Trabajo de Fin de Grado

Desarrollo de un videojuego 2D en Unity

Autor: ***AXEL RAMADA GONZÁLEZ***

Tutor: Rubén Escobedo Gutiérrez

Ciclo Formativo de Grado Superior:

Desarrollo de Aplicaciones Multiplataforma

Departamento de Informática

AÑO ACADÉMICO: 2022/2023

Índice

Resumen

Este proyecto intenta plasmar la forma de crear un videojuego 2D en Unity. Enseñando y explicando sus partes más elementales y explicando, además de ejemplificar con el trabajo realizado como en el caso de las animaciones, mapeado, y la creación de los scripts. Finalmente, se aprovechará para plasmar la dirección y organización de un proyecto complejo y creativo.

Introducción

A continuación explicaremos los antecedentes, la motivación y los objetivos principales de este proyecto.

Antecedentes y motivación

Se eligió la realización de este proyecto como Trabajo de Fin de Grado debido a que la creación de un videojuego y sus correspondientes fases son una excelente forma de realizar un proyecto de desarrollo que contenga tanto la parte más cercana al cliente (en este caso el jugador) como la más cercana al propio desarrollador. Analizando y creando los recursos que como jugador se van a usar (como interfaces gráficas, uso de físicas del propio juego, objetos del mismo, entre otros), así como teniendo que seguir patrones de diseño para un desarrollo óptimo, implementación de herencia, además de necesitar conocimientos profundos en programación.

Los videojuegos son una excelente forma visual de plasmar los conocimientos adquiridos a la hora de programar, ya que podemos ver de forma muy visual lo útil que puede llegar a ser la programación orientada a objetos y entender conceptos como la herencia o polimorfismo.

Objetivos

Tal y como hemos mencionado, el objetivo principal del proyecto será crear desde cero un videojuego de tipo TOP-DOWN [1] en el motor de desarrollo de videojuegos de Unity [2] implementando scripts en C#. El juego estará formado por múltiples escenas, un personaje que será controlado en movimiento y acciones, como atacar e interactuar con diversos elementos como pociones, trampas, enemigos y NPC. También deberán existir elementos de UI (*User Interface*), como un menú de inicio, un menú de configuración, sistema de guardado, efectos de sonidos, entre otros.

Todo el proyecto será desarrollado bajo la metodología Kanban [3], la cual ayudará a la organización y planificación de las tareas a realizar durante las horas establecidas para realizar el proyecto.

"El hombre sólo juega cuando es libre en el pleno sentido de la palabra, y sólo es plenamente hombre cuando juega."

- Friedrich Schiller.

Planificación

La planificación del proyecto será la siguiente: se invertirán tres horas al día, tres días a la semana, durante tres meses, siendo un total de 108 horas invertidas en el proyecto. Durante los dos primeros meses se elaborará el desarrollo del juego y, el medio mes restante, el desarrollo de la documentación.

Requisitos del proyecto

Durante este apartado se expondrán los distintos requisitos del proyecto divididos entre funcionales (funcionalidad del sistema), no funcionales (propiedades del sistema) y opcionales (aquellos que serán realizados solo en caso de que diere tiempo).

Requisitos funcionales:

- Movilidad del jugador, el jugador debe poder moverse en las direcciones arriba, abajo, izquierda y derecha.
- El jugador puede recoger objetos, el jugador debe ser capaz de recoger objetos que estén situados en la escena de juego.
- El jugador recibirá y aplicará daños a enemigos, el jugador debe ser capaz de reducir la vida de los enemigos mediante ataques y viceversa.
- El jugador dispondrá de un menú de configuración y otro de pausa, el menú de configuración debe estar disponible en el menú de inicio y debe poderse configurar alguna característica del juego, el menú de pausa estará disponible en la escena de jeugo y debe ser capaz de pausar y reanudar el juego.
- Los enemigos deben perseguir al jugador, los enemigos deben ser capaces de percibir y de perseguir al jugador, a su vez si el jugador se aleja un número determinado de casillas, el enemigo debe cesar su persecución.
- Un correcto sistema de diálogo, el jugador debe poder interactuar con un NPC, al hacerlo debe emerger un texto del cual se debe poder salir y volver a entrar a placer.
- Un sistema que contabilice las monedas que se van obteniendo, un contador fijo en la esquina superior izquierda de la pantalla en la que se puedan ver las monedas que llevamos recogidas.
- Un correcto sistema de guardado y cargado, un guardado persistente, que sea accesible al jugador y un cargado de juego mediante una tecla del teclado.

Requisitos no funcionales:

- El juego debe ser compatible en Windows 10.
- El juego debe tener unos recursos gráficos de bajos requisitos, por bajos requisitos se entiende que el juego debe ser accesible para ordenadores que tengan 4 GB de RAM, un procesador I3 de Intel, y una gráfica inferior a la Nvidia 1050 o disponer de una integrada.

Requisitos opcionales:

- Música dentro del juego.
- Animaciones de escenario, animaciones de luz, brillo en objetos, y de los diferentes elementos del escenario como pudieron ser las trampas.
- Sistema de mercado e inventario, poder intercambiar las monedas obtenidas por objetos dentro del juego y tener un inventario para poder almacenar las pociones.

Estructura de trabajo utilizada

Para describir las partes del proyecto, se plasmará en la Estructura de Descomposición del Trabajo (EDT a partir de ahora) se plasma en la Figura 1. Esta herramienta es utilizada en la gestión de proyectos para descomponer éstos en partes más pequeñas y manejables. La EDT muestra de manera jerárquica todas las tareas y elementos del proyecto, desde las actividades más generales hasta las más específicas.

Figura 1. EDT del proyecto.

Alcance del proyecto en horas

Como ya se dijo anteriormente se invertirán un total de 108 horas en el proyecto. Durante los dos primeros meses se elaborará el desarrollo del juego y, el medio mes restante, el desarrollo de la documentación.

Metodología a utilizar

La metodología Kanban representada en la Figura 2, es una representación visual del flujo de trabajo, generalmente dividido en columnas que muestran los estados del trabajo, como "Por hacer", "En progreso" y "Hecho". Cada elemento de trabajo se representa mediante tarjetas que se mueven a través de las columnas a medida que progresan. Es un enfoque ágil para la gestión de proyectos y el trabajo.

Figura 2. Ejemplo de la metodología Kanban.

Análisis

A continuación se analizará algunos aspectos a tener en cuenta a la hora de elaborar el proyecto, cuestiones tales como pueden ser la inspiración, las tecnologías que vamos a utilizar y las dificultades que esperamos tener a lo largo del proyecto y con qué elementos las tendremos.

Inspiración

"The Legend of Zelda" [4]: Este juego de aventuras de Nintendo para la consola NES, mostrado en la Figura 3, introdujo el estilo top-down(vista elevada) en un mundo abierto y sentó las bases del género RPG (Role Playing Game) en la industria de los videojuegos además de ser uno de los pocos juegos portátiles de la época que permiten guardar la partida.

Figura 3. The Legend of Zelda.

"The Binding of Isaac" [5]: Un juego de acción RPG con fuertes elementos de tipo Roguelike(género de videojuego) queda reflejado en la Figura 4, en el que los niveles son generados aleatoriamente. En el transcurso del viaje de un niño llamado Isaac, los jugadores encontrarán extraños tesoros que cambiarán la forma de Isaac, le darán habilidades y poderes sobrehumanos que le permitirán luchar contra hordas de criaturas aterradoras y diabólicas, descubrir secretos y abrirse camino a su supervivencia.

Figura 4. The Binding of Isaac.

Tecnologías a utilizar

El trabajo mostrado a continuación se ha realizado usando el motor de desarrollo de videojuegos de Unity, que a su vez utiliza Visual Studio, todo el código escrito en este proyecto está en el lenguaje de programación C#.

Como ya se ha mencionado, en este proyecto se ha utilizado exclusivamente el lenguaje de programación multiparadigma desarrollado y estandarizado por la empresa Microsoft como parte de su plataforma .NET. Para más información sobre C#.

Dificultades previstas

El proyecto se comenzará sin conocimientos previos sobre la creación de un videojuego, ya que en parte es un proyecto de investigación. Es obvio que una de las dificultades previstas es el desconocimiento tanto en la creación propia del proyecto, como en el uso de Unity como entorno de trabajo, ya que es un entorno nuevo en el que nunca antes se ha desarrollado otro proyecto..

La parte visual del proyecto (que es una de las más importantes: *sprites*, animaciones y diseño de niveles) supondrá una dificultad alta, de nuevo, por los conocimientos nulos sobre este aspecto.

También lo será el uso y conocimiento de la clase `MonoBehaviour`, entre otras, y sus diferentes métodos a la hora de programar en C#. El hecho de que no se deben duplicar elementos claves será indispensable para un correcto funcionamiento del juego. Para estos elementos se usará el patrón de diseño Singleton[1].

[1] El patrón de diseño Singleton es un patrón de diseño creacional que garantiza que solo exista una instancia de una clase y proporciona un punto de acceso global a ella.

Diseño

En este apartado se diseñarán los diferentes diagramas y diseños conceptuales que serán implementados posteriormente, con el propósito de mostrar de una forma visual cuáles son los elementos que van a componer el proyecto, sus clases y funcionalidades, permitiendo comprender de manera simple y visual la complejidad del proyecto.

Diagrama de casos de uso

Un Diagrama de Casos de Uso [6] es una representación gráfica que describe la interacción entre los actores (usuarios o sistemas externos) y el sistema en desarrollo, tal y como se plasma en la Figura 5. Proporciona una vista de alto nivel de las funciones y características del sistema desde la perspectiva de los actores involucrados. Es una herramienta útil para capturar y comunicar los requisitos funcionales del sistema.

