proyecto.

Teniendo esto en cuenta, la principal pretensión es cumplir las bases establecidas en el capítulo 2, sin embargo, en el caso de añadir algunas funcionalidades a mayores, podría considerarse la inclusión del modelo *Freemium* [58], previa fase de discusión sobre qué funcionalidades ameritan ser de pago.

Capítulo 6

Trabajo Futuro

6.1 Mejoras Futuras

A pesar de que la herramienta cumple con los objetivos designados en la planificación, a continuación se presentan una serie de críticas o limitaciones sobre el funcionamiento actual y posibles mejoras de cara a un futuro.

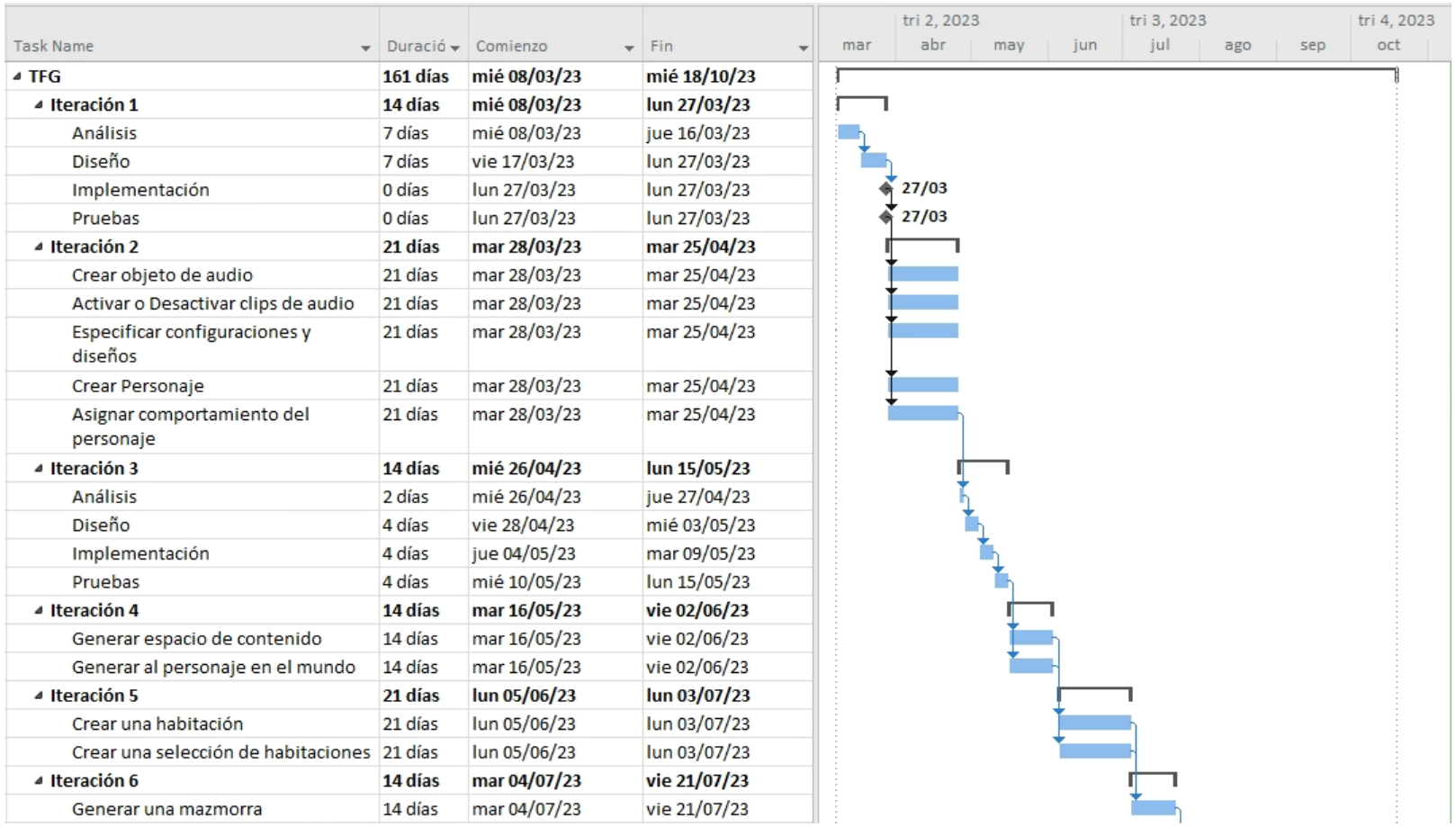
- **Módulo de diseño**: A pesar de que la recolección de datos mediante plantillas XML es un sistema fuerte, puede resultar tedioso para el usuario tener que cubrir los diseños de las habitaciones. Por ello, sería interesante añadir un módulo gráfico capaz de cubrir la parte de diseño y la traducción a las plantillas XML.
- **Control de cinemáticas**: Un aspecto del diseño de un juego que esta herramienta no cubre es el sistema para cinemáticas. Una posible mejora de cara al futuro podría ser añadir alguna clase de control para poder incorporarlas al resultado. Aunque se puede lograr un pseudocomportamiento de cinemática por medio de las clases de UI, esta herramienta no está diseñada para introducir cinemáticas.
- **Expansión de la cola a múltiples tipos**: Al trabajar con estados y una máquina de estados que permite recibir infinitas cantidades de tipos de estado, pudiera ser útil añadir la cola como herramienta versátil, ya que ahora mismo está especializada en ***RoomSpawner***.
- **Añadir habitaciones múltiples irregulares**: Actualmente, la restricción de la herramienta es a habitaciones cuadradas, aunque esto ya es suficiente para tener una amplia generación y una alta rejugabilidad. Sin embargo, es interesante considerar la posibilidad de añadir la capacidad de introducir habitaciones de más de una celda, con formas irregulares.

- **Añadir tiles volátiles**: Actualmente no es posible añadir una tile que tenga un comportamiento, así como un tiempo de vida. Aunque se puede codificar una tile como un objeto, podría también añadirse un modulo que controlase la capacidad de una tile para desaparecer.
- **Añadir mecanismo de cierre y apertura de puertas**: Enlazado con lo anterior, quizá para algunos usuarios sería interesante añadir un sistema de apertura y cierre de pasillos, para que el jugador tenga que realizar algún objetivo en la habitación.

Apéndices

Apéndice A

Diagramas de Gantt



Task Name	Duració	Comienzo	Fin
TFG	**161 días**	**mié 08/03/23**	**mié 18/10/23**
Iteración 1	**14 días**	**mié 08/03/23**	**lun 27/03/23**
Análisis	7 días	mié 08/03/23	jue 16/03/23
Diseño	7 días	vie 17/03/23	lun 27/03/23
Implementación	0 días	lun 27/03/23	lun 27/03/23
Pruebas	0 días	lun 27/03/23	lun 27/03/23
Iteración 2	**21 días**	**mar 28/03/23**	**mar 25/04/23**
Crear objeto de audio	21 días	mar 28/03/23	mar 25/04/23
Activar o Desactivar clips de audio	21 días	mar 28/03/23	mar 25/04/23
Especificar configuraciones y diseños	21 días	mar 28/03/23	mar 25/04/23
Crear Personaje	21 días	mar 28/03/23	mar 25/04/23
Asignar comportamiento del personaje	21 días	mar 28/03/23	mar 25/04/23
Iteración 3	**14 días**	**mié 26/04/23**	**lun 15/05/23**
Análisis	2 días	mié 26/04/23	jue 27/04/23
Diseño	4 días	vie 28/04/23	mié 03/05/23
Implementación	4 días	jue 04/05/23	mar 09/05/23
Pruebas	4 días	mié 10/05/23	lun 15/05/23
Iteración 4	**14 días**	**mar 16/05/23**	**vie 02/06/23**
Generar espacio de contenido	14 días	mar 16/05/23	vie 02/06/23
Generar al personaje en el mundo	14 días	mar 16/05/23	vie 02/06/23
Iteración 5	**21 días**	**lun 05/06/23**	**lun 03/07/23**
Crear una habitación	21 días	lun 05/06/23	lun 03/07/23
Crear una selección de habitaciones	21 días	lun 05/06/23	lun 03/07/23
Iteración 6	**14 días**	**mar 04/07/23**	**vie 21/07/23**
Generar una mazmorra	14 días	mar 04/07/23	vie 21/07/23

Figura A.1: Diagrama de Gantt con Casos de uso 1

Task Name	Duració	Comienzo	Fin
Iteración 6	**14 días**	**mar 04/07/23**	**vie 21/07/23**
Generar una mazmorra	14 días	mar 04/07/23	vie 21/07/23
Iteración 7	**21 días**	**lun 24/07/23**	**lun 21/08/23**
Crear una entidad	21 días	lun 24/07/23	lun 21/08/23
Asignar comportamiento de una entidad	21 días	lun 24/07/23	lun 21/08/23
Generar entidades sobre el espacio de contenido	21 días	lun 24/07/23	lun 21/08/23
Iteración 8	**14 días**	**mar 22/08/23**	**vie 08/09/23**
Crear un nivel	14 días	mar 22/08/23	vie 08/09/23
Crear flujo del juego	14 días	mar 22/08/23	vie 08/09/23
Activar o desactivar el flujo del juego	14 días	mar 22/08/23	vie 08/09/23
Iteración 9	**14 días**	**lun 11/09/23**	**jue 28/09/23**
Analisis	2 días	lun 11/09/23	mar 12/09/23
Diseño	4 días	mié 13/09/23	lun 18/09/23
Implementación	4 días	mar 19/09/23	vie 22/09/23
Pruebas	4 días	lun 25/09/23	jue 28/09/23
Iteración 10	**14 días**	**vie 29/09/23**	**mié 18/10/23**
Análisis	2 días	vie 29/09/23	lun 02/10/23
Diseño	4 días	mar 03/10/23	vie 06/10/23
Implementación	4 días	lun 09/10/23	jue 12/10/23
Pruebas	4 días	vie 13/10/23	mié 18/10/23

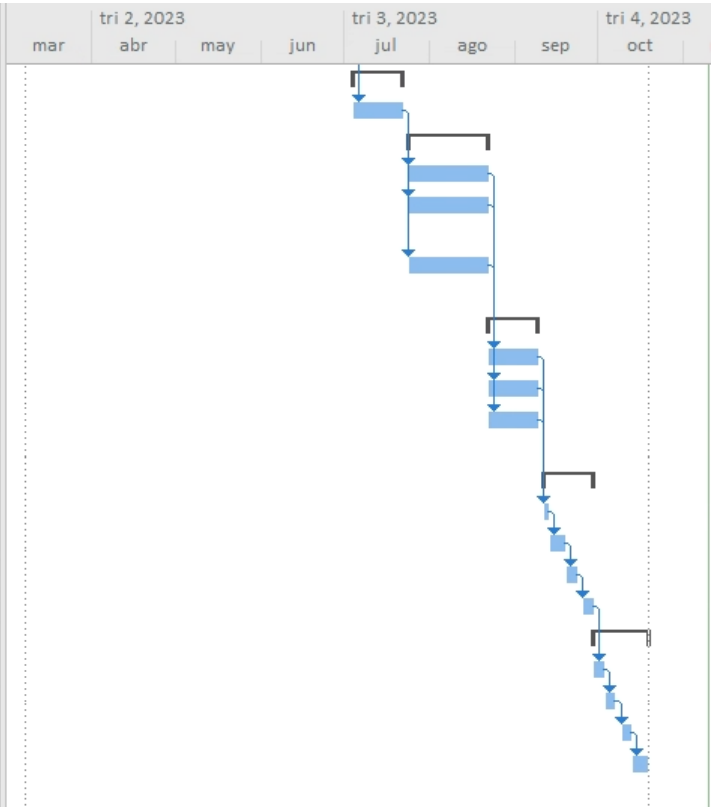


Figura A.2: Diagrama de Gantt con Casos de uso 2

Apéndice B

Tablas de requisitos y Casos de uso

En este apéndice se presenta la tabla de requisitos, tanto funcionales como no funcionales, y la tabla de casos de uso correspondientes a la herramienta desarrollada.

