MÁSTER UNIVERSITARIO EN DISEÑO Y PROGRAMACIÓN DE VIDEOJUEGOS

DISEÑO Y PROGRAMACIÓN DE UN VIDEOJUEGO CON ARQUITECTURA BASADA EN SCRIPTABLE OBJECTS

Tomás Gayo Perín

PROFESOR: Helio Tejedor Navarro

AULA: TFM - Programación Avanzada

Universitat Oberta de Catalunya

18 de Junio de 2023

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño y programación de un videojuego con arquitectura basada en Scriptable objects*
Nombre del autor:	*Tomás Gayo Perín*
Nombre del consultor/a:	*Helio Tejedor Navarro*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	06/2023
Titulación:	*Máster Universitario En Programación Y Diseño De Videojuegos*
Área del Trabajo Final:	*TFM - Programación Avanzada*
Idioma del trabajo:	*Castellano*
Palabras clave	*RPG, Scriptable Objects, Unity*

Resumen del Trabajo (máximo 250 palabras):

Este trabajo tiene como objetivo la creación de un videojuego en 3D con la ayuda del motor de videojuegos llamado Unity. El desarrollo se basará en la charla dada por Ryan Hipple en Austin sobre arquitectura de juegos con *Scriptable Objects* [1]. En ella se explican las bases para diseñar videojuegos a partir de datos, eventos y sistemas independientes, entre otras muchas cosas. A partir de aquí, se explorarán otros tipos de diseños de arquitectura y se investigará como adaptarlos a los *Scriptable Objects.*

El estilo de juego escogido para realizar este proyecto es el RPG, concretamente de acción en tiempo real con temática japonesa, similar a la saga Monster Hunter. Este tipo de juego encaja muy bien con el diseño modular basado en datos que se quiere alcanzar, ya que, se centra mucho en la interacción entre sistemas a base de eventos.

En un segundo plano, el trabajo se organizará mediante metodologías ágiles, específicamente, Scrum. Esto permite hacer un seguimiento más ordenado del trabajo realizado y, de esta manera, será posible gestionar mejor los tiempos.

En conclusión, aunque la meta es el videojuego, lo más relevante será la manera en cómo se construye para que, finalmente, el producto final acabe funcionando y pueda ser la portada de un portfolio para conseguir convencer y ser atractivo a primera vista.

Abstract (in English, 250 words or less):

The aim of this work is to create a 3D video game with the help of the game engine called Unity. The development will be based on a talk given by Ryan Hipple in Austin about game architecture with Scriptable Objects [1]. In this talk he explains the bases for designing videogames through data, events and modular systems, among other things. From here on, other types of architecture patterns will be explored and how to adapt them to Scriptable Objects will be investigated.

The game style chosen to make this project is the RPG, specifically, a real-time action with Japanese theme, something like the Monster Hunter saga. This kind of games fits very well with the modular pattern based on data that it wants to be achieved, since it focuses a lot on the event system interaction.

On the other hand, the work will be organized using agile methodologies, in this case, Scrum. This allows a more orderly follow-up of the work carried out and, in this way, it will be possible to manage the time properly.

In conclusion, although the game is the goal, the most important part of the project will be the way in which it is built, so that, finally, the final product ends up working and can be the cover of a portfolio to convince and be attractive at first sight.

Resumen

Este trabajo tiene como objetivo la creación de un videojuego en 3D con la ayuda del motor de videojuegos llamado Unity. El desarrollo se basará en la charla dada por Ryan Hipple en Austin sobre arquitectura de juegos con *Scriptable Objects* [1]. En ella se explican las bases para diseñar videojuegos a partir de datos, eventos y sistemas independientes, entre otras muchas cosas. A partir de aquí, se explorarán otros tipos de diseños de arquitectura y se investigará como adaptarlos a los *Scriptable Objects.*

El estilo de juego escogido para realizar este proyecto es el RPG, concretamente de acción en tiempo real con temática japonesa, similar a la saga Monster Hunter. Este tipo de juego encaja muy bien con el diseño modular basado en datos que se quiere alcanzar, ya que, se centra mucho en la interacción entre sistemas a base de eventos.

En un segundo plano, el trabajo se organizará mediante metodologías ágiles, específicamente, Scrum. Esto permite hacer un seguimiento más ordenado del trabajo realizado y, de esta manera, será posible gestionar mejor los tiempos.

En conclusión, aunque la meta es el videojuego, lo más relevante será la manera en cómo se construye para que, finalmente, el producto final acabe funcionando y pueda ser la portada de un portfolio para conseguir convencer y ser atractivo a primera vista.

Palabras clave: *RPG, Scriptable Objects, Unity*

Abstract

The aim of this work is to create a 3D video game with the help of the game engine called Unity. The development will be based on a talk given by Ryan Hipple in Austin about game architecture with Scriptable Objects [1]. In this talk he explains the bases for designing videogames through data, events and modular systems, among other things. From here on, other types of architecture patterns will be explored and how to adapt them to Scriptable Objects will be investigated.