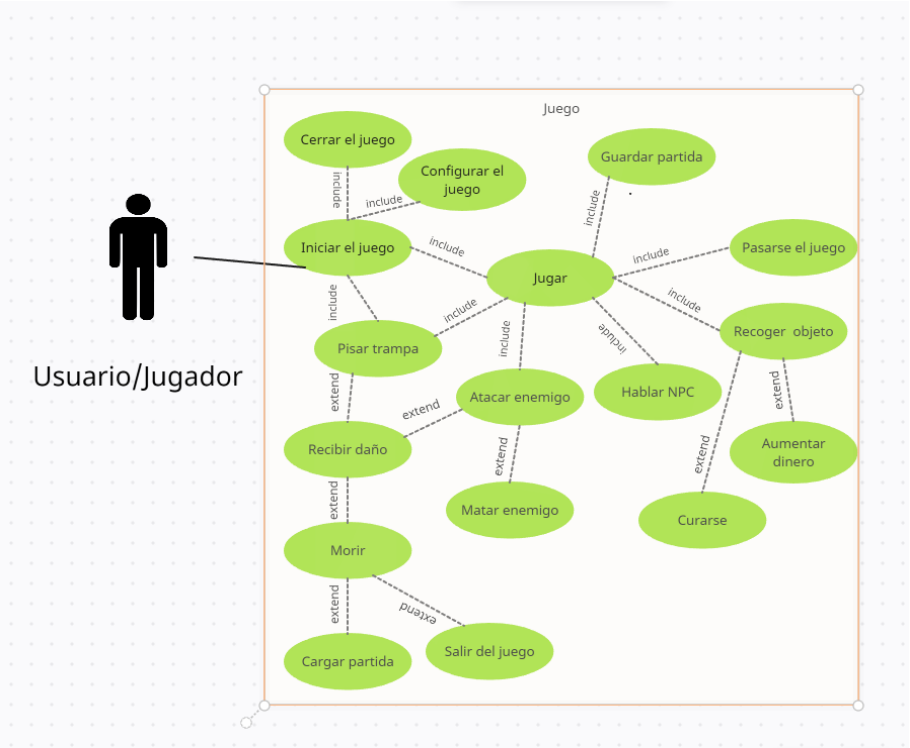

Figura 5. Diagrama de Casos de Usos del proyecto.

A continuación mostramos las fichas correspondientes a cada uno de los casos de uso.

Tarjeta del caso de uso: Cerrar el juego	
Actores:	Jugador.
Precondición:	El juego debe estar abierto.
Secuencia	El usuario pulsa sobre el botón de cerrar del menú o de la ventana. 2.
Postcondición	El juego se cierra.

Tarjeta del caso de uso: Iniciar el juego	
Actores:	Jugador
Precondición:	El juego debe estar cerrado.
Secuencia	El usuario pulsa sobre el icono de la aplicación
Postcondición	El juego se abre..

Tarjeta del caso de uso: Configurar el juego	
Actores:	Jugador.
Precondición:	El jugador debe estar en el menú de inicio.
Secuencia	El usuario pulsa sobre el botón de configuración.
Postcondición	El usuario es llevado al menú de configuración.

Tarjeta del caso de uso: Jugar	
Actores:	Jugador.
Precondición:	El jugador debe estar en la escena de juego.
Secuencia	El usuario realiza acciones dentro de la escena de juego.
Postcondición	El funcionamiento de la escena y sus elementos incluido el jugador responde correctamente.

Tarjeta del caso de uso: Pisar trampa	
Actores:	Jugador.
Precondición:	El jugador debe estar jugando.
Secuencia	El usuario colisionó con una trampa.
Postcondición	El usuario recibe un daño.

Tarjeta del caso de uso: Morir	
Actores:	Jugador
Precondición:	El jugador debe haber recibido daño.
Secuencia	Aparece el menú de muerte
Postcondición	El usuario puede cargar partida o salir del juego

Tarjeta del caso de uso: Atacar enemigo	
Actores:	Jugador
Precondición:	El jugador debe estar jugando.
Secuencia	El jugador ataca a un enemigo.
Postcondición	El enemigo muere tras varios ataques y el jugador puede haber recibido daño por parte del enemigo.

Tarjeta del caso de uso: Matar enemigo	
Actores:	Jugador.
Precondición:	El jugador debe haber atacado al enemigo.
Secuencia	El enemigo desaparece de la escena de juego.
Postcondición	El enemigo desaparece y el jugador permanece en la escena.

Tarjeta del caso de uso: Guardar partida	
Actores:	Jugador.
Precondición:	El jugador debe colisionar con los pilares de guardado.
Secuencia	El juego es guardado.
Postcondición	Al cargar el juego, tener la cantidad de vida, posición y dinero con la que se guardó.

Tarjeta del caso de uso: Pasarse el juego	
Actores:	Jugador.
Precondición:	El jugador mata al enemigo de la última sala del juego.
Secuencia	El enemigo desaparece.
Postcondición	El enemigo no vuelve a aparecer.

Tarjeta del caso de uso: Hablar NPC	
Actores:	Jugador.
Precondición:	El jugador debe colisionar con el NPC.
Secuencia	Aparece un cuadro de diálogo en la pantalla de juego.
Postcondición	Se puede leer el dialogo del NPC.

Tarjeta del caso de uso: Recoger objeto	
Actores:	Jugador.
Precondición:	El jugador colisionó con un objeto coleccionable
Secuencia	El objeto desaparece.
Postcondición	Los contadores de vida o dinero varían según el caso.

Tarjeta del caso de uso: Curarse	
Actores:	Jugador.
Precondición:	Recoger objeto
Secuencia	Se recoge el objeto poción del suelo.
Postcondición	Si la vida está reducida, esta se restablece un 25%.

Tarjeta del caso de uso: Aumentar dinero	
Actores:	Jugador.
Precondición:	Recoger objetos.
Secuencia	Se recoge el objeto moneda del suelo.
Postcondición	La cantidad de dinero es incrementada.

Diagrama de clases

En el Diagrama de Clases [7] se presentan las relaciones entre clases, atributos y métodos de los que disponen las clases del proyecto, tal y como se muestra en la Figura 6, a excepción de las clases relacionadas con los menús, puesto que se hará un diseño conceptual más adelante.

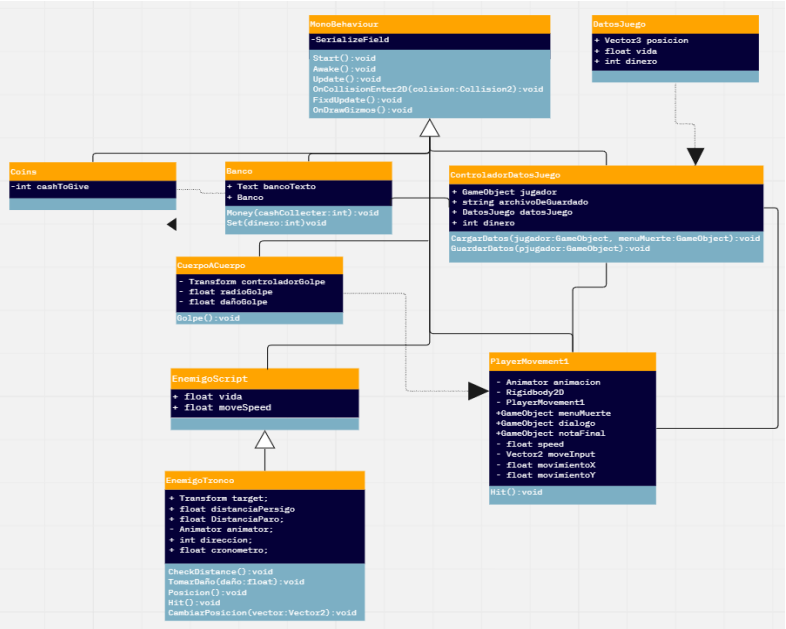

Figura 6. Diagrama de Clases del proyecto.

Diseño conceptual

Durante este capítulo se realizará el diseño conceptual del proyecto. Para ello, separaremos el capítulo en los tres siguientes apartados de diseño conceptual: interfaces, juego y niveles.

Diseño conceptual de las interfaces

En este apartado se mostrará un diseño conceptual de las interfaces que estarán presentes en el juego, incluyendo las del jugador denominadas IU Jugador, tal y como se plasma en la Figura 7. Se mostrará un diagrama con la estructura y una breve descripción.

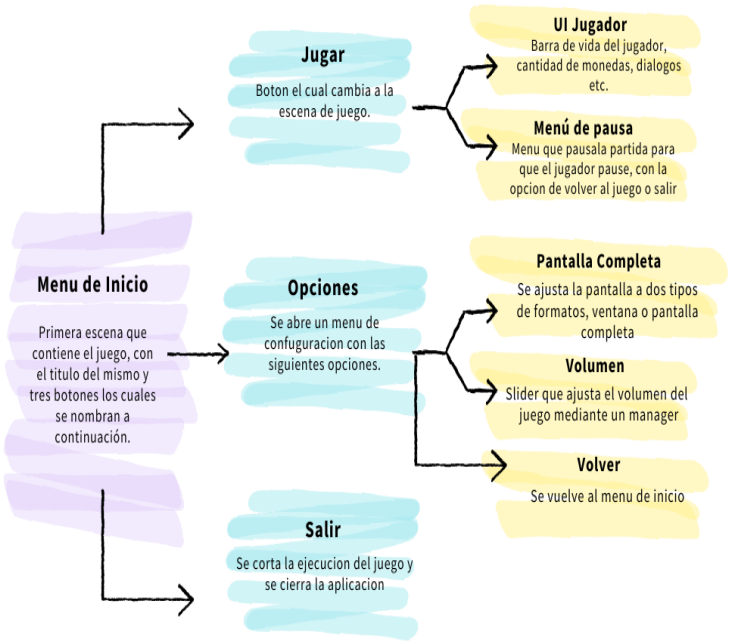

Figura 7. Diseño conceptual de las interfaces del juego y sus acciones.

Diseño conceptual del nivel

En este diseño conceptual simple (mostrado en la Figura 8) se intenta dar un primer esbozo de lo que va a ser el juego en lo que concierne al apartado del diseño del nivel y composición del mapa. Como se puede observar, está dividido en salas con temáticas. En la sala que está marcada como "Spawn" es donde se situará el personaje del jugador al inicio del mismo, más adelante se encontrará la zona del pueblo, donde se podrá interactuar con algunos NPC (*non-playable character*) y objetos. Seguidamente, se encuentra la primera zona de mazmorra, donde estarán situados los enemigos, trampas, pociones y monedas. Posteriormente se encontrará la zona de curación: una sala sin enemigos reservada para que el jugador se cure y pueda tener un respiro después de dos zonas de combate. A continuación habrá dos zonas de mazmorra que darán paso a una zona especial donde no hay enemigos, habrá recompensas y trampas. Finalmente, tras superar las dos últimas zonas de enemigos, se dará lugar a la sala del jefe. El juego se guardará mediante *checkpoints (puntos de guardado)* situados en la zona del pueblo, la zona de curación, la zona especial y la zona del jefe.