Cuadro B.1: Tabla de requisitos funcionales

ID	Nombre	Descripción
Req-1	**Gestionar recursos multimedia**	Identificar, recoger y almacenar todos los tipos de recursos que vaya a necesitar el juego para su uso posterior. Transformar aquellos objetos que lo necesiten para ser utilizados por Unity y asignarlos a sus correspondientes elementos del juego.
Req-2	**Manejar datos de plantillas de texto**	Acceder a los ficheros de texto que contienen la configuración de los elementos del juego. Deserializarlos y transformarlos en tipos de datos legibles y utilizables por la Unity.
Req-3	**Gestionar el personaje**	Crear al personaje jugable, instanciarlo, vincularlo con los controles y gestionar su comportamiento.
Req-4	**Gestionar cámara**	Crear e inicializar un objeto cámara, vincularla con el personaje y configurar sus parámetros.

. . . . . *(continúa en la siguiente página)* . . . . . .

Cuadro B.1 – *(viene de la página anterior)*

ID	Nombre	Descripción
Req-5	**Gestionar el espacio de contenido**	Crear, acceder y modificar la información del espacio de trabajo, como su tamaño o sus posiciones locales.
Req-6	**Gestionar una selección de habitaciones**	Crear, modificar y destruir habitaciones. Almacenarlas en conjuntos que compartan una configuración concreta.
Req-7	**Generar aleatoriamente una mazmorra sobre el espacio de contenido**	Combinar aleatoriamente habitaciones de una selección sobre el espacio de contenido, colocarlas y activarlas.
Req-8	**Generar entidades en la mazmorra**	Generar entidades en las habitaciones de la mazmorra.
Req-9	**Gestionar entidades**	Crear una entidad, asignarle su apariencia y comportamiento, así como destruirla.
Req-10	**Gestionar un nivel**	Crear, destruir y modificar un nivel en base a una configuración.
Req-11	**Gestionar el flujo del juego**	Almacenar un conjunto de niveles e inicializarlos para que puedan ser iterables. Activar y pausar el flujo, así como desactivarlo cuando haya terminado.
Req-12	**Gestionar elementos de Interfaz de Usuario**	Crear, configurar y destruir los elementos de interfaz de usuario, así como activarlos y desactivarlos cuando sea necesario.
Req-13	**Gestionar el juego**	Almacenar y ensamblar los elementos del juego y crear el propio juego.

Cuadro B.2: Tabla de requisitos No funcionales

ID	Nombre	Descripción
RNF-1	**Generación Procedimental**	La base del sistema debe ser la Generación Procedimental.
RNF-2	**PseudoAleatoriedad**	El contenido no especificado por el usuario debe generarse correctamente de manera pseudoaleatoria.
RNF-3	**Estándares de mazmorra**	Las mazmorras deben seguir unas reglas básicas, tales como tener una salida y una entrada, o que sus habitaciones estén totalmente conectadas.
RNF-4	**Personalización**	El usuario debe poder introducir una serie de datos de personalización y el resultado debe generarse en base a esos datos.
RNF-5	**Escalabilidad**	El sistema debe introducir mecanismos para añadir fácilmente y de forma segura nuevas funcionalidades.
RNF-6	**Control de excepciones**	Las excepciones deben estar controladas y presentadas al usuario de forma precisa.
RNF-7	**Principios SOLID de programación**	El código debe seguir en la medida de lo posible los principios SOLID de programación.

Cuadro B.3: Casos de uso iniciales del sistema

ID	Nombre	Actores	Descripción
CU-1	**Crear objeto de audio**	Usuario, sistema, Unity	El usuario debe poder especificar un recurso de audio y el sistema lo transformará en algo manipulable por Unity.

. . . . . *(continúa en la siguiente página)* . . . . .

Cuadro B.3 – *(viene de la página anterior)*

ID	Nombre	Actores	Descripción
CU-2	**Activar o Desactivar clips de audio**	Usuario, Sistema	El usuario debe poder indicar qué sonido debe reproducirse y cuándo, además de en qué habitaciones, mazmorras o entidades y el sistema indicará a Unity cuando debe reproducirlos.
CU-3	**Especificar configuraciones y diseños**	Usuario, Sistema	El usuario debe poder indicar la configuración y diseño de los elementos que conforman el juego, a través de ficheros de texto, y el sistema recogerá los datos y los transformará en información legible por Unity y el sistema.
CU-4	**Crear Personaje**	Usuario, Sistema	El usuario debe poder aportar la configuración de un personaje y el sistema la transformará en el personaje dentro de Unity.
CU-5	**Asignar comportamiento del personaje**	Usuario, Sistema	El usuario debe poder aportar un comportamiento del personaje a través de un *script* y el sistema lo traducirá a un comportamiento dentro de Unity.
CU-6	**Generar espacio de contenido**	Usuario, Sistema	El usuario debe poder indicar los parámetros del espacio de contenido y el sistema los transformará en un objeto manipulable por Unity donde generar el contenido del juego.

. . . . . *(continúa en la siguiente página)* . . . . . .

Cuadro B.3 – *(viene de la página anterior)*

ID	Nombre	Actores	Descripción
CU-7	**Generar al personaje en el mundo**	Usuario, Sistema	El usuario debe poder indicar en que habitación desea que el personaje se genere. El sistema debe generar al personaje en el espacio de contenido y asignarle una cámara para que sea visible.
CU-8	**Crear una habitación**	Usuario, Sistema	El usuario debe poder aportar el diseño y configuración de una habitación y el sistema lo transformará en un objeto manipulable por Unity.
CU-9	**Crear una selección de habitaciones**	Usuario, Sistema	El usuario debe poder especificar que habitaciones pertenecen a una misma selección. El sistema transformará esta información en una estructura en Unity que contenga todas las habitaciones especificadas.
CU-10	**Generar una mazmorra**	Usuario, Sistema	El usuario debe poder aportar todos los elementos correspondientes a una mazmorra. El sistema generará una mazmorra dentro de Unity, en el espacio de contenido, utilizando un algoritmo pseudoaleatorio, basado en programación procedimental.
CU-11	**Crear una entidad**	Usuario, Sistema	El usuario debe poder aportar la configuración de una entidad y el sistema la transformará en la entidad dentro de Unity.

. . . . . *(continúa en la siguiente página)* . . . . . .

Cuadro B.3 – *(viene de la página anterior)*

ID	Nombre	Actores	Descripción
CU-12	**Asignar comportamiento de una entidad**	Usuario, Sistema	El usuario debe poder aportar un comportamiento de una entidad a través de un *script* y el sistema lo traducirá a un comportamiento dentro de Unity.
CU-13	**Generar entidades sobre el espacio de contenido**	Usuario, Sistema	El usuario debe poder indicar donde desea que se generen sus entidades. El sistema generará esas entidades en el espacio deseado, dentro de Unity.
CU-14	**Crear un nivel**	Usuario, Sistema	El usuario debe poder indicar los elementos que conformarán un nivel del juego. El sistema debe crear el nivel dentro de Unity utilizando esa información.
CU-15	**Crear flujo del juego**	Usuario, Sistema	El sistema debe poder indicar el orden y descripción de los eventos que ocurran en el juego. El sistema lo transformará en una secuencia de eventos dentro de Unity.
CU-16	**Gestionar el flujo del juego**	Usuario, Sistema	El usuario debe poder indicar al sistema como y cuando quiere que empiece y acabe el flujo del juego, así como indicar los elementos pertenecientes a la interfaz de usuario que interfieren en el mismo. El sistema lo traducirá en puntos de inicio, final e hito de juego dentro de Unity.

Apéndice C

Manual de usuario

En este apéndice se presenta el manual de usuario.

C.1 Primeros pasos

Para familiarizarse con el sistema que seguirá la aplicación, hay que comenzar hablando del sistema de archivos. Con la herramienta instalada, se encontrará fácilmente la carpeta **Resources**. En ella se almacenarán todos los archivos. El sistema es el indicado en la figura C.1, página 91.

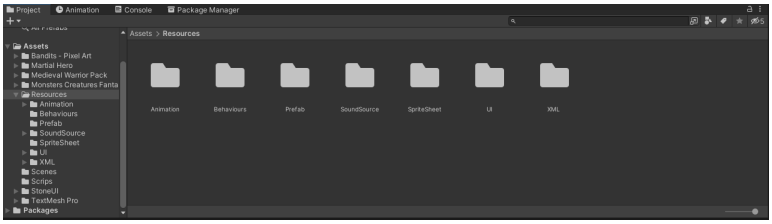


Figura C.1: Sistema de archivos

Se pueden apreciar siete carpetas, las cuales contendrán lo siguiente:

- **Animation**: Los diferentes archivos de animación, *AnimationClip* y *AnimatorController* de las entidades que estarán en el juego.
- **Behaviours**: Almacenará los scripts que describan comportamientos de entidades.
- **Prefab**: Los objetos *Prefab*, tanto del player como del resto de entidades.
- **SoundSource**: Estarán tanto los efectos como *clips* musicales.
- **SpriteSheet**: Estarán los *SpriteSheet* utilizados para diseñar niveles.

- **UI**: Hay dos subcarpetas, una contendrá los scripts asociados a elementos UI que se decida añadir, la otra, los *Prefabs* asociados a ellos.
- **XML**: Contendrá todos los archivos de configuración XML que, como usuario de la herramienta, se deben cubrir.

Preparación de los Sprites

Para poder utilizar una hoja de sprites en esta herramienta, es necesario procesarla. Para ello, una vez el sprite esté dentro de su carpeta, se accederá a su parámetro *Sprite Editor* en el inspector. Este parámetro se muestra en la figura C.2, página 98, enmarcado en rojo.

Una vez abierto, es necesario acceder a la opción **Slice**, en la esquina superior izquierda, y configurarlo como se muestra en la imagen de la figura C.3, página 98, sustituyendo los valores de **Pixel Size** por el tamaño que tenga cada tile en la hoja.

De esta manera se podrá acceder al identificador de cada tile, para poder referenciarlo en el descriptor de tiles. Debería haberse obtenido algo como lo que se indica la figura C.4, página 99.

Es posible crear objetos de un tamaño más grande accediendo al mismo parámetro *Sprite Editor* y modificando a mano el tamaño de los cuadrados que enmarcan cada tile. Habrá que hacer esto en caso de querer referenciar un objeto de mayor tamaño que una tile con un único identificador.

Preparando los prefabs de entidades

El diseño de las entidades, incluido el *player*, queda a la libre disposición del usuario. Sin embargo, hay una serie de limitaciones y recomendaciones que se exponen a continuación:

- Como mínimo debe crearse el *Prefab* del player.
- Cada *Prefab* debe tener sus componentes **RigidBody 2D**, con el parámetro *Body Type* en Kinematic, y un componente **BoxCollider** ajustado a su tamaño.
- Todas aquellas entidades que no sean el player debe asignárseles la capa *Enemies*, en el parámetro **Layer** del inspector, al lado del parámetro **tag**.
- Si hay un jefe, se recomienda añadirle la tag *Boss*, para identificarlo más facilmente al escribir el comportamiento.

- Se recomienda añadir un objeto hijo al *Prefab*, que contenga un componente **BoxCollider 2D** y simbolice el rango de ataque, como se ve en la figura C.5, página 99.