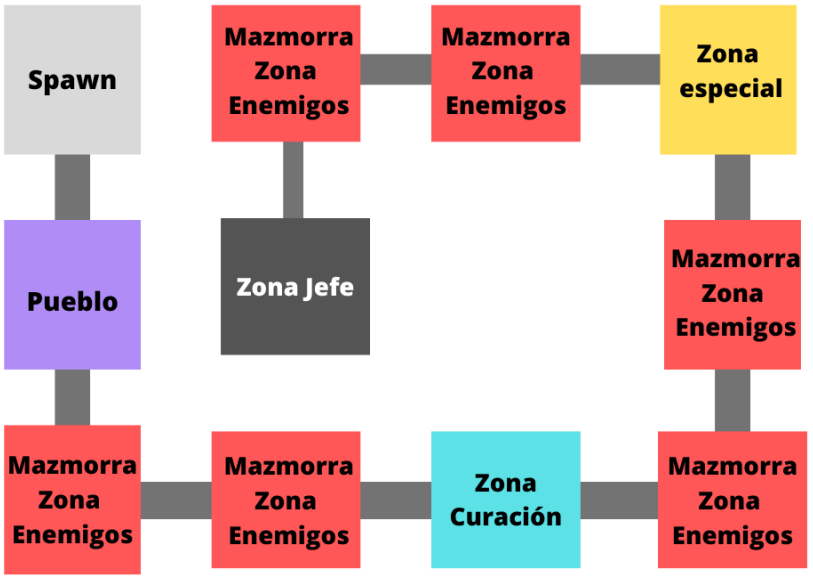

Figura 8. Esquema del mapa del juego.

Implementación

A continuación desarrollaremos el capítulo más largo de la memoria: la implementación del proyecto. En él explicaremos minuciosamente cada uno de los detalles de la implementación: *assets*, *tilemaps*, animaciones, efectos de sonido, interfaces y mecánicas del juego.

Assets

Los assets de un videojuego son todos aquellos elementos sonoros y visuales que lo componen, desde la música y efectos de sonido hasta los *sprites* y animaciones que tenemos en él.

Elección de *sprites* para el juego

Para la creación de escenarios, personajes y UI en este proyecto, se eligió una selección de sprites que cumplían los siguientes requisitos:

1. Eran gratuitos.
2. Fueron creados para un formato 16x16 píxeles (formato del juego).
3. Disponían de licencia de uso abierta.

Los *sprites* elegidos son los siguientes:

- Personaje principal, NPC y enemigos [8].
- Mazmorra, coleccionables y trampas [9].
- Diálogos y notas [10].
- Interfaces de usuario [11].

Tilemap

En este apartado explicaremos qué es un *tilemap* y cómo lo hemos utilizado en nuestro proyecto.

¿Qué es?

En Unity, un *Tilemap* es una herramienta que permite crear y organizar fácilmente elementos gráficos en forma de mosaicos (*tiles*) para construir niveles o escenarios en juegos 2D, tal y como se muestra en la Figura 9. Es una cuadrícula bidimensional compuesta por celdas o cuadros del mismo tamaño. Cada celda puede contener un *tile*, que es una imagen o *sprite* que representa una parte específica del nivel, como un bloque, un objeto o un terreno. Los *tiles* suelen ser gráficos pequeños y se colocan en la cuadrícula para formar la estructura y la apariencia del nivel.

Composición

Para la composición de nuestro `TileMap` en nuestra escena, hemos recogido las diferentes secciones de éste en un `Grid`, que se plasma en la Figura 10. El `Grid` puede ser especialmente útil cuando se trabaja en entornos 2D o 3D, ya que proporciona una referencia visual para ayudar a colocar objetos correctamente y mantener una alineación consistente. Una vez habilitado el `Grid`, se mostrará como una cuadrícula en el editor de Unity, lo que permitirá trabajar con mayor precisión al posicionar y alinear objetos en la escena.

Figura 9. *TileMap* del proyecto.

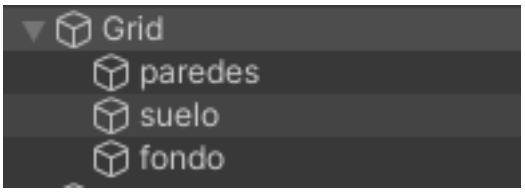

Figura 10. Composición del `Grid`.

Primera capa (pared)

Como se puede observar en la Figura 11, el `Grid` tiene asociados tres `TileMap`, cada uno de ellos haciendo referencia a una parte concreta del mapa, ya que es más eficiente dividir la construcción del mapa por capas en vez de todo en una sola.

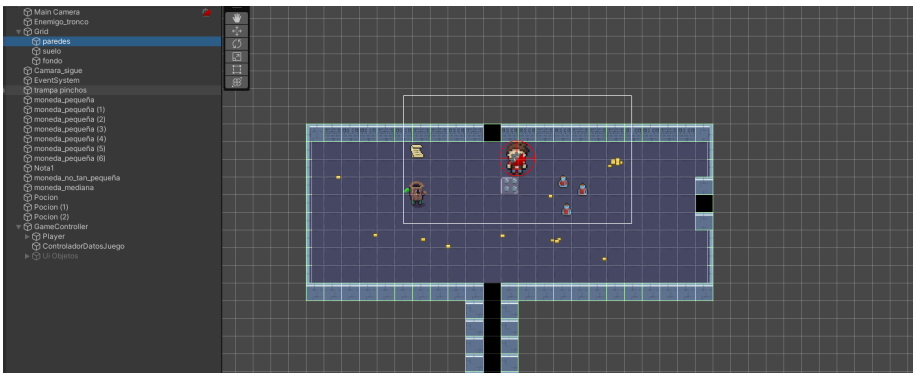

Figura 11. Composición de la primera capa.

Como se puede observar, esta capa hace referencia a los elementos que hacen función de pared en el videojuego. Al contar cada elemento con su `TileMap`, se puede juntar en una misma casilla varios elementos, como en este caso paredes y suelo.

Como puede observarse en la Figura 12, el elemento más destacable de las paredes, que no se da en ningún otro `TileMap` es el *Tilemap Colider 2D,* que hace que todos los elementos de este `TileMap` tengan cuerpo capaz de hacer colisión con otros cuerpos. De no tener el mapa dividido en elementos `TileMap`, deberíamos incluir a mano un `colider` a cada elemento que queramos que actúe como pared, de esta forma es automático.

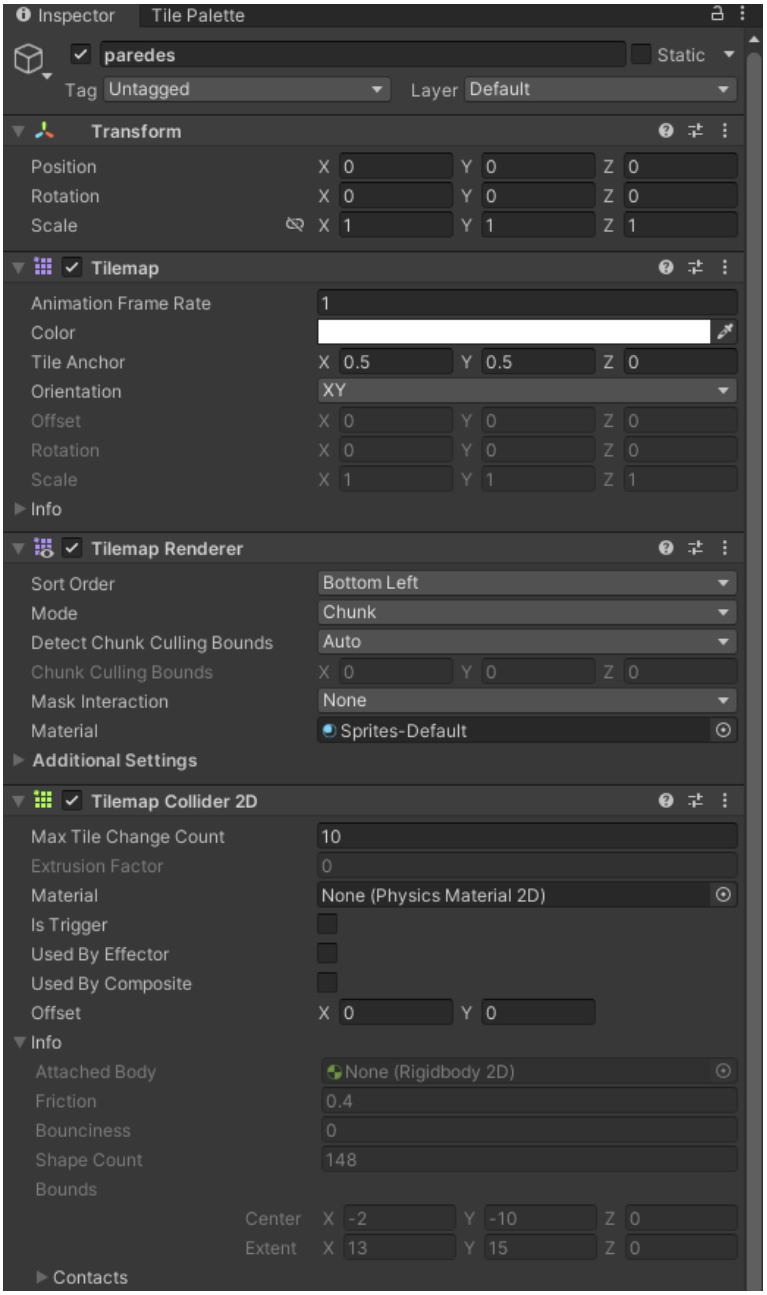

Figura 12. Propiedades del elemento pared.

Segunda capa (suelo)

Como puede observarse en la Figura 13, la capa de suelo es la capa intermedia entre el fondo y la pared. El jugador siempre estará situado encima de un suelo y nunca sobre una pared o un fondo, el suelo es una capa sin colider, es el terreno de acción delimitado por la capa de pared. Sus elementos se encuentran plasmados en la Figura 14.

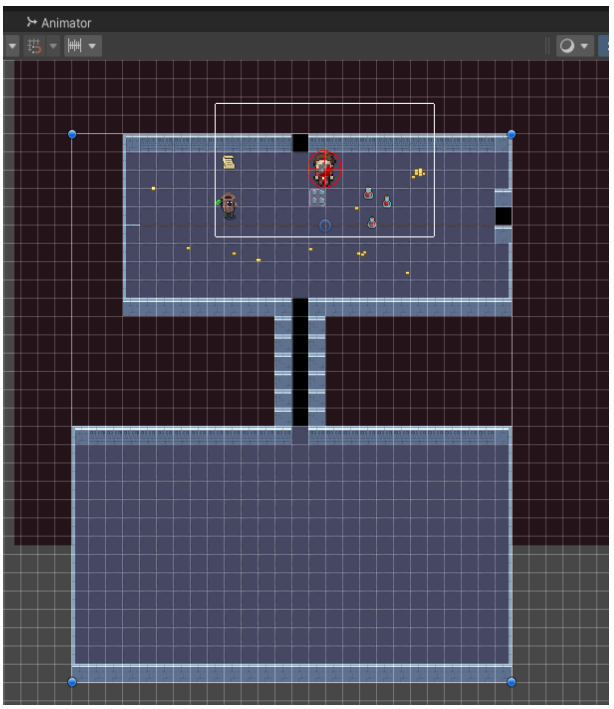

Figura 13. El suelo del `TileMap`.

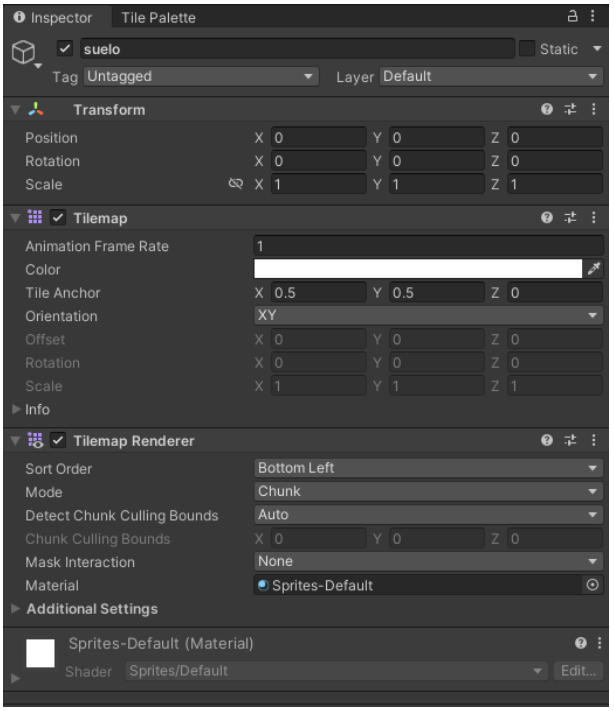

Figura 14. Propiedades del elemento suelo.

Tercera capa (fondo)

Como puede observarse en la Figura 15, la capa de suelo y fondo son similares, lo que diferencia al fondo del suelo es que el suelo es por donde el jugador se va a mover; el fondo es solo un elemento estético para no usar un fondo por defecto de Unity. Estos elementos son técnicamente iguales, excepto por el orden en el mapa, ya que como vemos, el suelo está por encima del fondo, se puede

decir que el fondo es aquello que va de las paredes hacia fuera y el suelo de las paredes hacia dentro. Se pueden observar sus propiedades en la Figura 16.

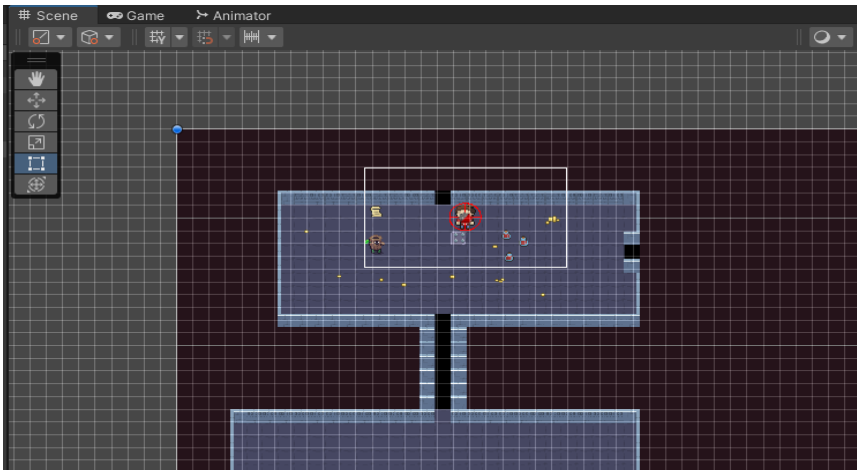

Figura 15. Capa que representa el fondo en nuestro `TileMap`.

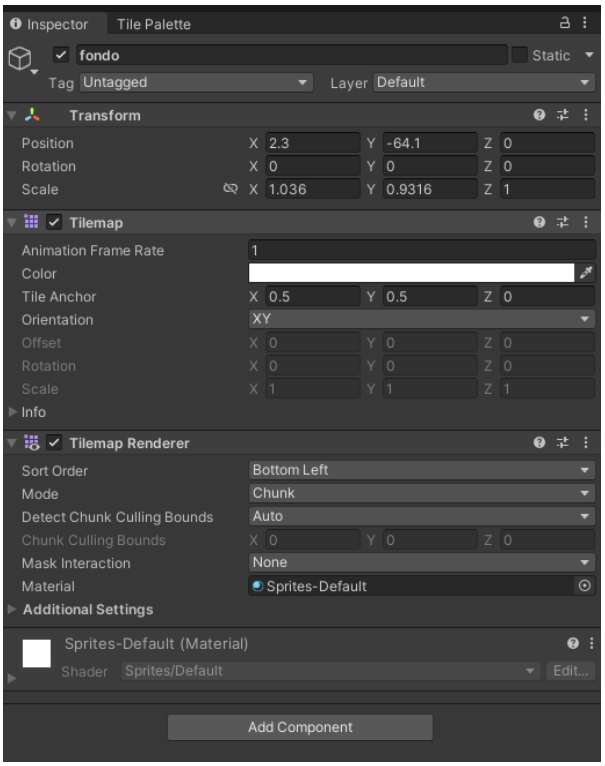

Figura 16. Propiedades del elemento Fondo.

Animaciones

Las animaciones son un elemento visual imprescindible para que un juego resulte atractivo. El objetivo de estas es dar vida y movimiento a objetos, personajes o elementos visuales dentro del juego. Consiste en crear una secuencia de imágenes en rápida sucesión que generan la ilusión de movimiento fluido. Estas imágenes, conocidas como cuadros clave o frames, se muestran consecutivamente a una velocidad determinada, lo que produce la sensación de animación.

Objeto *Animator*

En Unity, el objeto `Animator` es un componente que se utiliza para controlar y coordinar la animación de objetos dentro de un juego o una aplicación, tal y como se indica en la Figura 17. Su principal función es permitir la transición suave y controlada entre diferentes estados de animación. El objeto `Animator` de Unity es una herramienta esencial para la animación en tiempo real de objetos dentro del motor de juego. Permite crear y gestionar transiciones de animación fluidas y controladas, lo que contribuye a dar vida a los personajes y objetos en un entorno de juego.

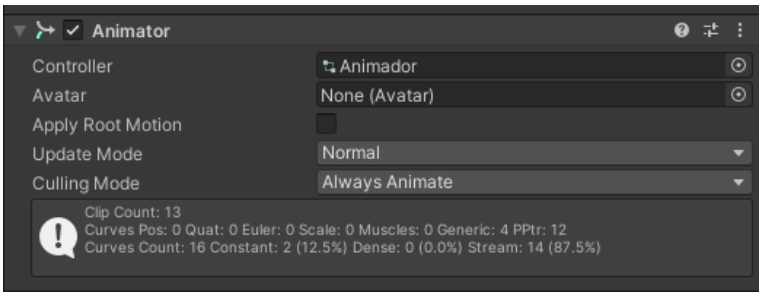

Figura 17. Ejemplo de un objeto `Animator`.

Creación de animaciones.

Para crear una animación en Unity debemos seleccionar un componente que tenga un `animator` vinculado, como se puede observar a continuación en la Figura 18.

Figura 18. Composición del jugador hasta el momento.

Una vez seleccionado tendremos acceso a una vista como la mostrada en la Figura 19.

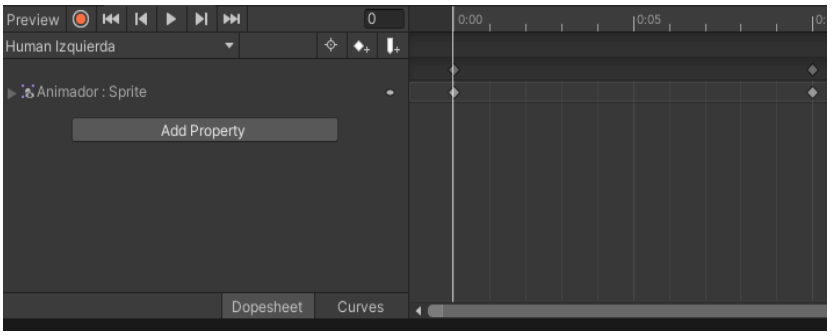

Figura 19. En esta vista haciendo uso de nuestro *sprites*, podrán crear las animaciones para el jugador, enemigos y elementos del `TileMap`.