Las animaciones también quedan a la creatividad del usuario, para su posterior control personalizado.

Preparando los elementos UI

De nuevo, el diseño y confección de los elementos UI queda bajo control de cada usuario. Sin embargo, hay ciertas cuestiones que se deben cumplir:

- Es necesario crear una pantalla de muerte, una barra de salud, una pantalla de carga, una pantalla de pausa, y una pantalla de victoria.
- Cada una de ellas debe tener, como mínimo, su componente correspondiente de esta lista:
  - ***DeathScreen***
  - ***HealthBar***
  - ***LoadScreen***, donde se indicará una referencia a su elemento *Slider*.
  - ***PauseMenu***
  - ***OptionsMenu***, solo en caso de que haya un menú de opciones.
  - ***VictoryScreen***
- El menú principal se podrá crear completamente en la escena *UI* de la carpeta **Scenes** de la herramienta.
- Al crear el menú principal, es necesario enlazar el disparador del juego, ya sea un botón u otro elemento, con el método *loadLevel* del componente ***LoadScreen*** de la pantalla de carga. En el caso de ser un botón, se hará desde el apartado *onCLick()* de su componente ***Button***, como se ve en la figura C.6, página 99.

En la carpeta *scripts*, se podrán añadir aquellos controles extra que se desee asignar a los elementos UI.

Preparando los recursos de audio

Aquellos recursos de audio que se deseen utilizar, se colocarán en sus carpetas correspondientes, según sean efectos o temas, de la carpeta **SoundSource** del sistema de archivos.

C.2 Rellenar las plantillas de configuración

En la carpeta XML, se encuentra una subcarpeta llamada *sheets*. En ella se encuentran cinco plantillas que albergarán toda la información del juego.

ControlsConfSheet

En esta plantilla únicamente habrá que cubrir la etiqueta <*key*>. En ella se añadirán los identificadores de las teclas que se traducirán en las acciones que ya están reflejadas en la propia plantilla.

Es necesario que los identificadores sean los que utiliza la clase **KeyCode** de Unity. Consultar la documentación de **KeyCode** [59].

PrefabConfSheet

En esta plantilla se encontrarán los descriptores de entidades. Cada entidad se describirá en un elemento etiquetado como <*Prefab*> y no hay máximo de entidades que se pueden añadir.

En esta plantilla hay varios parámetros que deben respetar un formato, que se expone a continuación:

- Es obligatorio que el se identifique el parámetro <*type*> como *NPC*.
- El parámetro <*prefab*> debe ser el nombre exacto del objeto *Prefab* al que referencia la entidad.
- El parámetro <*behaviour*> debe ser el nombre exacto de la clase, no necesariamente el archivo, de la carpeta **Behaviours** que implementa el comportamiento de la entidad.

TileConfSheet

En este archivo se introducirán los descriptores e identificadores de tiles con las siguientes restricciones:

- El parámetro <*name*> debe ser el nombre exacto del identificador de la tile, ya creado y obtenido, de la forma que mostraba la figura C.4, página 99.
- El código que identificará la tile en el diseño, debe ser un único carácter ASCII, y no puede ser 0, ya que está resevado para la tile vacía.
- El parámetro <*solid*> será 0 si la tile no es sólida, y 1 si lo es.

RoomSheet

En este fichero se indicará el diseño y parámetros de una habitación.
Hay ciertas cuestiones que es importante respetar a la hora de diseñar una habitación:

- El parámetro *<gridUnits>* significa el número de unidades del espacio de contenido que la habitación ocupa. El tamaño de las unidades del espacio de contenido se definirá en el último archivo de configuración.
- El parámetro *<size>* debe indicar en su primer elemento, la base de la habitación, es decir, el número de tiles que ocupa de ancho, y su segundo elemento la altura de la habitación.
- El número de elementos del parámetro *<doors>* debe ser igual al número del parámetro *<gridUnits>*.
- Cada elemento del parámetro *<doors>* describe una unidad del diseño. Sus subelementos deben indicar los límites de la unidad de la siguiente manera:
  - El primer elemento corresponde al límite superior de la habitación.
  - El segundo elemento corresponde al límite inferior de la habitación.
  - El tercer elemento corresponde al límite izquierdo de la habitación.
  - El cuarto elemento corresponde al límite derecho de la habitación.
  - Si en cualquier límite hay una puerta, el valor será 1.
  - Si en cualquier límite hay un muro, el valor será 0.
  - Si cualquier límite está en el interior de una habitación de varias unidades, el valor será 2.
- Los elementos del parámetro *<doors>* se corresponden a las unidades como indica la figura C.7, página 100.
- El parámetro *<gameWorld>* contiene varios elementos, los cuales representan las capas de profundidad del diseño, siendo la primera la más profunda, y la última la menos profunda.
- Cada elemento de *<gameWorld>* debe tener la forma descrita en el parámetro *<size>*, siendo el ancho el número de caracteres que tendrá cada uno de los elementos etiquetados con *<string>*.
- **SOLO** si esa habitación será un final de nivel, se introducirá el parámetro *<levelEnd>*, junto a la posición del objeto que actuará de salto de nivel y la capa a la que pertenece.

- **SOLO** si la habitación tendrá generadores de entidades se añadirá el parámetro *<entities>*, que es una lista de elementos que representarán un generador. Hay que indicar de ellos su posición y las entidades que pueda generar.
- La habitación puede tener tema musical, y se reproducirá cada vez que el player pase por ella.

GameConfSheet

Esta es la plantilla más larga, representa el conjunto de configuraciones del juego.
Es necesario indicar los parámetros descritos a continuación:

- El player, que se entenderá como una entidad.
- El nombre del archivo de configuración de controles, que debe estar en la carpeta **xml**.
- El nombre del archivo de configuración de entidades.
- Los nombres de los *prefabs* de elementos UI.
- Las selecciones de habitaciones en el parámetro *<pools>*. Cada elemento *<PoolConf>* representa una selección, de la siguiente manera:
  - *<cellSize>* será el tamaño que ocupará una unidad del espacio de contenido. Debe ser la referencia para todas las habitaciones pertenecientes a la selección.
  - El nombre exacto de su hoja de sprites estará en el parámetro *<tileset>*
  - *<tileSize>* representa el tamaño de las tiles de la anterior hoja de sprites.
  - *<tileInfo>* contendrá el nombre del archivo descriptor de las tiles.
  - Se indicará el nombre del archivo de las habitaciones que sean principio y final de mazmorra, así como la distancia mínima entre ellas.
  - Se indicarán los nombres de archivo de todas las habitaciones que pertenezcan a la selección.
- Por último, de los niveles se indicará el tamaño del espacio de contenido donde se generará su mazmorra, la selección que utiliza y su tema musical.

Dentro de la carpeta XML se hayan ejemplos de uso de todas estas plantillas, con diferentes configuraciones para mostrar al jugador.

C.3 Construyendo los comportamientos de las entidades

Lo último que queda por hacer es elaborar los comportamientos de las entidades.

Para ello, se debe crear un script con una clase, que es la que se referencia en la plantilla de las entidades, que herede de *BehaviourConf*. Esto implementará una serie de métodos, que son los que se deben cubrir para crear el comportamiento:

- *move*: Define lo que sucede cuando la entidad se mueve.
- *idle*: Define lo que sucede cuando la entidad está inactiva.
- *npcBehave*: Define el comportamiento íntegro de un *NPC*.
- *takeDamage*: Define lo que sucede cuando la entidad recibe un ataque.
- *die*: Define lo que sucede cuando la entidad muere.
- *attack*: Define lo que sucede cuando la entidad ataca.

Realmente no hay ninguna restricción en el comportamiento, sin embargo, si hay una serie de recomendaciones que pueden resultar útiles:

- Se recomienda gestionar los efectos de audio desde aquí. Para ello se creará un objeto *AudioConf* con el clip a utilizar.
- Se recomienda gestionar todo lo relativo a animación de la entidad desde su comportamiento, ya que la herramienta no tiene una manera de controlarlo.
- El control de las colisiones debe realizarse desde el comportamiento.
- Para crear el comportamiento del player, no es necesario cubrir *npcBehave*.
- Se puede acceder a elementos ya existentes en la escena inicial, como el player o el director.

Dos ejemplos de implementación de comportamientos se pueden encontrar en la carpeta *Behaviours*. En ellos se encuentran aplicadas todas las recomendaciones anteriormente mencionadas, para que puedan servir de inspiración o punto de partida.

Para terminar, solo es necesario indicar al objeto director el nombre del archivo que contiene la configuración del mismo, y la herramienta ya estará lista para generar un juego.

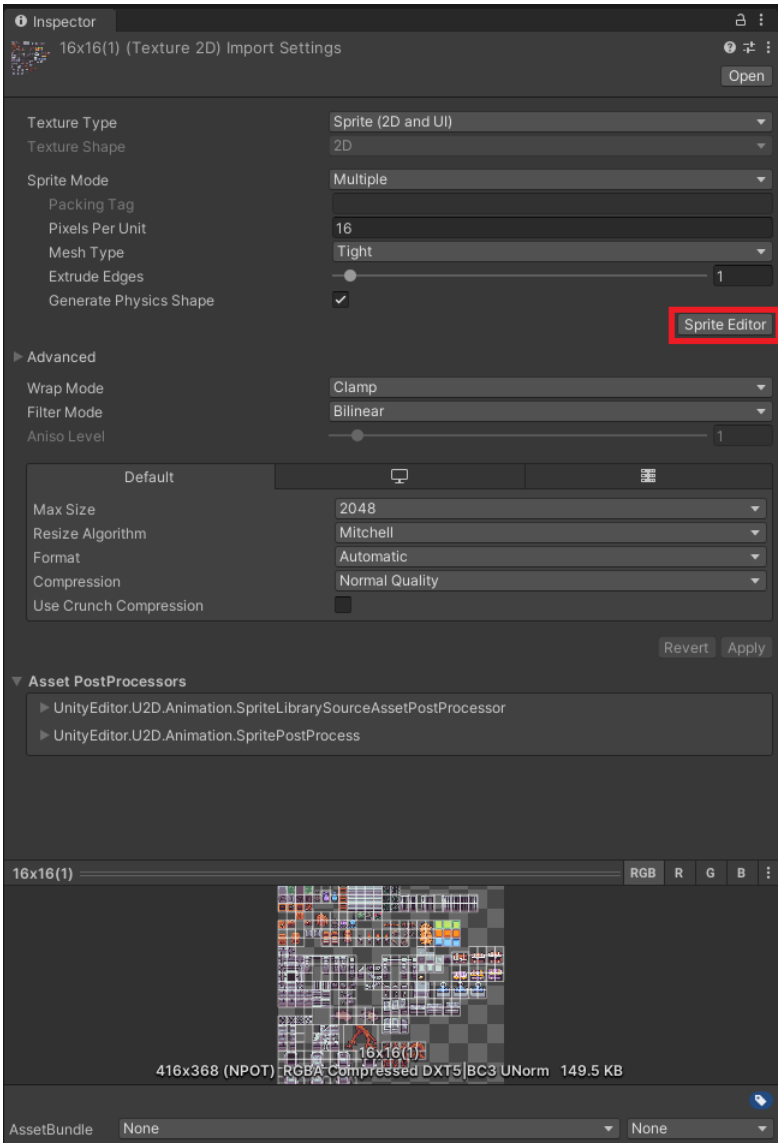


Figura C.2: Sprite Editor

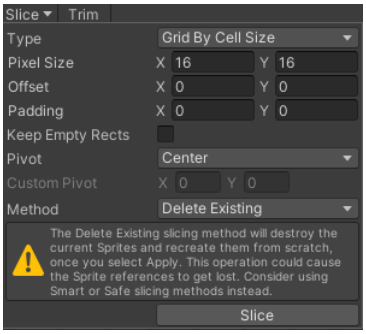


Figura C.3: Configuración Slice

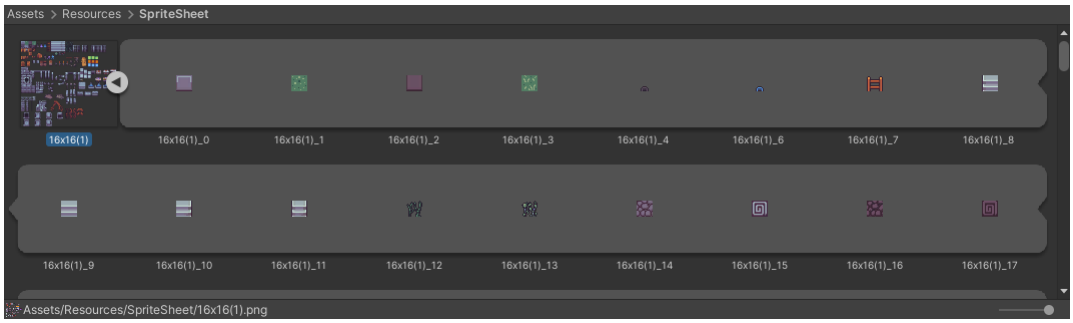


Figura C.4: Identificadores

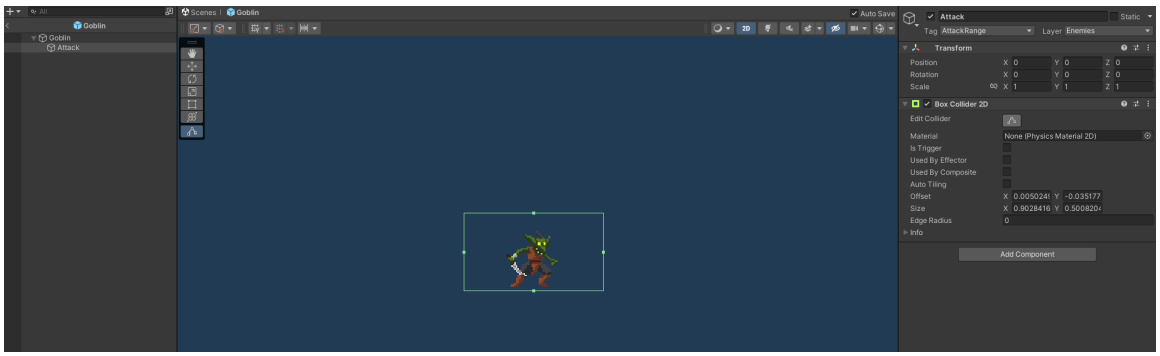


Figura C.5: Rango

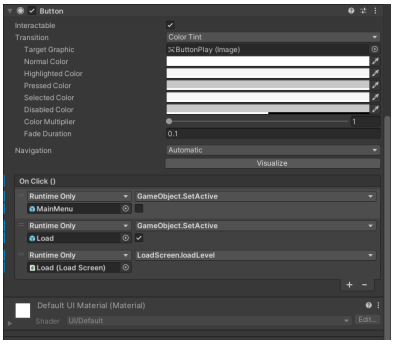


Figura C.6: Enlazado de un botón con el apartado onClick

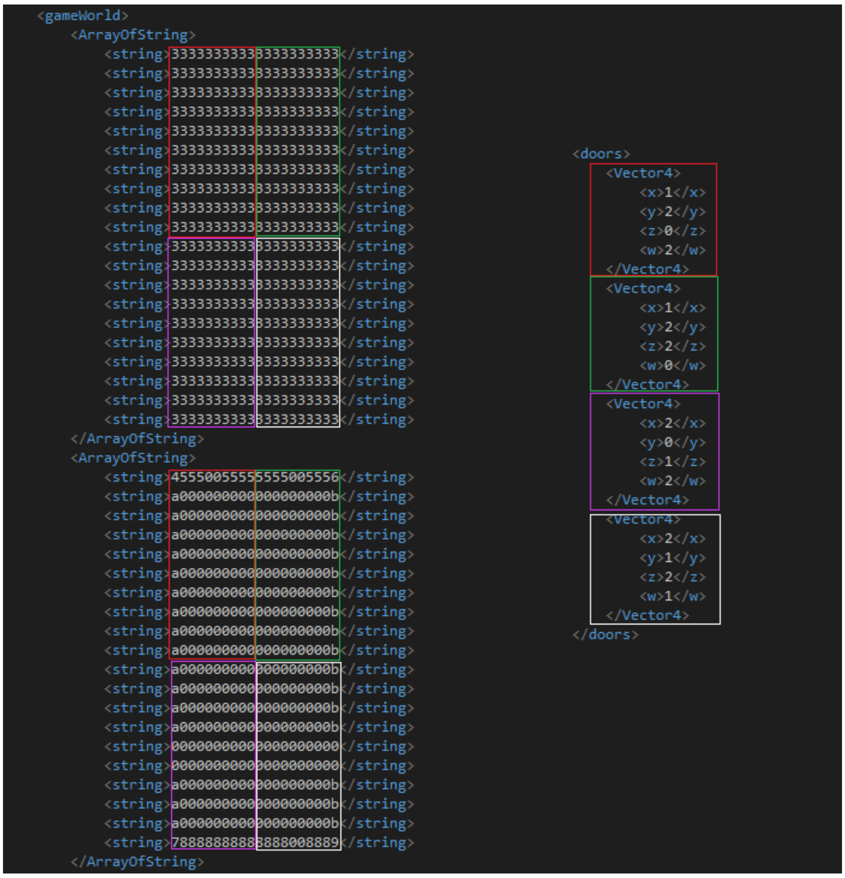


Figura C.7: Correspondencia **gameWorld** - **Doors**

Apéndice D

Diagramas y Figuras

En este anexo se presentan algunas figuras y diagramas del proyecto.