Layer de animaciones

En Unity, el "*layer* de animaciones" (*animation layer* en inglés) se refiere a una funcionalidad del `Animator` que permite superponer y mezclar diferentes capas de animación. Cada capa de animación puede contener su propia secuencia de animación y parámetros de mezcla, lo que permite controlar y combinar varias animaciones de manera más flexible y compleja. El uso de capas de animación es especialmente útil en situaciones en las que se requiere combinar y sincronizar múltiples animaciones al mismo tiempo, como cuando se anima un personaje que camina mientras

lleva un arma y reacciona a eventos del entorno, como puede observarse en la Figura 20. Al asignar cada aspecto de la animación a una capa separada, se pueden mezclar y controlar de forma individualizada para lograr un resultado más realista y sofisticado. En resumen, el *layer* de animaciones en Unity es una característica del `Animator` que permite superponer y mezclar diferentes capas de animación, lo que proporciona un mayor control y flexibilidad en la reproducción de animaciones complejas y combinadas.

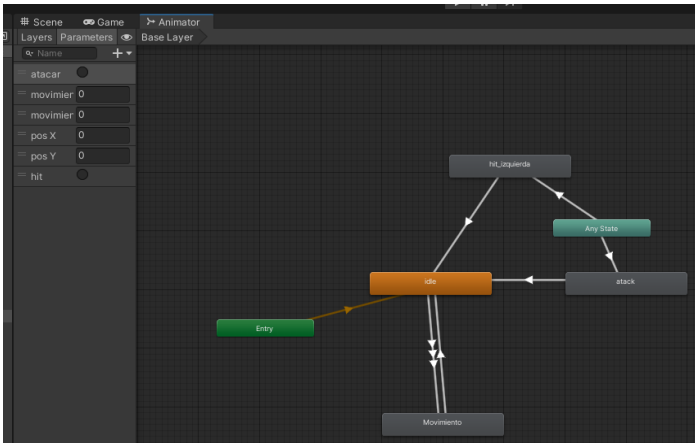

Figura 20. Layer de animación del jugador.

Blend Tree (Árbol de mezcla)

En Unity, un *blend tree* (árbol de mezcla) es una técnica utilizada para controlar y combinar animaciones de personajes de manera suave y fluida. Es especialmente útil para animaciones complejas que implican transiciones entre diferentes movimientos, como caminar, correr, saltar, atacar, etc. El árbol de mezcla organiza estas animaciones y define cómo se mezclan entre sí. Puede tener nodos de mezcla, que combinan múltiples *clips* basados en una combinación de parámetros y nodos de transición, que permiten crear transiciones suaves entre diferentes estados o movimientos. Los nodos de mezcla y la transición se conectan entre sí formando un árbol. El motor de animación de Unity utiliza este árbol para calcular y reproducir la animación adecuada según los parámetros proporcionados. El *blend tree* es una herramienta poderosa en Unity, ya que permite crear sistemas de animación complejos y realistas para personajes en juegos o aplicaciones interactivas. Proporciona un control preciso sobre cómo se combinan las animaciones y cómo responden a las diferentes situaciones del juego, lo que permite lograr movimientos suaves y naturales en tiempo real.

La Figura 21 hace referencia a un *blend tree* del proyecto, usado para la animación del movimiento del personaje. Teniendo dos parámetros en cuenta, `movimientoX` y `MovimientoY` que hacen referencia a los puntos cardinales en los que se encontraría el personaje.

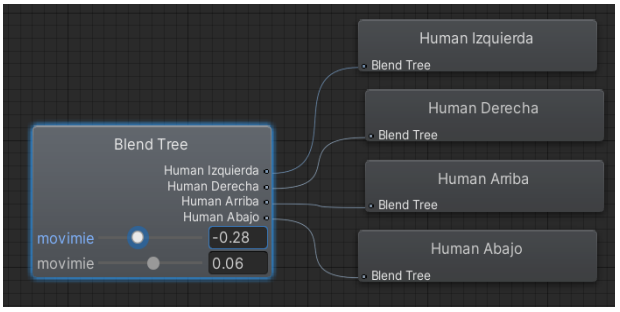

Figura 21. Según el valor de los parámetros, se ejecutaría una animación u otra.

Script de movimiento

Ahora que se ha visto cómo funciona el animador, el *layer* de animaciones y el *blend tree*, se verá un ejemplo de animación del proyecto, en este caso se mostrará el script de animación de movimiento del personaje principal. El objeto de nuestro personaje principal cuenta con un componente que es mostrado en la Figura 22, el cual cuenta con tres parámetros: el código de C# asociado, un parámetro público llamado `speed` (que puede modificarse a gusto y que se dejó público a fines de un testeo más rápido), y los parámetros *input* que también son lúdicos para ver los valores de los ejes X e Y en tiempo de ejecución, aunque en este caso no se le pasa nada.

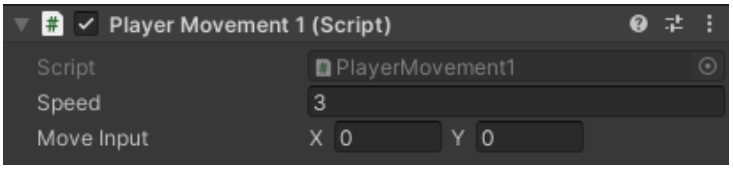

Figura 22. Script asociado al jugador.

El script `PlayerMovement1` es un *script* grande, ya que recoge casi todos los funcionamientos del personaje principal. En este caso se marcará lo relativo al movimiento y la animación, lo demás se explicará más adelante.

Para definir dicho *script*, se crea una relación de herencia entre el *script* y la clase `MonoBehaviour` (que es una clase clave a la hora de desarrollar videojuegos en Unity con métodos y propiedades indispensables). El método `Start()` es el método que empieza la ejecución del código, se deben iniciar los elemento o asignar los valores de las variable saque se van a utilizar previamente en este código si es que no los tienen ya. Por otra parte, el método `Update()` es un método que está constantemente en ejecución, lo que permite recoger, en cada instante, los valores de las flechas de nuestro teclado en los atributos `movimientoX` y `movimientoY`. Además con el método `SetFloat()` del objeto `animacion` estamos pasando al *blend tree* el valor que tendrán los parámetros de tipo `float` llamados `movimientoX` y `movimientoY`. Posteriormente, se evalúa si los valores son cero; de no serlo, se les pasará ese mismo valor a los parámetros `posX` y `posY` que determinan la posición de nuestro personaje principal.

Dentro de este método tendremos que controlar también la normalización del movimiento (de forma que no se vaya más rápido en diagonal que en línea recta). Para ello, le pasamos a la propiedad `moveInput` un nuevo `Vector2` que es un objeto propio de la clase. El valor que tendrá esta propiedad es el `movimientoX` y `movimientoY` aplicando el método `normalized()`, que aplicará esa normalización sobre el movimiento.

A la instancia del objeto `RigiBody` llamada player `RigiBody` utilizando el método `MovePosition()`, método propio de la clase, le pasamos el propio objeto `RigiBody` sumandole el movimiento utilizado (la tecla de movimiento utilizada) multiplicado por la velocidad (en este caso tres) y multiplicado, también, por el tiempo en formato `delta`. En este punto, el *script* de movimiento estará finalizado.

Efectos de sonido y música

La música es un aspecto fundamental en cualquier videojuego, ya que es la que cobrará parte del protagonismo del juego. Para este proyecto se ha elegido una canción sin derechos de autor y de uso gratuito. La música elegida [12] puede encontrarse en el apartado de la bibliografía.

Para utilizar la música en el proyecto, se usó el siguiente sistema: se creó un `AudioManager`, que es un objeto vacío al que se le dió una propiedad `transform` para ubicarlo en nuestro juego y un *script* sencillo que evita que el objeto se destruya al cambiar de escena. De esa forma, la música se escucha sin interrupciones o reinicios, durante todo el juego. Su configuración es mostrada en la Figura 23.

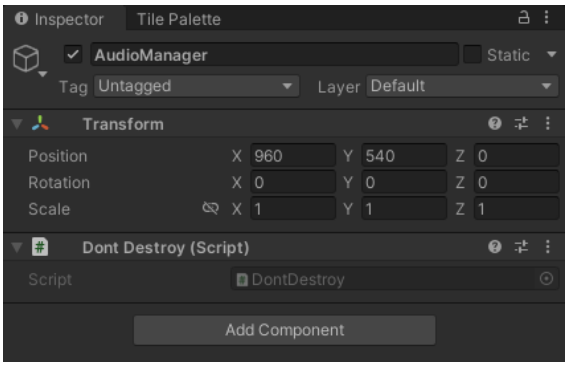

Figura 23. Instancia del objeto AudioManager.

Al utilizar el método `DontDestroyOnLoad` se puede ver que se ha creado un objeto llamado con su mismo nombre y que tiene vinculado el objeto recién creado `AudioManager`. Esto es debido a que todos los objetos del proyecto que no se quieran destruir al cambiar de escena pasarán a ser parte de `DontDestroyOnlLoad`, que es un objeto definido en Unity para ser compartido por todas las escenas, por lo que los objetos vinculados a él también. Se puede observar como el `AudioManger` también tiene vinculado otro objeto, el objeto es el mostrado en la Figura 24.

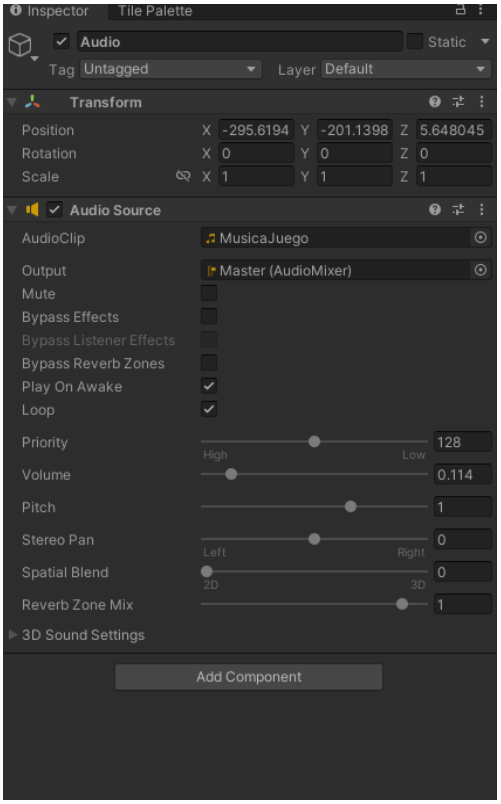

Figura 24. El objeto audio cuenta con la propiedad `AudioSouce` (fuente de audio), por la cual se indicará la canción a sonar en el juego. Además, se marcará la casilla de *loop* para que la canción no finalice en ningún momento.