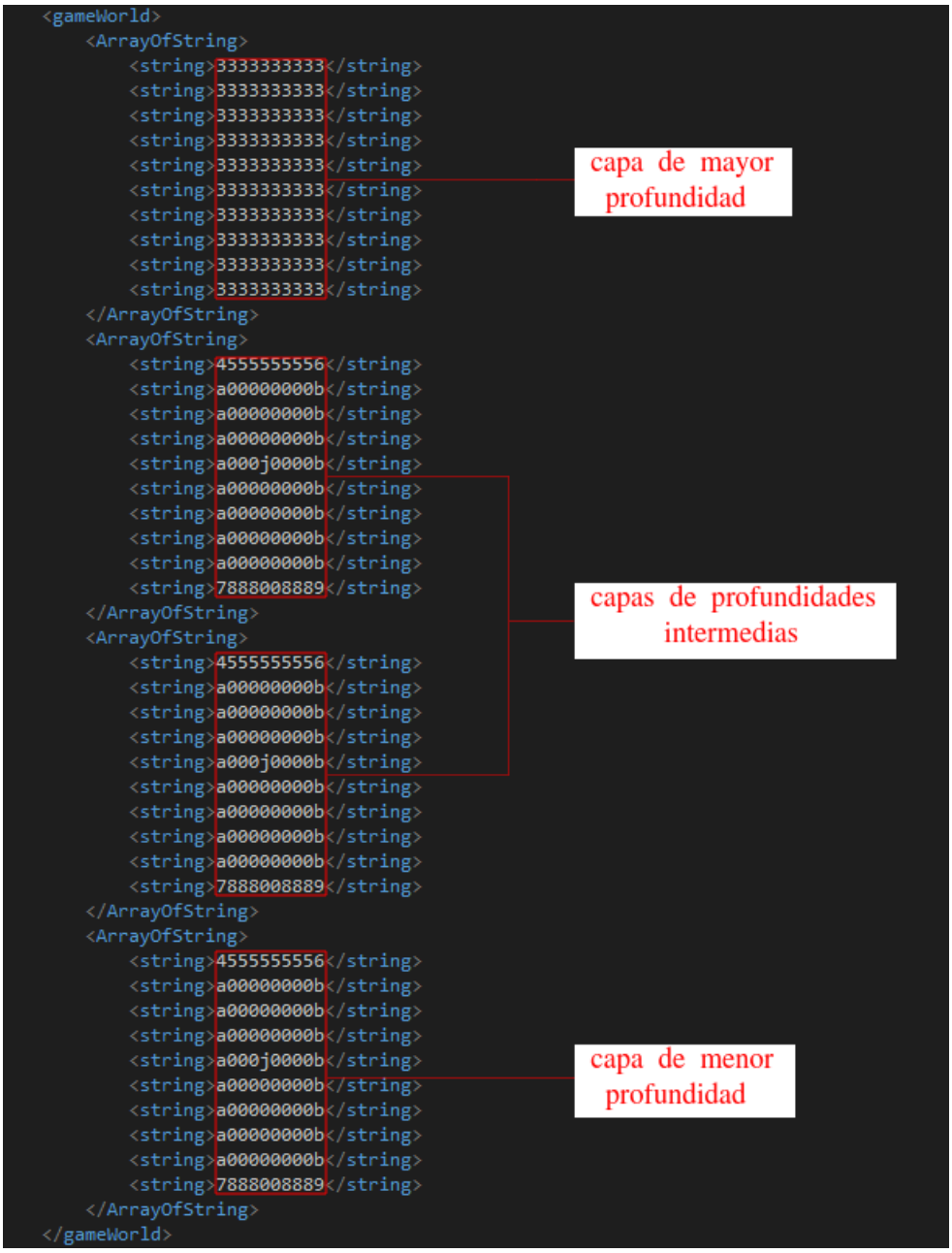


Figura D.1: Ejemplo de capas

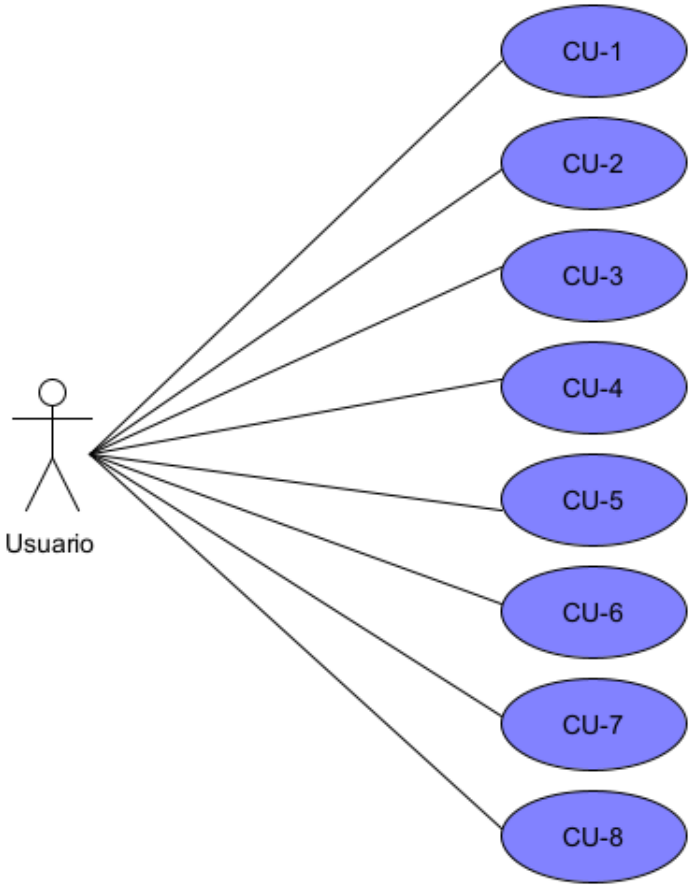


Figura D.2: Primera parte del diagrama general de casos de uso

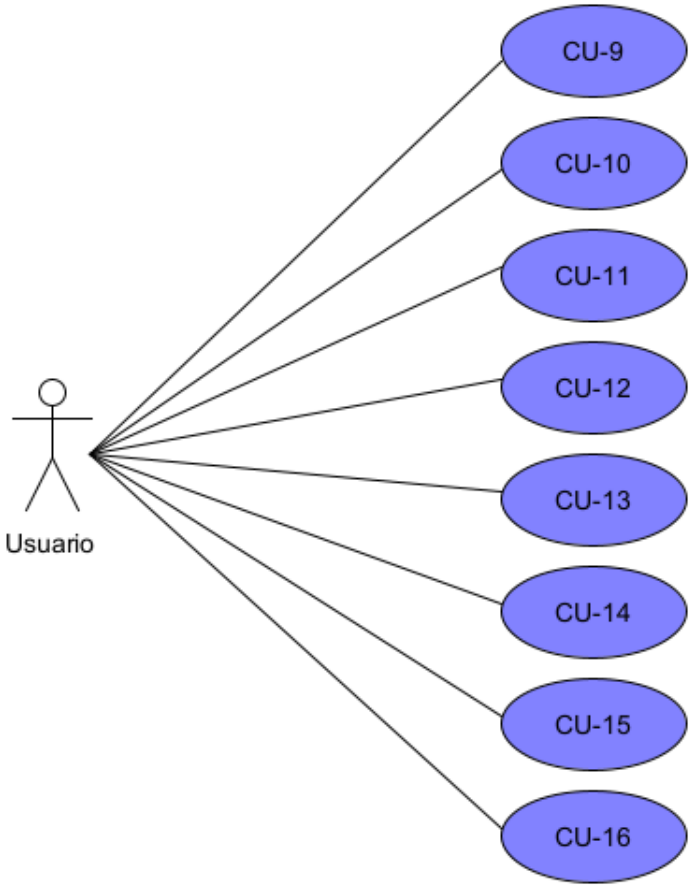


Figura D.3: Segunda parte del diagrama general de casos de uso

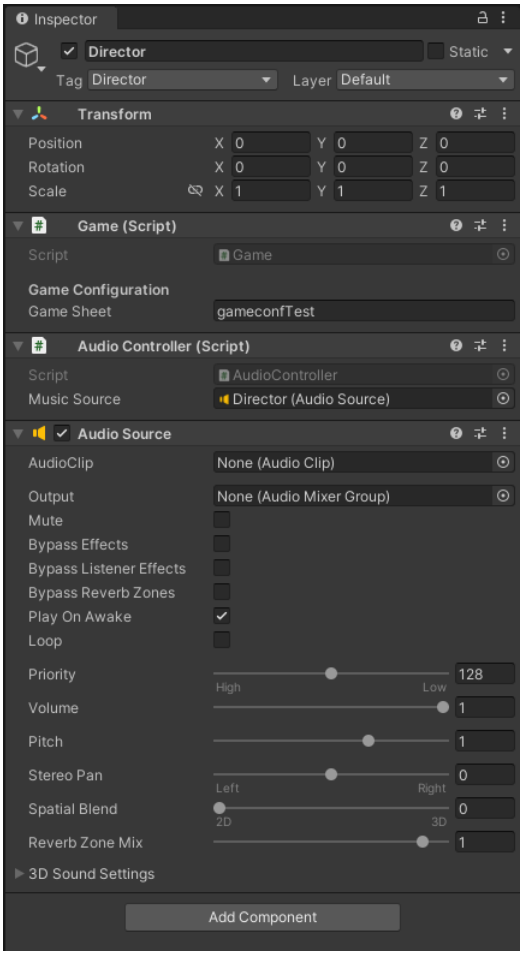


Figura D.4: Objeto Director

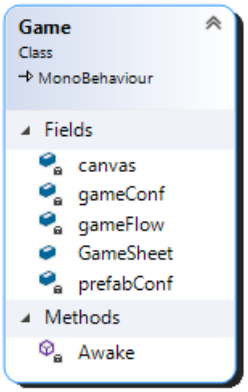


Figura D.5: Clase Game

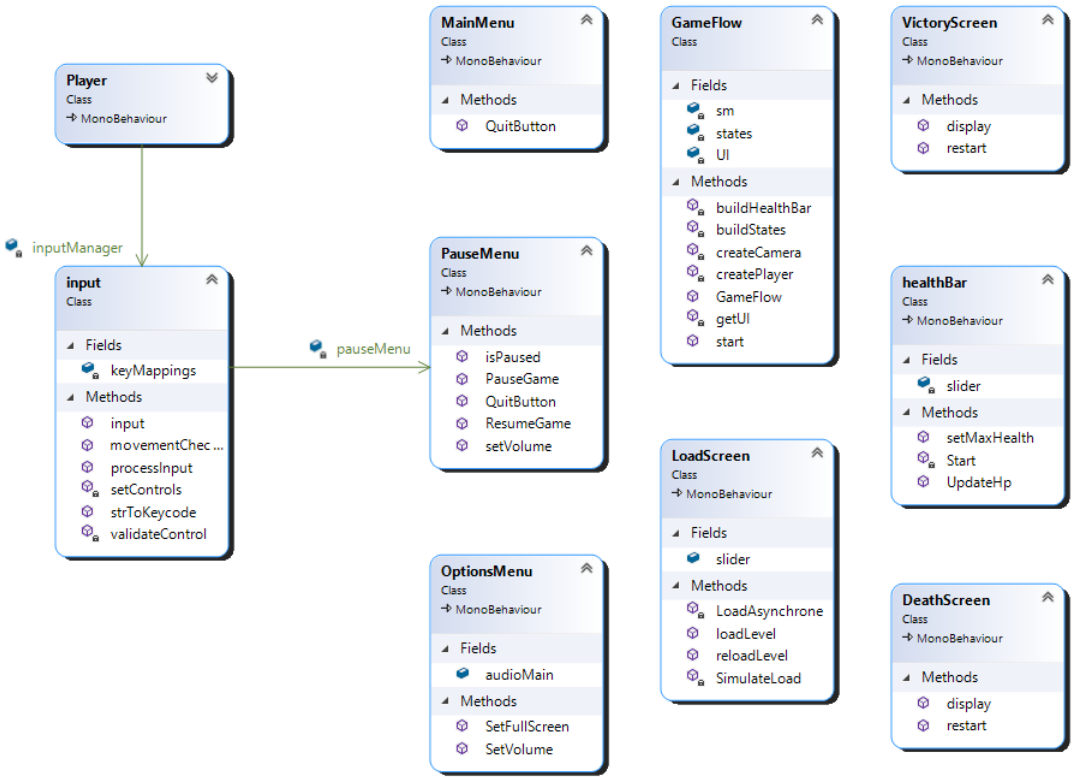


Figura D.6: Diagrama de clases para la iteración 9

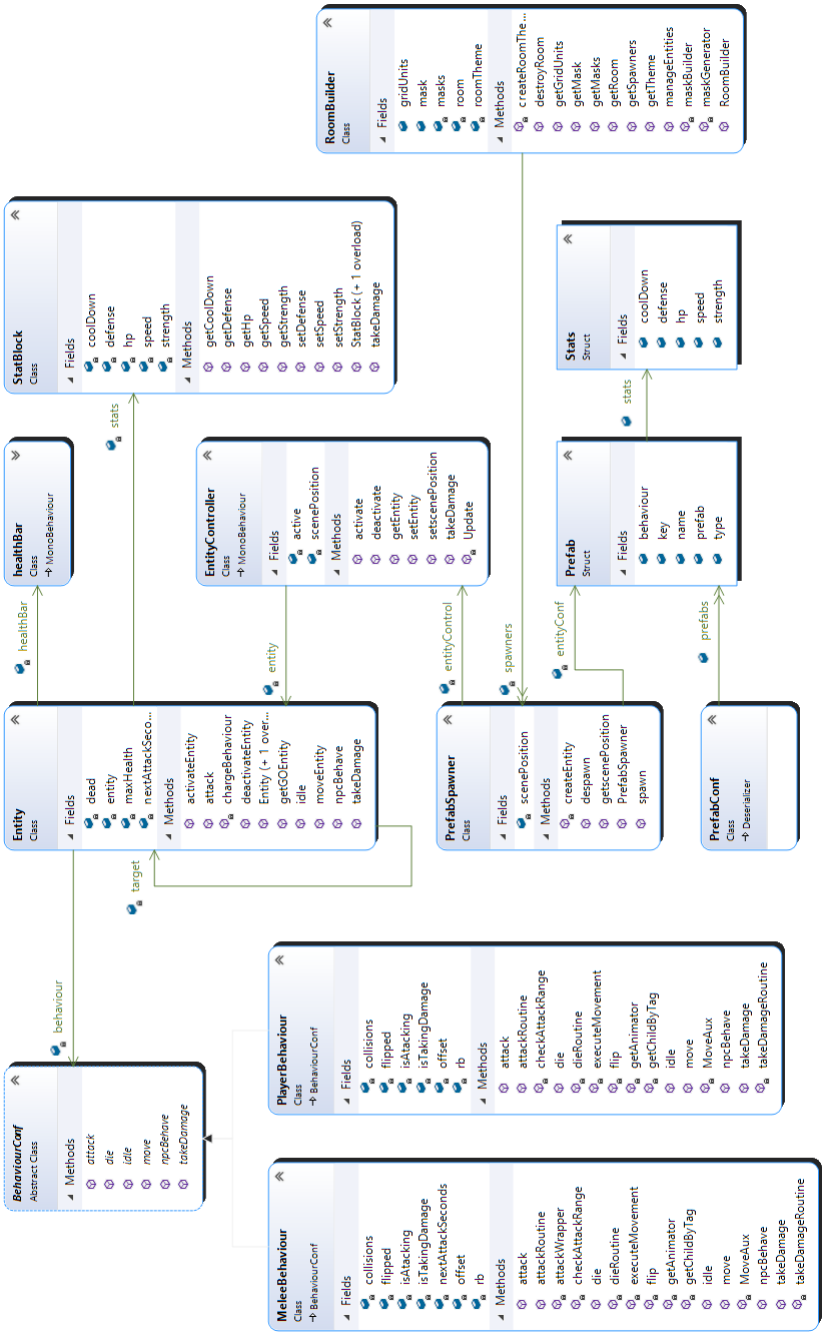


Figura D.7: Diagrama de clases para la Iteración 7

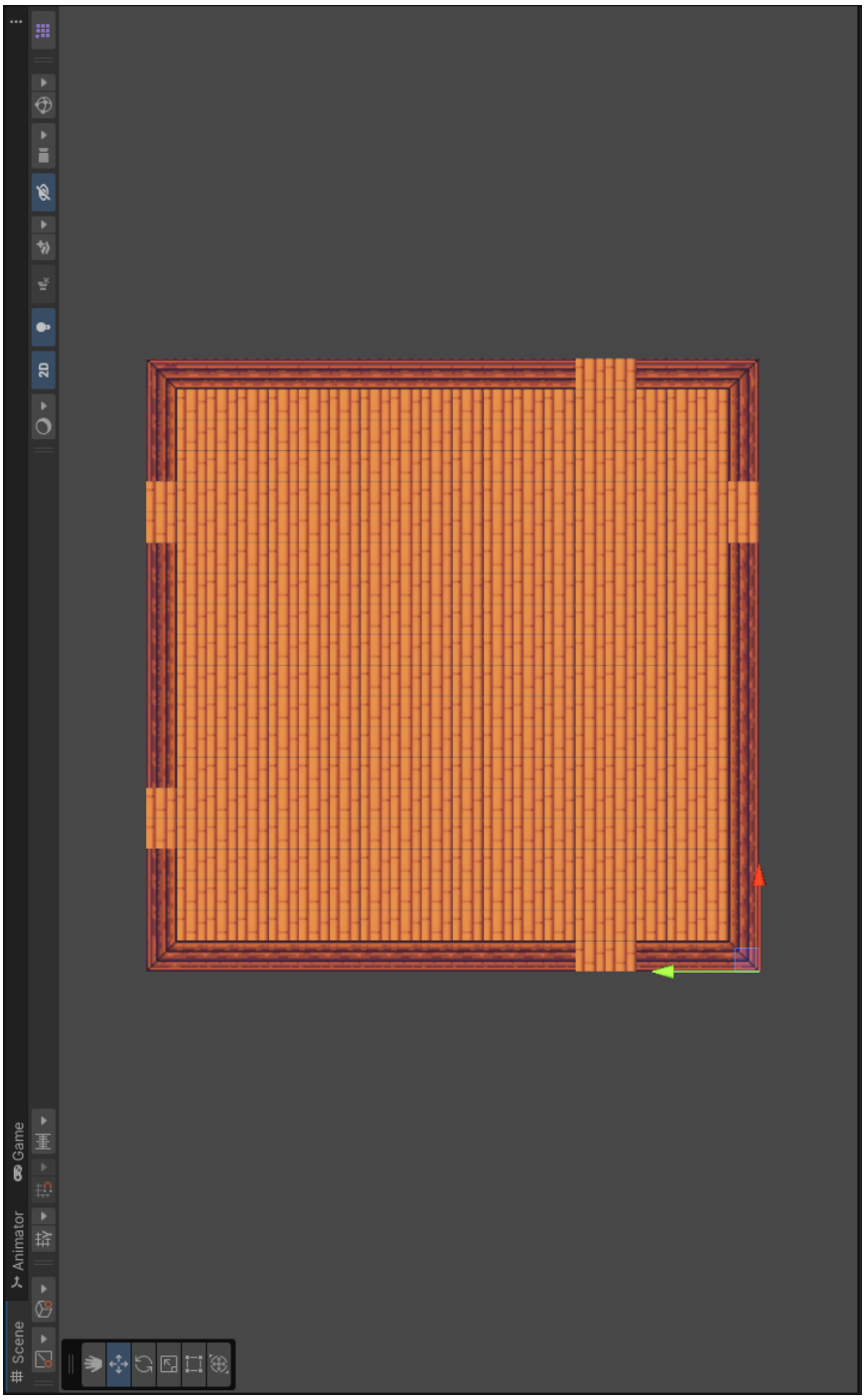


Figura D.8: Habitación de múltiples unidades de tamaño

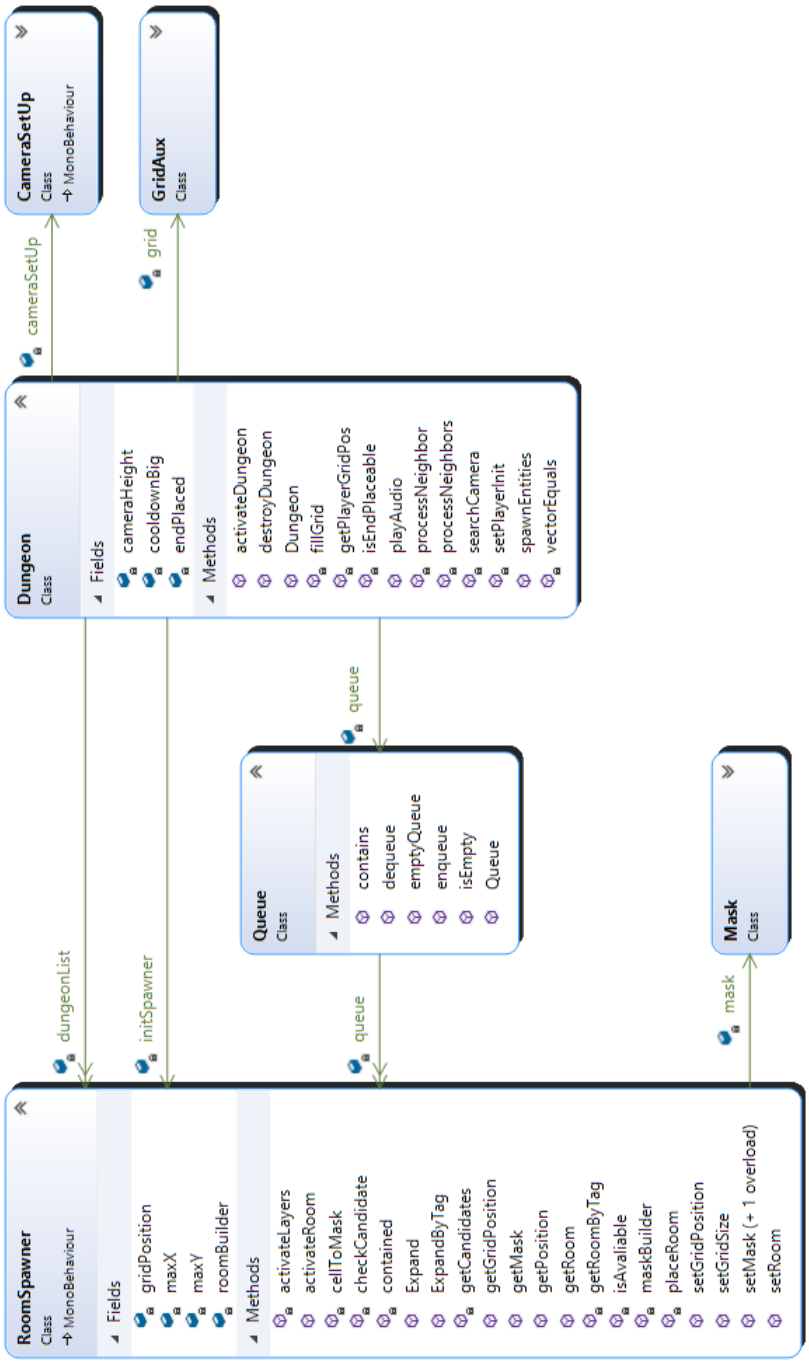


Figura D.9: Diagrama de clases para la iteración 6

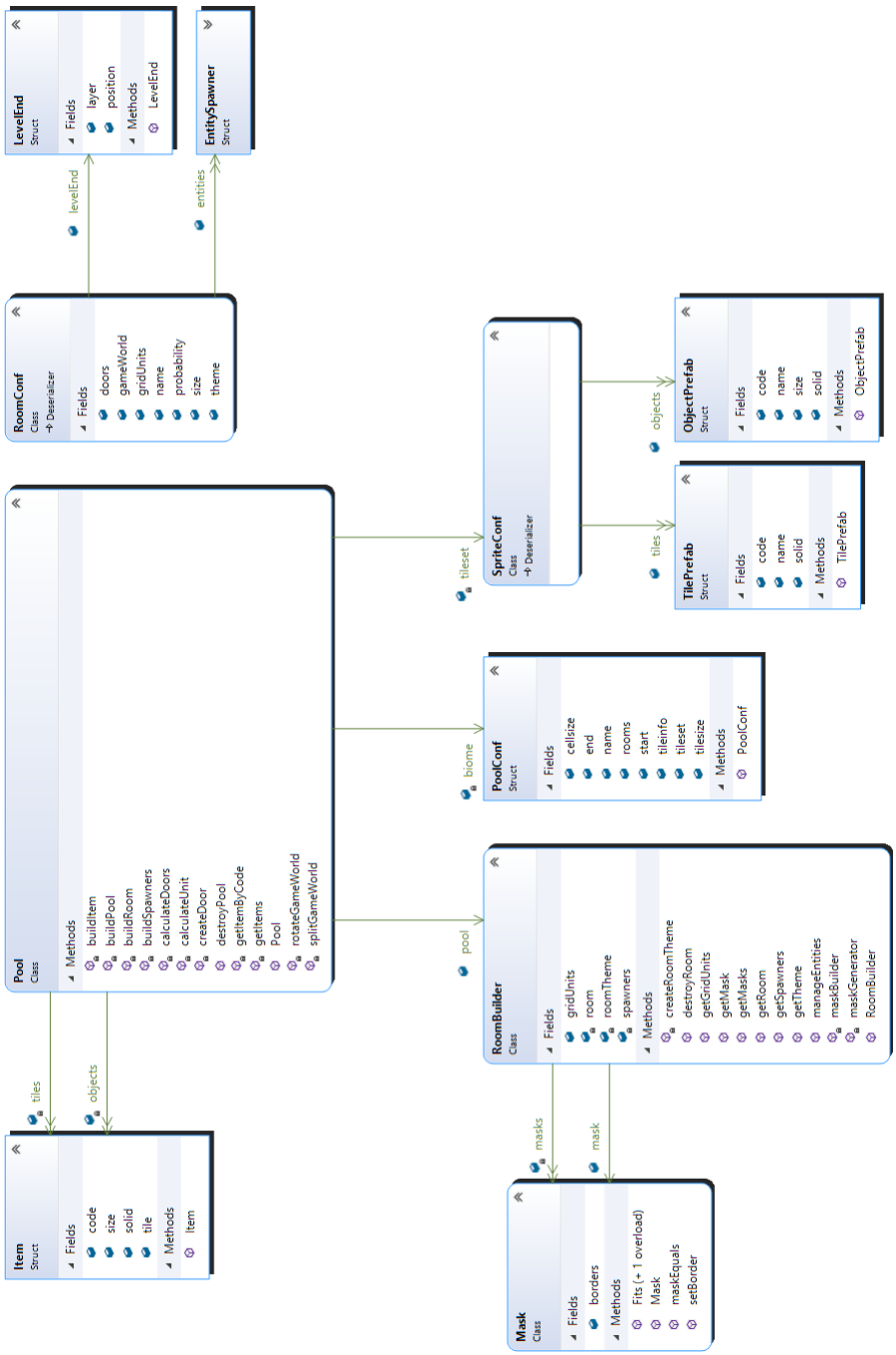


Figura D.10: Diagrama de clases de CU-8 y CU-9

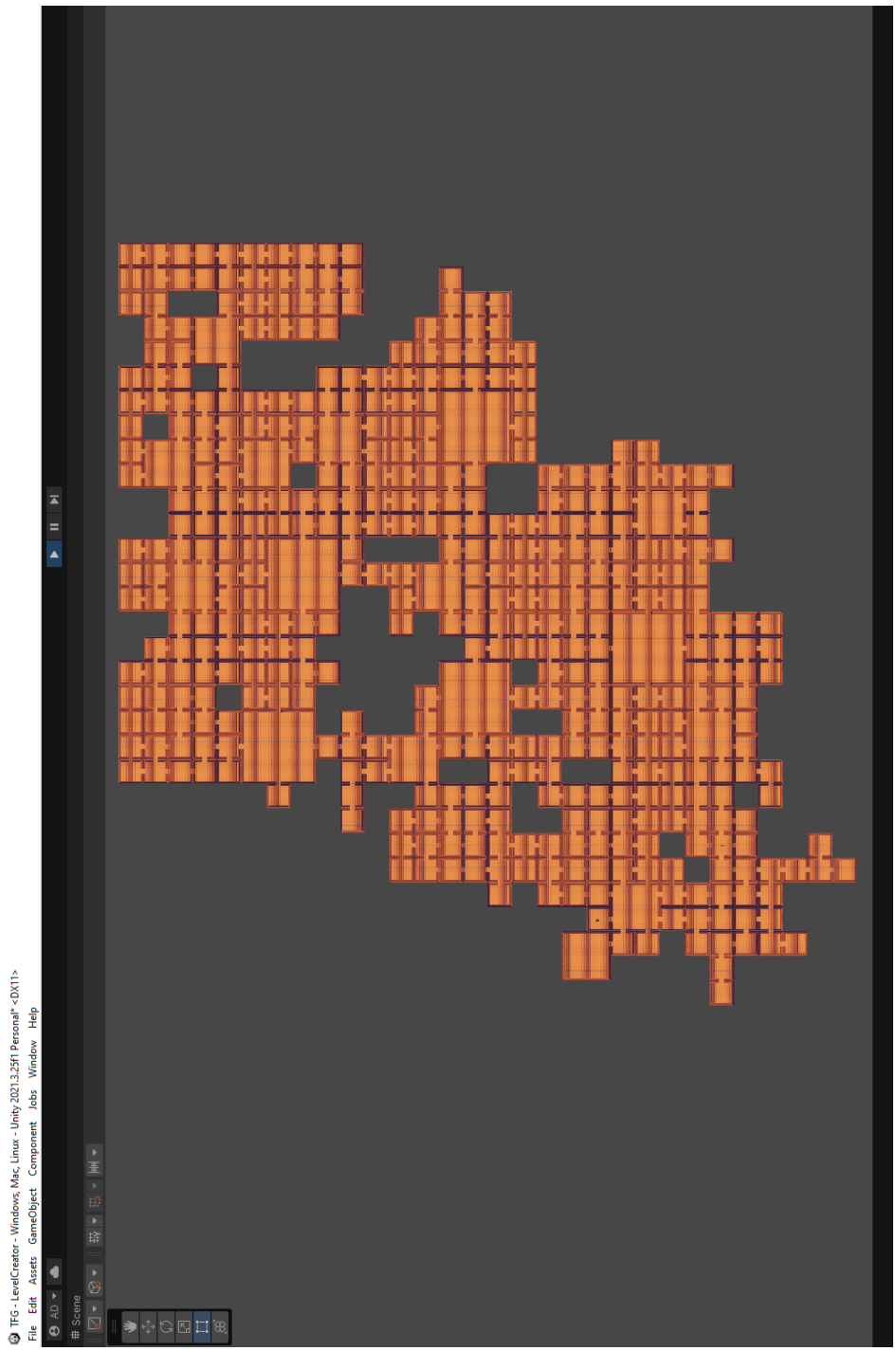


Figura D.11: Ejemplo de generación de un nivel

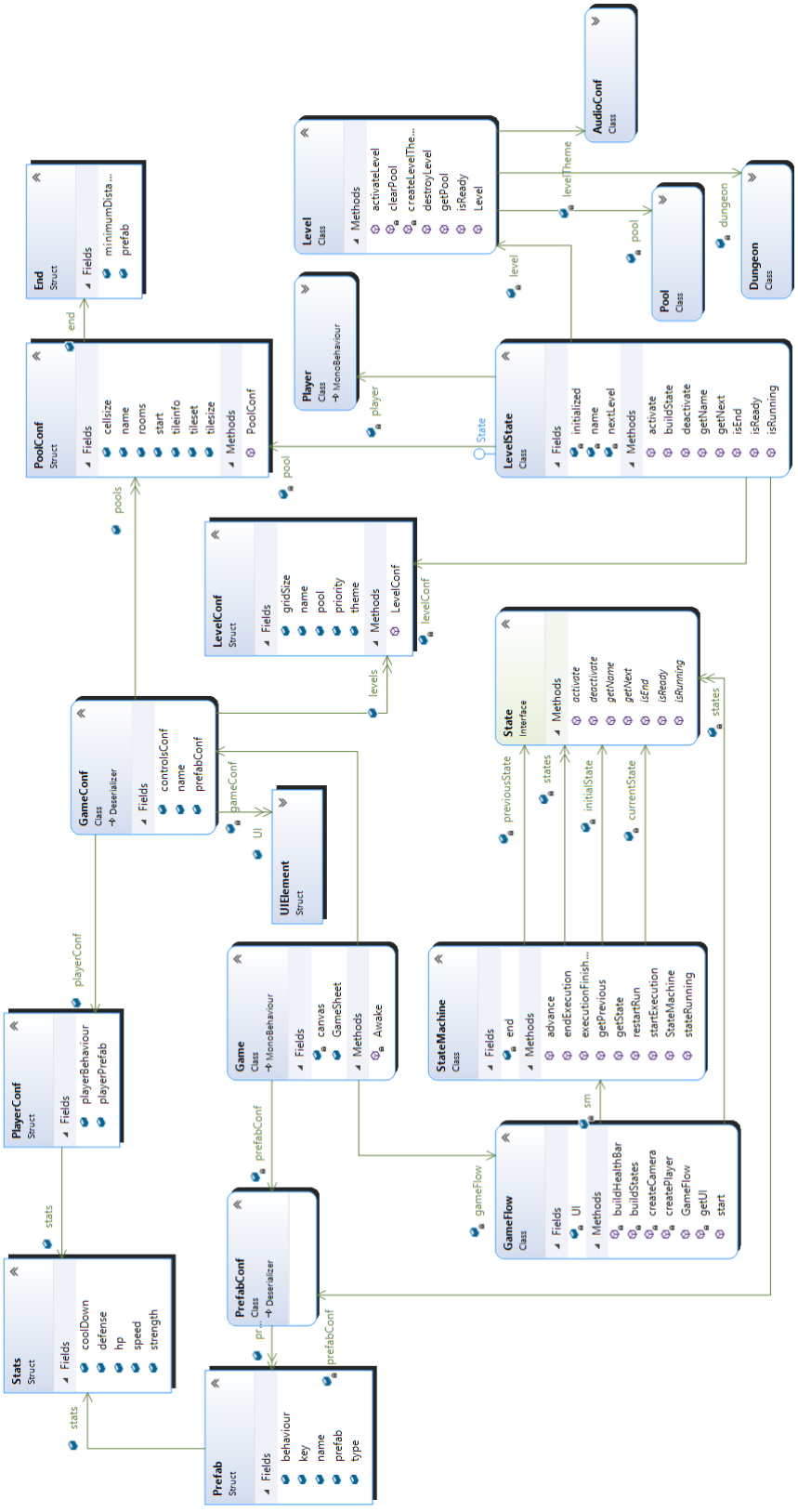


Figura D.12: Diagrama de clases para la iteración 8

Lista de acrónimos

2D Bidimensional. 1, 2, 11, 12, 14, 16

3D Tridimensional. 1, 6, 7, 11, 12, 14

IA Inteligencia Artificial. 6, 7

PP Programación procedimental. 1, 2, 4–9, 14

UI User Interface. 12, 21, 67, 70–73, 75, 79, 92, 93, 96

Glosario

2.5D Tipo de dimensionalidad que, dentro del ámbito de los videojuegos, ocurre cuando un videojuego se restringe a un plano bidimensional, pero se representa y simula en un entorno tridimensional. 12

Animator Componente que es utilizado para asignar una animación a un GameObject en su escena. 34, 60, 64

API De las siglas *Application Programming Interface*, sección de código que proporciona una serie de funcionalidades para que dos sistemas independientes puedan comunicarse, compartir información, o modificarse entre sí. 19

ASCII De las siglas *American Standard Code for Information Interchange*, código ligado al sistema de representación de caracteres del lenguaje natural en lenguaje de máquina. El ASCII es un código basado en el alfabeto latino que utiliza 7 bits para representar 128 caracteres diferentes. 9, 45, 94

Cola Tipo de dato abstracto, se trata de una colección de elementos con dos operaciones principales. La operación "encolar", que añadirá un elemento a la cola, al final de la colección, y la operación "desencolar", que devolverá el primer elemento de la colección y lo eliminará de la colección, dejando su lugar al siguiente elemento. 57

daily Scrum Reunión diaria breve, en la que los miembros de un proyecto, planificado con la metodología **Scrum**, llevan a cabo una sincronización sobre las tareas del proyecto. Normalmente se discuten las tareas realizadas desde la anterior reunión, y las tareas a llevar a cabo hasta la siguiente reunión. 17

dungeon Término que, dentro del ámbito de los videojuegos, se refiere a un nivel o zona cerrada, de estructura laberíntica y compuesto mayormente por pasillos y habitaciones. El término fue acuñado en este ámbito por el juego de rol **Dungeons & Dragons**. 8, 9

Frame Fotograma, define un marco de imagen que, en conjunto con otros y, reproducidos a cierta frecuencia, se genera la sensación de tener una imagen en movimiento. 33, 62, 63

fuzzy logic Tipo de lógica que trabaja con expresiones que tienen asignado un cierto porcentaje de veracidad. Estas expresiones no son al 100% ciertas o falsas, sino que oscilan en un intervalo entre esos mismos extremos. 7

grid Estructura de celdas, las cuales tienen el mismo tamaño y los límites de las mismas son paralelos entre sí, y con los límites de las demás celdas. Un sinónimo podría ser cuadrícula, o matriz, si el contexto es matemático. 12, 40–43, 45, 48, 53

IDE De las siglas *Integrated Development Environment*, aplicación informática que proporciona al usuario un entorno gráfico y una serie de servicios para facilitar el desarrollo de software. 19

layout Término que hace referencia a la disposición de los objetos contenidos en un espacio. Algunos sinónimos podrían ser, plano, maqueta o diseño. 9, 10, 12

miniMax Término que, dentro del ámbito de los videojuegos, se refiere al proceso de elección de cualquier estrategia dentro del juego cuyo objetivo sea minimizar la máxima pérdida posible del jugador. También se puede definir como el proceso de elección de cualquier estrategia que menos beneficie al jugador más aventajado del grupo, buscando un equilibrio entre todos los jugadores del grupo. 10

máquina de estados Se define como cualquier algoritmo o programa, donde a través de entradas y salidas se pueda avanzar en un grafo de estados, donde cada estado es un nodo, y cada conexión del grafo es la condición para poder avanzar entre los nodos que conecta. El algoritmo o programa debe recibir una serie de entradas y generar una serie de salidas. Estas salidas dependen tanto de las entradas recibidas, como de las salidas del anterior estado. 21, 67, 69, 79