Script del audio

Tal y como se puede observar en la Figura 25, le pasamos al objeto que hace de canvas de nuestro menú principal (se expondrá y explicará más adelante que es y su funcionamiento) el `AudioMixel` por un parámetro del *script* `MenuOpciones` que veremos en la Figura 26.

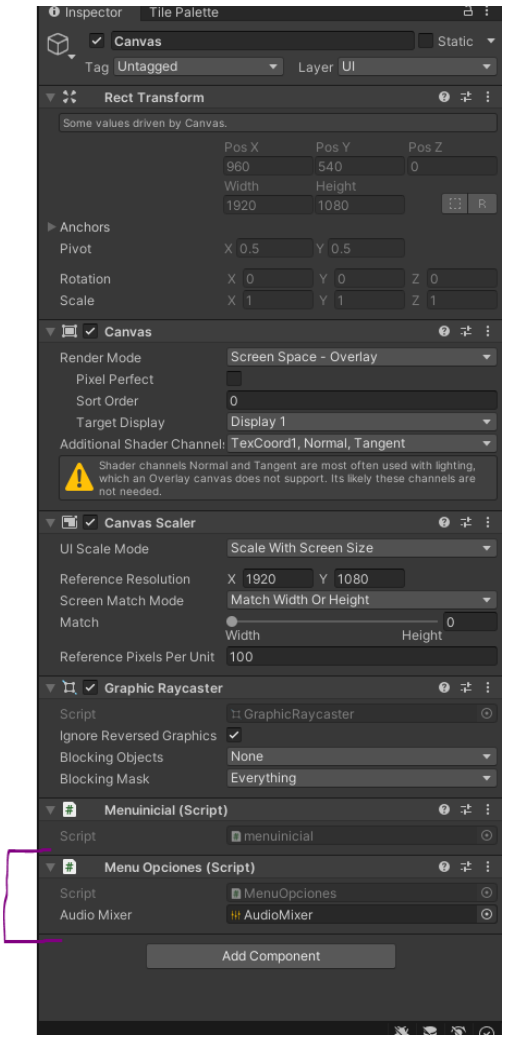

Figura 25. Canvas del menú de inicio.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

Script de Unity (2 referencias de recurso) | 0 referencias
public class MenuOpciones : MonoBehaviour
{
    [SerializeField] private AudioMixer audioMixer;
    0 referencias
    public void PantallaCompleta(bool pantallaCompleta) {
        Screen.fullScreen= pantallaCompleta;
    }

    0 referencias
    public void CambiarVolumne(float volumen)
    {
        audioMixer.SetFloat("Volumen", volumen);
    }
}
```

Figura 26. Script para variar el volumen del juego

Se explicará más en detalle la clase en el apartado de interfaces de usuario dentro de mecánicas del juego. En este punto el método que se utiliza de esta clase no es otro más que el llamado `CambiarVolumen`, que como se puede observar recibe un `float` y en base a él utilizando el método `SetFloat` del objeto `AudioMixer` que le hemos pasado por parámetros, ajusta el volumen. El `float` que recibe es pasado mediante una interfaz de usuario de tipo `Slider` que se ha configurado en un menú de opciones, como puede observarse en la Figura 27.

Figura 27. Barra de volumen del menú de configuración.

La barra de volumen está vinculada al objeto volumen que tiene una propiedad llamada `Slider` a la que se le ha pasado la barra como `TargetGraphic` y la cual al moverse de un lado a otro devuelve un valor de tipo `float`, siendo el menor -50 y el mayor 15, mínimo y máximo volumen respectivamente, tal y como se refleja en la Figura 28.

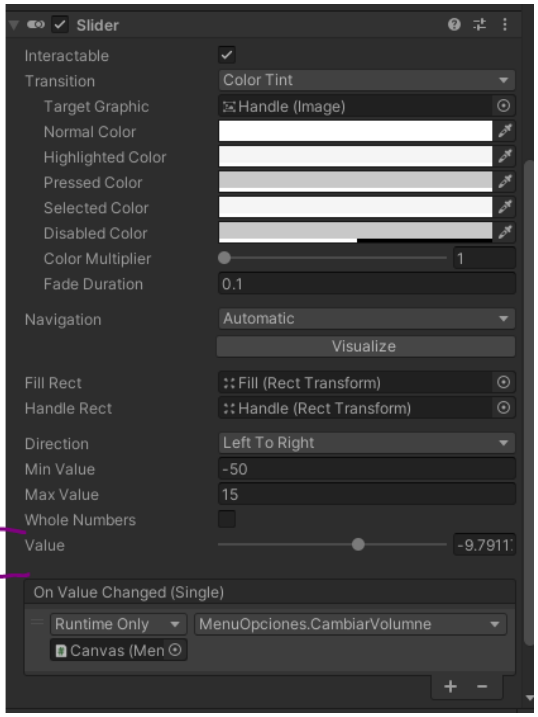

Figura 28. Slider, elemento de control del volumen en el juego

Interfaces de usuario

Siguiendo el diseño conceptual de las UI del proyecto explicado anteriormente, crearemos las interfaces de usuario del proyecto y sus correspondientes *scripts*.

Menú de inicio

En la escena llamada "Menú de inicio del proyecto", tenemos los elementos correspondientes al menú de inicio y al menú de opciones. A continuación, en la Figura 29, se presenta la composición y estructura del menú de inicio.

Figura 29. El menú de inicio es un *Canvas,* el cual contiene un panel que a su vez tiene tres botones asociados y una etiqueta.

El *Canvas* tiene el atributo `canvasscaler` para poder adaptar la pantalla del jugador. En este caso, se le han pasado los valores más comunes de pantalla (1920x1080 píxeles). Además, este *canvas* tiene asociados dos *scripts:* `menuinicial` y `MenuOpciones;` el último, visto anteriormente en *Script de audio*.

El menú principal es, de nuevo, un *script* que hereda de la clase `MonoBehaviour` y el cual solo contiene dos métodos, vinculados a los botones llamados de su misma forma. El botón "Jugar" permite cambiar la escena en la que se encuentra el jugador, dando paso al comienzo del juego. Esto es posible dado que se utiliza el método `LoadScene` del `SceneManager`, al cual se le invoca utilizando el número del índice de la escena a la que se quiere pasar (al solo haber dos es fácil, ya que solo se suma uno al valor de la escena actual). El método llamado `Salir`, utilizando el método `Quit` de la clase `Application`, hace que el juego finalice. El botón restante, llamado "opciones", tiene la función de dar paso al menú opciones de la siguiente manera: los botones tienen un método llamado `OnClick`, que determina qué se ejecuta cuando se pulsa sobre el botón, en este caso lo hemos configurado para que al pulsar en él se desactive el menú inicial y se active el menú de opciones, dando paso a un cambio de escena limpio.

Menú de opciones

Una vez el menú de opciones ha sido lanzado, nos queda en pantalla el siguiente *canva* representado en la Figura 30.

Figura 30. Menú de opciones.

La composición del menú de opciones (cuyo código se encuentra en la Figura 31) es la siguiente: un *canvas*, un panel, una etiqueta en la que se muestra el mensaje de "opciones", un botón que indica"pantalla completa" (el cual tiene la función de cambiar el estado de la ventana del juego a pantalla completa o devolverla a su estado inicial), un *Slider* que ya se explicó en el apartado *Script del audio* y, finalmente, un botón que muestra el mensaje "volver" que tiene la misma función que el botón citado anteriormente, pero a la inversa: desactiva el menú de opciones y activa el menú de inicio.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

Script de Unity (2 referencias de recurso) | 0 referencias
public class MenuOpciones : MonoBehaviour
{
    [SerializeField] private AudioMixer audioMixer;
    0 referencias
    public void PantallaCompleta(bool pantallaCompleta) {
        Screen.fullScreen= pantallaCompleta;
    }

    0 referencias
    public void CambiarVolumne(float volumen)
    {
        audioMixer.SetFloat("Volumen", volumen);
    }
}
```

Figura 31. El *script* vinculado al menú de opciones.

Como se ha podido observar en la Figura 31, el metodo `PantallaCompleta`, al cual se le invoca con un atributo lógico, usa el atributo `fullScreen` de la clase `Screen`, permitiendo cambiar el formato de la ventana del juego a completa o ventana, según su estado actual. Para implementarlo, se usó la propiedad `OnClick` del botón correspondiente.

Menú de pausa

El menú de pausa se activa en cualquier momento del juego pulsando el botón de pausa disponible en la esquina superior derecha de la pantalla. Este botón sirve para llamar al menú de pausa del juego, el cual congela el tiempo del juego (lo pone en pausa) y también permite al usuario salir del juego. Todos los elementos vistos en pantalla estando en la escena del juego y no en la de menú de inicio están anclados al canvas "UI Objetos", que se encarga de agrupar y controlar los diferentes elementos dentro de la escena de juego, con sus diferentes scripts.

En las Figuras 32-33 se muestra la posición del botón y el menú descrito, respectivamente.

Figura 32. Botón de pausa del juego.

Figura 33. Menú de pausa.

El funcionamiento del botón de pausa es idéntico al del botón opciones que se vió en el subapartado *Menú de inicio*, solo que en este caso, además, se establece el tiempo del juego a cero en el *script* asociado. Además, se asocia a la propiedad `OnClick` del botón la inicialización del *canvas* del menú pausa y la desactivación del propio botón. El botón "volver" del menú pausa hace exactamente lo mismo, pero llamando a la escena de juego y desactivando el panel del menú pausa. Finalmente, al botón "salir" del menú pausa, se le asocia un método llamado `Cerrar` de la clase.

Interfaz de jugador

La interfaz del jugador tiene como objetivo proporcionar información del estado del jugador y del juego en la pantalla.

Como se puede observar en la Figura 34: la barra la vida aparece en la esquina inferior izquierda, la cantidad de oro lo hace en la esquina superior izquierda y, finalmente, el botón de pausa está situado en la esquina superior derecha.

Figura 34. Elementos que componen la interfaz del jugador.

Barra de vida

La barra de vida es un objeto *slider*, anteriormente mencionado, que tiene un funcionamiento similar al de la barra de volumen, solo que esta vez se ocupará de marcar la vida del personaje. Anclado al objeto `BarraVidaFondo`, se encuentra el objeto `ControladorVida`, que es un objeto con la propiedad de imagen al cual se le ha pasado una imagen de color rojo para hacer de barrra de vida. Para lograr que la zona roja de la barra de vida disminuya a la hora de recibir daño, se asigna a la propiedad `image` del objeto actual (que es del tipo `Filled` y tiene una propiedad

–`FillMethod`– que marcaremos en horizontal para que la barra disminuya por la derecha). La barra, en definitiva, disminuirá según un float almacenado en la propiedad `FillAmount`. En este caso, posee el valor 1, que permite que la barra de vida se vea entera. Si disminuye su valor, ésta también lo hace, tal y como se muestra en la Figura 35.

Figura 35. Ejemplo visual de cómo queda la barra de vida reducida tras un impacto enemigo.

Monedas

Tal y como se ha dicho previamente, en la esquina superior izquierda existe una imagen (un montón de monedas) y el texto, que hace referencia a la cantidad de monedas que posee el personaje. Ésto es apreciable en la Figura 36.

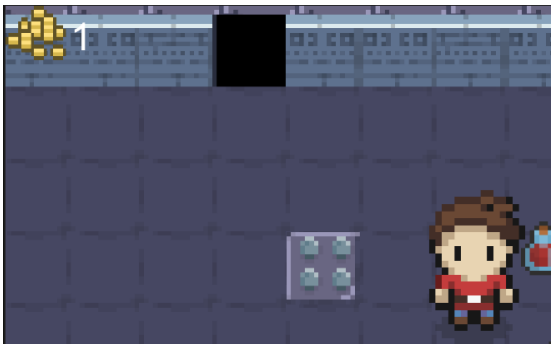

Figura 36. *Zoom* sobre la interfaz del jugador que muestra, de forma más detallada, las monedas del jugador.

Script monedas

A continuación se explicará cómo se ha logrado que el personaje, al recoger una moneda, la elimine y aumente el número de dinero en la pantalla. En este caso, el *script* asociado al objeto moneda se llama `Coins`.

Existe una clase `Coins` que se ocupa de que el objeto asociado al *script*, tras ser colisionado por el jugador, añada al atributo `Money` de la clase `Banco` el entero que le asociamos al *script* y, después, destruya el objeto asociado. Es importante notar que existen varios montones de monedas en el juego: algunos dan más monedas que otros. De esta forma se justifica que exista un método genérico al que se le indique la cantidad de oro que se ha recogido en el momento de la colisión. Otro detalle a tener en cuenta es que, en la clase `Banco`, se utiliza el patrón Singleton para evitar que este objeto se repita.

Mecánicas de juego

En este apartado se explicarán las mecánicas dentro del proyecto y se expondrá su implementación.

¿Que es una mecánica en un videojuego?

En el contexto de los videojuegos, una mecánica de juego se refiere a las reglas y sistemas que definen la interacción entre el jugador y el mundo del juego. Estas mecánicas establecen cómo el jugador puede controlar y afectar el entorno del juego, así como la respuesta del propio juego ante esas acciones.

Combate

El combate es la mecánica de juego por la cual un jugador puede herir y matar a un enemigo y viceversa.

Jugador

A continuación se explicarán las mecánicas de ataque del jugador, de recibir daño (impacto del jugador con un enemigo o con una trampa) y la modificación de la vida del jugador utilizando el script de vida.

Ataque del jugador

Para que el jugador pueda golpear al enemigo se utilizó la siguiente metodología:

- Se creó una clase que contenía tres variables: la primera, un *transform* para la posición del enemigo; la segunda, un `float` que marcase el radio del golpe; y, por último, otro `float` que marcase el daño que realizarán los golpes del jugador.
- En el método `Update` (ya mencionado en anteriores ocasiones), se evalúa si se ha pulsado el botón derecho del ratón y, de ser así, se llama al método `Golpe`. Este método se encarga de instanciar el *colider* del ataque que se va a lanzar, con su posición y lugar. Además, llama al método `Hit` que lanzará la animación de daño del enemigo.
- Posteriormente, se llama al método `TomarDaño`, que infligirá tanto daño al enemigo como valor tenga la variable `dañoGolpe`.
- Finalmente, se debe cambiar la posición del enemigo, como si el golpe le empujase, utilizando un método de la clase del enemigo.
- El último método de la clase permite dibujar el radio del golpe en el elemento `Scene`, para que, en la hora de la configuración, saber cuánto abarca el área del golpe con certeza.

Todo lo descrito anteriormente se muestra en la Figura 37, marcando el radio de ataque del jugador; y en la Figura 38, en forma de código.

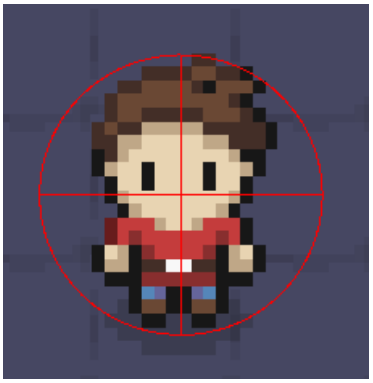

Figura 37. Radio de ataque del jugador.

```
Script de Unity (1 referencia de recurso) | 0 referencias
public class CuerpoACuerpo : MonoBehaviour
{
    [SerializeField] private Transform controladorGolpe;
    [SerializeField] private float radioGolpe;
    [SerializeField] private float dañoGolpe;

    Mensaje de Unity | 0 referencias
    private void Update()
    {
        if (Input.GetButtonDown("Fire1"))
        {
            Golpe();
        }
    }

    1 referencia
    private void Golpe()
    {
        EnemigoTronco a = new();
        Collider2D[] objetos = Physics2D.OverlapCircleAll(controladorGolpe.position, radioGolpe);
        foreach (Collider2D colisionador in objetos)
        {
            if (colisionador.CompareTag("Enemigo"))
            {
                colisionador.transform.GetComponent<EnemigoTronco>().Hit();
                colisionador.transform.GetComponent<EnemigoTronco>().TomarDaño(dañoGolpe);
                var pos = colisionador.transform.GetComponent<EnemigoTronco>().Posicion();
                colisionador.transform.GetComponent<EnemigoTronco>().CambiarPosicion(pos);
            }
        }
    }

    Mensaje de Unity | 0 referencias
    private void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(controladorGolpe.position, radioGolpe);
    }
}
```

Figura 38. Clase que gestiona el ataque del jugador.

La animación de ataque se activa en el método `Update` de la clase del jugador cuando se pulsa el botón derecho del ratón.

Recibir daño jugador: Script de vida

A continuación se mostrará el método que se encarga de gestionar el daño y la curación recibida por parte del jugador. El método `onColision2D`, marca lo que sucederá en las colisiones que tenga el jugador con los diferentes elementos interactivos.

La clase `ControladorVida` utiliza el patrón Singleton para tener la certeza de que solo haya una instancia de esa clase, para poder mantener el funcionamiento correcto del código y el juego. Dentro del *script* de los enemigos, existe un método que les permiteherir al personaje.

En el método `Update` del personaje se controla si la vida baja hasta el valor cero y, en tal caso, se activa la destrucción del objeto jugador y se ejecuta su animación de muerte.

Además, se muestra por pantalla el panel de muerte, mostrado en la Figura 39.

Figura 39. Un panel que indica nuestra muerte y dos botones: uno para recargar la partida desde el último punto de guardado y otro para cerrar el juego.

Enemigo

A continuación explicaremos cómo se han implementado los diferentes enemigos del juego, así como sus mecánicas básicas.

Herencia en enemigo

Los enemigos del juego heredan dos atributos de su clase padre `Enemigo`: la vida y la velocidad de movimiento. De esta forma se pueden definir de forma sencilla distintos tipos de enemigo, que compartirán las características comunes que tienen todos los enemigos, sin tener que repetir código.

Ataque enemigo

El sistema de ataque del enemigo es realmente simple. En el método que se muestra en la Figura 40, se puede ver cómo se evalúa si el *colider* que ha entrado en colisión con el enemigo pertenece a un jugador y, de ser así, se activará la animación de daño del jugador, se moverá hacía atrás el enemigo y, finalmente, se llamará al método estático `Herir`, de nuestro controlador de vida, que bajará en veinticinco puntos la vida del enemigo.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Player")
    {
        //que reste vida
        collision.transform.GetComponent<PlayerMovement1>().Hit();
        CambiarPosicion(transform.position);
        ControladorVida.Herir();
    }
}
```

Figura 40. Sistema de colisión entre los enemigos y el jugador.

Recibir daño enemigo

Como se vió en la subsección *Ataque del jugador*, el jugador, al atacar al enemigo, llama a este método (que se ocupa de recoger el valor pasado por parámetro, que es el daño del ataque) y restarlo a la vida actual del enemigo. También tiene en cuenta que, si la vida del enemigo está por debajo de uno, se debe destruir el enemigo. Toda esta funcionalidad es mostrada en la Figura 41.

```
public void TomarDaño(float dañoGolpe)
{
    vida -= dañoGolpe;
    float vidaActual= vida;
    if (vida <1)
    {
        //animator.SetTrigger("Morir");
        if (!gameObject.IsDestroyed() && vida < 1)
        {
            Destroy(gameObject);
        }
    }
}
```

Figura 41. Gestión del daño en los enemigos.

Detección y persecución

La función de persecución de los enemigos para seguir al jugador es explicada a continuación.. El método `update`, que está constantemente siendo invocado, llama al método `CheckDistance`, que evaluará si la posición del personaje con respecto al enemigo es menor o igual a la variable

`distanciaPersigo` y menor a la variable `distanciaParo`, a ambas variables se les asigna un valor en la vista que proporciona Unity de la instancia del objeto enemigo en su *script*.

Script de carga y guardado

El guardado de datos se hará sobre un archivo JSON (JSON es un formato de texto sencillo para el intercambio de datos) donde se almacenará la vida, monedas y posición del jugador, con el *script* mostrado en la Figura 42.

El método `CargarDatosJuego`, asigna a los atributos correspondientes la vida, posición y dinero (que son los datos almacenados en el JSON). Esta función está asignada a la letra F6 del ordenador. Por otro lado el método de guardado, aunque se asigne en el botón F5, se utilizará cuando el jugador entre en colisión con el objeto `Guardado`. Este objeto tiene asociado al evento `OnColider2D` la función de guardado cuando el que colisiona es el jugador.