NPC De las siglas “Non-Playable Character”, representa el concepto opuesto al jugador. Es una entidad, igual que el jugador, controlada por el sistema, que es quién define y aplica su comportamiento. Es el contrapunto al control por teclado del jugador principal. 60, 61

pseudoaleatorio Tipo de generación que utiliza un procedimiento determinista para producir resultados que asemejan aleatorios. Las salidas generadas pseudo-aleatoriamente no presentan ningún patrón, pero su proceso de generación es determinista por lo que,

introducir la misma entrada más de una vez genera el mismo resultado. 1, 2, 6, 8, 52–55, 57, 74, 75

ranking Lista o relación ordenada de cosas o personas en relación a un criterio determinado. 10

render Término inglés que se refiere a representar gráficamente uno o varios elementos sobre un entorno animado digitalmente. 7, 11, 50

rigging Proceso por el que se añaden puntos de control a un modelo 3D para poder animarlo o controlar sus movimientos de forma más precisa. Se puede comparar con crear una marioneta, donde cada punto de control del modelo 3D representa un punto en el cual la cuerda se une y controla al muñeco. 7

run Término que, dentro del ámbito de los videojuegos, se refiere a cualquier intento de completar un juego, independientemente de si tiene o no éxito. 9

sprite Es un elemento bidimensional, generalmente utilizado en formato de fotografía digital, PNG o JPG, el cual es un mapa de bits que forman una figura que se podrá instanciar dentro del juego. Si un conjunto de sprites se combinan, crean una animación. 2, 50

subrutina Conjunto de instrucciones que realizan una tarea. 4

tile Término que, dentro del ámbito de los videojuegos, se refiere a la unidad gráfica que se utiliza para construir el entorno en un videojuego 2D. El entorno, normalmente, se construye como un montaje, de tipo Collage, de tiles. Las tiles son todas del mismo tamaño, y un conjunto de ellas, las cuales compartan el tipo de entorno para el que están pensadas, colocadas en una misma imagen, conforman un TileSet. 20, 42, 44, 45, 49, 50, 80, 92, 94–96

unity Editor Hace referencia a la interfaz gráfica del motor Unity. El término ”Unity Editor” no solo se refiere a la pantalla principal de la aplicación, sino también a todas las pantallas emergentes de esta. 19

unity Inspector Sección de la interfaz gráfica de Unity que permite acceder y modificar un objeto de la escena, así como acceder a sus propiedades y atributos. 19

XML De las siglas *eXtensible Markup Language*, es un lenguaje estándar para el almacenaje e intercambio de información estructurada entre diferentes plataformas. 2, 20, 30, 33, 44, 45, 48, 49, 53, 61, 62, 64, 65, 67, 69, 70, 74, 79, 92, 94, 96

Bibliografía

[1] DEV, Asociación Española de Empresas Productoras y Desarrolladoras de Videojuegos y Software de Entretenimiento, *Libro blanco del desarrollo Español de videojuegos*, 9th ed. DEV - Asociación Española de Empresas Productoras y Desarrolladoras de Videojuegos y Software de Entretenimiento, 2022.

[2] ——, "Libro blanco del desarrollo español de videojuegos." [En línea]. Disponible en: https://www.dev.org.es/images/stories/docs/libro%20blanco%20del%20desarrollo%20espanol%20de%20videojuegos%202022.pdf

[3] A. Orús, "Industria mundial del videojuego - datos estadísticos." [En línea]. Disponible en: https://es.statista.com/temas/9150/industria-mundial-del-videojuego/#topicOverview

[4] I. Fernández, "Procedurally generated content." [En línea]. Disponible en: https://acortar.link/76mG4x

[5] ESAT Innovation School, "El arte procedural de no man's sky." [En línea]. Disponible en: https://www.esat.es/el-arte-procedural-de-no-mans-sky/

[6] Boris, "Dungeon generation in binding of isaac." [En línea]. Disponible en: https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/

[7] EBAC - Escuela Británica de Artes Creativas y tecnologías, "¿qué es unity y para qué sirve?" [En línea]. Disponible en: https://ebac.mx/blog/que-es-unity-y-para-que-sirve#:~:text=Unity%20es%20un%20motor%20gr%C3%A1fico,dispositivos%20sin%20cambiar%20de%20plataforma

[8] Unity Technologies, "Unity asset store." [En línea]. Disponible en: https://assetstore.unity.com/

[9] LMU - Loyola Marymount University, "Programming paradigms." [En línea]. Disponible en: https://cs.lmu.edu/~ray/notes/paradigms/

[10] CurioTek, "¿son los juegos procedimentales el futuro de la industria de los videojuegos?" [En línea]. Disponible en: https://curiotek.com/son-los-juegos-procedimentales-el-futuro-de-la-industria-de-los-videojuegos/#:~:text=%C2%BFQu%C3%A9%20es%20un%20juego%20Procedimental,la%20mayor%C3%ADa%20de%20los%20videojuegos

[11] C. Aguilla, "Generación procedural en los videojuegos: cuando las matemáticas facilitan el trabajo creativo." [En línea]. Disponible en: https://www.xataka.com.mx/videojuegos/generacion-procedural-videojuegos-cuando-matematicas-facilitan-trabajo-creativo

[12] Massive Software, "Página principal de massive software." [En línea]. Disponible en: https://www.massivesoftware.com/

[13] SideFX Software, "Página principal de sidefx, creador de houdini." [En línea]. Disponible en: https://www.sidefx.com/

[14] R. Cervantes, "¿qué es un roguelike? aquí tienes los 5 mejores ejemplos del género." [En línea]. Disponible en: https://www.3djuegosguias.com/otros-generos/que-roguelike-aqui-tienes-5-mejores-ejemplos-genero

[15] Michael C. Toy and Kenneth C. R. C. Arnold, "A guide to the dungeons of doom." [En línea]. Disponible en: https://docs-archive.freebsd.org/44doc/usd/30.rogue/paper.pdf

[16] TIME USA, LLC, Newspaper, "The 50 best video games of all time." [En línea]. Disponible en: https://time.com/4458554/best-video-games-all-time/

[17] Unity Technologies, "Página principal de unity." [En línea]. Disponible en: https://unity.com/es

[18] Nappin, "Roguelike generator pro: Procedural level and dungeon generator." [En línea]. Disponible en: https://assetstore.unity.com/packages/tools/level-design/roguelike-generator-pro-level-dungeon-procedural-generator-224345

[19] OndrejNepozitek, "Edgar pro - procedural level generator." [En línea]. Disponible en: https://assetstore.unity.com/packages/tools/utilities/edgar-pro-procedural-level-generator-212735

[20] J. Rodriguez, "Procedural level generator." [En línea]. Disponible en: https://assetstore.unity.com/packages/tools/ai/procedural-level-generator-136626

[21] O. Vezz, "Advanced dungeon generator - lite." [En línea]. Disponible en: https://assetstore.unity.com/packages/tools/level-design/advanced-dungeon-generator-lite-213225

[22] D. Miranda, "New dungeon pack modular low poly (free version)." [En línea]. Disponible en: https://assetstore.unity.com/packages/3d/environments/dungeons/new-dungeon-pack-modular-low-poly-free-version-174858

[23] asana, "Scrumban: lo mejor de dos metodologías ágiles." [En línea]. Disponible en: https://asana.com/es/resources/scrumban

[24] OpenWebinars, "7 consejos para hacer la daily scrum perfecta." [En línea]. Disponible en: https://openwebinars.net/blog/consejos-para-hacer-daily-scrum/

[25] deloitte, "Claves para ser un buen scrum master." [En línea]. Disponible en: https://www2.deloitte.com/es/es/pages/technology/articles/claves-para-ser-buen-scrum-master.html

[26] Unity Technologies, "Unity." [En línea]. Disponible en: https://unity.com/es

[27] Microsoft, "Visual studio." [En línea]. Disponible en: https://visualstudio.microsoft.com/es/

[28] Talent.com, "Salario medio para jefe de proyecto en españa, 2023." [En línea]. Disponible en: https://es.talent.com/salary?job=jefe+de+proyecto#:~:text=Descubre%20cu%C3%A1l%20es%20el%20salario%20medio%20para%20Jefe%20De%20Proyecto&text=%C2%BFCu%C3%A1nto%20gana%20un%20Jefe%20de%20proyecto%20en%20Espa%C3%B1a%3F&text=El%20salario%20jefe%20de%20proyecto,%E2%82%AC%2019%2C23%20por%20hora.

[29] ——, "Salario medio para analista programador en españa, 2023." [En línea]. Disponible en: https://es.talent.com/salary?job=analista+programador#:~:text=El%20salario%20analista%20programador%20promedio,hasta%20%E2%82%AC%2040.000%20al%20a%C3%B1o.

[30] ——, "Salario medio para programador junior en españa, 2023." [En línea]. Disponible en: https://es.talent.com/salary?job=programador+junior#:~:text=El%20salario%20programador%20junior%20promedio,hasta%20%E2%82%AC%2027.000%20al%20a%C3%B1o.

[31] Profile, "Solid: los 5 principios que te ayudarán a desarrollar software de calidad." [En línea]. Disponible en: https://profile.es/blog/principios-solid-desarrollo-software-calidad/

[32] ——, "Principio de responsabilidad Única." [En línea]. Disponible en: https://profile.es/blog/principios-solid-desarrollo-software-calidad/#1_Principio_de_Responsabilidad_Unica

[33] ——, "Principio de abierto/cerrado." [En línea]. Disponible en: https://profile.es/blog/principios-solid-desarrollo-software-calidad/#2_Principio_de_AbiertoCerrado

[34] Unity Technologies, "Documentación de resources." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Resources.html

[35] ——, "Api de audiosource." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/AudioSource.html

[36] ——, "Documentación de monobehaviour." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[37] Microsoft, "Xmlserializer clase." [En línea]. Disponible en: https://learn.microsoft.com/es-es/dotnet/api/system.xml.serialization.xmlserializer?view=net-7.0

[38] Unity Technologies, "Documentación de animator." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Animator.html

[39] ——, "Documentación de la clase input." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Input.html

[40] Usuario de Github @ftvs, "Simple camera shake effect for unity3d." [En línea]. Disponible en: https://gist.github.com/ftvs/5822103

[41] windexglow en Unity Forum, "Fly cam (simple cam script)." [En línea]. Disponible en: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.6/manual/index.html

[42] Unity Technologies, "Documentación de cinemachine." [En línea]. Disponible en: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.6/manual/index.html

[43] Isabelle Riva en Unity Blog, "Unity wins its first technology and engineering emmy® award." [En línea]. Disponible en: https://blog.unity.com/technology/unity-wins-its-first-technology-and-engineering-emmy-award

[44] Raidenwins en Unity Forum, "Cinemachine vs. traditional third person camera controller." [En línea]. Disponible en: https://forum.unity.com/threads/cinemachine-vs-traditional-third-person-camera-controller.838165/

[45] ONE WHEEL STUDIO, Dev Log, "Cinemachine. if you're not. you should." [En línea]. Disponible en: https://onewheelstudio.com/blog/2021/9/11/cinemachine

[46] Unity Technologies, "Documentación de transform." [En línea]. Disponible en: https://docs.unity3d.com/es/2018.4/Manual/Transforms.html

[47] ——, "Documentación de debug." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Debug.html

[48] ——, "Documentación de grid." [En línea]. Disponible en: https://docs.unity3d.com/Manual/class-Grid.html

[49] ——, "Documentación de tilemap." [En línea]. Disponible en: https://docs.unity3d.com/es/2017.4/Manual/Tilemap.html

[50] ——, "Documentación de rigidbody." [En línea]. Disponible en: https://docs.unity3d.com/es/2019.4/Manual/class-Rigidbody.html

[51] ——, "Documentación de collider 2d." [En línea]. Disponible en: https://docs.unity3d.com/Manual/Collider2D.html

[52] ——, "Documentación de random.range." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/Random.Range.html

[53] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. The MIT Press, 2004.

[54] Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

[55] Unity Technologies, "Documentación de canvas." [En línea]. Disponible en: https://docs.unity3d.com/es/2019.4/Manual/UICanvas.html

[56] Anton Puzanov, "Male 4." [En línea]. Disponible en: https://assetstore.unity.com/packages/3d/characters/humanoids/humans/male-4-266387

[57] Tropical Studio, "Huge playground post apocalyptic." [En línea]. Disponible en: https://assetstore.unity.com/packages/3d/environments/huge-playground-post-apocalyptic-266663

[58] HubSpot, "Qué es el modelo freemium (y 7 empresas que lo utilizan con éxito)." [En línea]. Disponible en: https://blog.hubspot.es/sales/que-es-freemium#:~:text=El%20modelo%20freemium%20es%20una,pago%20para%20funcionalidades%20m%C3%A1s%20avanzadas.

[59] Unity Technologies, "Documentación keycode." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/KeyCode.html