```
Script de Unity (22 referencias de recurso) | 4 referencias
public class ControladorDatosJuego : MonoBehaviour
{
    public GameObject jugador;
    public string archivoDeGuardado;
    public DatosJuego datosJuego=new DatosJuego();
    public int dinero;
    Mensaje de Unity | 0 referencias
    private void Awake()
    {
        //localizacion de la carpeta donde estamos trabajando
        jugador = GameObject.FindGameObjectWithTag("Player");
        //CargarDatos(jugador);
    }
    Mensaje de Unity | 0 referencias
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.F6))
        {
            CargarDatos(jugador,null);
        }
        if (Input.GetKeyDown(KeyCode.F5))
        {
            GuardarDatos(jugador);
            Debug.Log("Archivo Guardado");
        }
    }
    2 referencias
    public void CargarDatos(GameObject jugador, GameObject menuMuerte) {
        if (File.Exists(Application.dataPath + "/datosJuego.json"))
        {
            string contenido=File.ReadAllText(Application.dataPath + "/datosJuego.json");
            datosJuego=JsonUtility.FromJson<DatosJuego>(contenido);
            Debug.Log("La posicion del jugador es: " + datosJuego.posicion);
            jugador.transform.position = datosJuego.posicion;
            ControladorVida.SetVida(datosJuego.vida);
            Banco.Set(datosJuego.dinero);
            if (menuMuerte != null)
            {
                menuMuerte.SetActive(false);
            }
            Time.timeScale = 1;
        }
        else
        {
            Debug.Log("El archivo no existe");
        }
    }
    2 referencias
    public void GuardarDatos(GameObject pjugador)
    {
        DatosJuego nuevosDatos = new DatosJuego()
        {
            posicion = pjugador.transform.position,
            vida = ControladorVida.instance.vida,
            dinero = Banco.instance.banco,
        };
        string cadenaJson=JsonUtility.ToJson(nuevosDatos);
        File.WriteAllText(Application.dataPath + "/datosJuego.json", cadenaJson);
        Debug.Log("Archivo Guardado");
    }
}
```

Figura 42. Clase de carga de datos.

Validación y pruebas

En este apartado se explicaran y expondrán ejemplos reales de las diferentes pruebas que se han tenido en consideración para validar el correcto funcionamiento del proyecto.

Pruebas unitarias

En Unity, las pruebas unitarias son un enfoque utilizado para verificar el funcionamiento individual y aislado de componentes o unidades de código dentro de un proyecto. Estas pruebas se centran en probar pequeñas secciones de código de manera independiente para asegurarse de que funcionen correctamente.

Las pruebas unitarias realizadas en este proyecto serán las comprobaciones de la integración y validación del código en el juego. Por ejemplo, que al ejecutar el script de movimiento del jugador, se arrancó la consola de Unity para verificar si efectivamente el jugador se movía; de no moverse, se tendrá que depurar el código hasta encontrar el motivo por el cual este no funciona. Otro ejemplo de una prueba unitaria: el sistema de vida. En este proyecto, en un inicio, se presentó con la idea de ser tres corazones en vez de una barra de vida. Sin embargo, a la hora de eliminar estos corazones y al cambiar de escena, este sistema de vida daba más problemas que la solución adoptada finalmente, puesto que había que destruir los corazones cada vez y, al curarlos, había que construirlos de nuevo. Como eran copias de una instancia de corazón (no eran los mismos), así que no pudo solucionarse con un patrón Singleton. De esta forma, se afrontó el problema modificando el sistema por una barra de vida, que era un único objeto que representaba un valor el cual era más fácil hacer variable.

Este problema se pudo encontrar y solucionar gracias a las pruebas unitarias que se han ido implementando durante el proyecto.

Pruebas de integración

Las pruebas de integración se realizan después de que se hayan completado las pruebas unitarias y se centran en probar la interacción entre módulos, subsistemas o componentes completos del sistema. Estas pruebas se utilizan para detectar problemas que pueden surgir debido a la combinación de diferentes elementos y asegurar que funcionen correctamente en conjunto.

En Unity, dado que lo que se está creando es un videojuego, la mejor forma de hacer una prueba de integración es jugarlo. Así que una vez se terminaron las pruebas unitarias, se jugó con la intención de encontrar errores en la integración de todas las partes. Por ejemplo, en la primera prueba se pudo observar cómo las monedas esparcidas por el juego también eran recogidas por los enemigos y el dinero era añadido al banco del jugador, esto era un error terrible. Se solucionó. El error estaba en el código de las monedas, en el método `OnColider2D`, ya que no se discriminaba correctamente quién colisionaba con el objeto, por lo que cualquier *colider* que chocase con las monedas activaba el *script*.

Este error se encontró gracias a la realización de pruebas de integración.

Conclusiones y trabajo futuro

Para finalizar la memoria, plasmamos las conclusiones del proyecto y las líneas de trabajo futuro.

Conclusiones

A continuación, se expondrán las conclusiones a las que se llegó al realizar este trabajo, lo aprendido y las dificultades más grandes superadas.

Dificultades superadas

Las dificultades más notables superadas han sido sin duda la toma de decisiones sobre la implementación de las diferentes formas de enfocar una solución a problemas tales como la implementación de patrones Singleton (para evitar que objetos que no pueden repetirse efectivamente no lo hicieran). El sistema de diálogos fue un reto, dado que no había documentación accesible para poder crear una que se adaptara al proyecto. Y, sobre todo, el trabajar con un entorno totalmente nuevo y la creación de las animaciones, ya que el creador de este proyecto carece de gusto y habilidad para todo aquello relacionado con un medio visual.

Aprendizaje

Durante el proyecto se han adquirido conocimientos y habilidades que se referencian a continuación:

- Buen nivel de manejo con el motor Unity.
- Creación de animaciones.
- Se afianzó los conocimientos en C# y la programación orientada a objetos.
- Se afianzó los conocimientos sobre gestión de interfaces de usuario.
- Se aprendió a guardar información en archivos JSON.
- Se afianzo los conocimientos sobre la implementación del patrón Singleton.
- Se aprendió a solucionar y gestionar los problemas y quehaceres de un proyecto.
- Se aprendió la búsqueda de información en la documentación oficial e investigación propia.

Conclusión final

En conclusión, este proyecto no solamente es una excelente guia y trabajo sobre cómo se debería crear un videojuego en unity 2D, sino que además es una prueba que muestra que la creación de videojuegos no solamente proporciona nuevos conocimientos sobre cómo programar, sino que además permite afianzar los conocimientos ya obtenidos y obtener un abanico nuevo de conocimientos.

Trabajo futuro

Por otra parte, se deja pendiente un conjunto de funcionalidades que podrían ser desarrolladas e implementadas en un futuro en este proyecto, se enumeran a continuación:

- Al morir, los enemigos vuelven a sus posiciones iniciales.

- Añadir sfx (sonidos de -efecto-).
- Los enemigos no deberían poder cruzar o estar dentro de un muro.
- Al superar el juego, deberían salir los créditos de éste.
- Podrían existir varios *slots* de guardado.
- Añadir variedad de enemigos.
- Añadir un sistema de combate a distancia.

Referencias

1. Wikipedia contributors. Perspectiva top-down. En: Wikipedia, The Free Encyclopedia [Internet]. Recuperado: https://es.wikipedia.org/w/index.php?title=Perspectiva_top-down&oldid=123013287

2. Descargar archivo, Beta P, Unity P y. R. Plataforma de desarrollo en tiempo real de Unity. En: Unity [Internet]. [citado 27 de mayo de 2023]. Recuperado: https://unity.com/es

3. Ahmad MO, Markkula J, Oivo M. Kanban in software development: A systematic literature review. 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. 2013. pp. 9-16. doi:10.1109/SEAA.2013.28

4. Wikipedia contributors. The Legend of Zelda (videojuego). En: Wikipedia, The Free Encyclopedia [Internet]. Recuperado: https://es.wikipedia.org/w/index.php?title=The_Legend_of_Zelda_(videojuego)&oldid=150296590

5. Wikipedia contributors. The Binding of Isaac. En: Wikipedia, The Free Encyclopedia [Internet]. Recuperado: https://es.wikipedia.org/w/index.php?title=The_Binding_of_Isaac&oldid=151028550

6. Ibrahim M, Ahmad R. Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques. 2010 Second International Conference on Computer Research and Development. 2010. pp. 200-204. doi:10.1109/ICCRD.2010.71

7. Fakhroutdinov K. UML Class and Object Diagrams Overview. 6 de agosto de 2011 [citado 27 de mayo de 2023]. Recuperado: https://www.uml-diagrams.org/class-diagrams-overview.html

8. ArMM. Zelda-like tilesets and sprites. En: OpenGameArt.org [Internet]. 16 de febrero de 2017 [citado 27 de mayo de 2023]. Recuperado: https://opengameart.org/content/zelda-like-tilesets-and-sprites

9. Top-down dungeon TileSet 2D by RedSteve. En: itch.io [Internet]. [citado 27 de mayo de 2023]. Recuperado: https://redsteve.itch.io/top-down-rouguelike-dungeon

10. Fantasy wooden GUI : Free. En: itch.io [Internet]. [citado 27 de mayo de 2023]. Recuperado: https://kanekizlf.itch.io/fantasy-wooden-gui-free

11. stockgiu, Recursos 88k. Mundo de juego de arcade y escena de píxeles. En: Freepik [Internet]. [citado 27 de mayo de 2023]. Recuperado: https://www.freepik.es/vector-gratis/mundo-juego-arcade-escena-pixeles_4815143.htm

12. O'Quinn C. 8 Bit Music - «Battle Man» (Mega Man Style Chiptune). Youtube; 5 de febrero de 2022 [citado 27 de mayo de 2023]. Recuperado: https://www.youtube.com/watch?v=zA1SwzvHsZw&list=PLtkjJsGOW8yPVOauP48WCyeq7KARWxD7r&index=1