The game style chosen to make this project is the RPG, specifically, a real-time action with Japanese theme, something like the Monster Hunter saga. This kind of games fits very well with the modular pattern based on data that it wants to be achieved, since it focuses a lot on the event system interaction.

On the other hand, the work will be organized using agile methodologies, in this case, Scrum. This allows a more orderly follow-up of the work carried out and, in this way, it will be possible to manage the time properly.

In conclusion, although the game is the goal, the most important part of the project will be the way in which it is built, so that, finally, the final product ends up working and can be the cover of a portfolio to convince and be attractive at first sight.

Keywords: *RPG, Scriptable Objects, Unity*

Índice

Figuras y tablas

Índice de figuras

Índice de tablas

1. Introducción

1.1. Contexto y justificación

Cuando jugamos a un videojuego, por norma general, lo primero que tenemos en mente son las mecánicas, el diseño de niveles, la IA de los enemigos, etc. Resulta evidente puesto que es lo más visual y, por lo tanto, lo primera que entra a la vista. En paralelo a esto, en el mundo del desarrollo, cuando eres principiante las primeras cosas que uno quiere aprender son exactamente esas, las más vistosas porque es lo que siempre se ha visto por pantalla.

Este trabajo se va a enfocar desde otra perspectiva donde lo más importante no será lo visual, sino lo que hay detrás de esos sistemas y componentes, es decir, se va a construir una arquitectura que haga que el juego funcione en sintonía entre todas sus partes.

Esta necesidad y curiosidad de aprender cómo funcionan los videojuegos es el motivo principal de este proyecto, así pues, en los siguientes apartados se describirán las bases y estructura del resto de documento.

"For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it. I can solve a task with just a few choice function calls that slot in perfectly, leaving not the slightest ripple on the placid surface of the code." (Nystrom, 2014, p. 10) [2]

1.2. Definición

En esta sección se describirá la propuesta del TFM y se matizarán los puntos más importantes.

1.2.1. Punto de partida

Cuando un estudiante empieza en el mundo de la programación, en este caso, la de videojuegos, es muy común que todo el conocimiento que adquiera sea por información muy dispar encontrada en Internet, sin embargo, esa gran cantidad puede llegar a ser confusa o muy simplificada y, en consecuencia, consigue el efecto contrario sobre la persona que la consume. La razón principal para realizar este proyecto es justamente la de aclarar todas esas ideas que han ido surgiendo durante todo este aprendizaje.

1.2.2. Propuesta

Este trabajo tiene como objetivo la creación de un videojuego en 3D con la ayuda de un motor de videojuegos. Su desarrollo se planificará de dentro hacia afuera, es decir, se trabajará desde el punto de vista arquitectónico intentando proporcionar un código bien estructurado.

La arquitectura que se va a trabajar en este proyecto es aquella que estudia las partes más pequeñas de una aplicación y como interactúan entre ellas. Más específicamente, se está haciendo referencia a las clases, componentes y paquetes que puede contener un proyecto de manera más interna.

Todo esto se va a conseguir aplicando las buenas prácticas y los patrones de diseño que han sido descritas en multitud de ocasiones a lo largo de la historia de la computación. No obstante, todo

esto se aplicará al entorno de los videojuegos, por lo tanto, el objetivo es doble puesto que se estudiará desde una perspectiva más general dentro del *software, pero* se deberá aplicar en un videojuego.

La herramienta que se utilizará en el desarrollo es el motor de videojuegos Unity. Esta herramienta permite el uso de uno de los puntos clave de este proyecto que son los *Scriptable Objects* que permitirán hacer uso tanto de las buenas prácticas como de los patrones de diseño. A partir de aquí, se definirán los usos que pueden tener en el proyecto y como aplicarlos a una arquitectura de videojuegos.

En definitiva, se propone diseñar un videojuego desde el punto de vista más estructural del código, haciendo más énfasis en la manera en cómo se consiguen los resultados, más que en los resultados mismos.

1.2.3. Descripción del videojuego

El tipo de juego escogido es el ARPG o, dicho de otra manera, un juego de rol de acción. Será en 3D y la estética que se quiere buscar es de estilo japonés. La inspiración directa será la serie de videojuegos Monster Hunter (MH). De ahí, se seleccionarán las mecánicas más básicas y se harán funcionar en este trabajo simplificando su funcionamiento. Las características principales que se quieren añadir son: un sistema de combate, un inventario y un sistema de misiones. Más adelante se especificarán con detalle cada una de las partes que componen este videojuego.

1.3. Objetivos generales

Sintetizando todas las ideas y explicaciones vistas hasta ahora, los objetivos se pueden dividir en los siguientes puntos.

1.3.1. Objetivos principales

- Crear una arquitectura diseñada para un videojuego.
- Aplicar buenas prácticas al proceso de trabajo a partir de los principios SOLID y patrones de diseño.
- Aprender el uso de los *Scriptable Objects* y demostrar su viabilidad.
- Diseñar los patrones de diseño a partir de *Scriptable Objects*.

1.3.2. Objetivos secundarios

- Crear un videojuego con unos objetivos claros y que tenga flujo cerrado con principio y final.
- Trabajar con metodologías ágiles para organizar el trabajo.

1.4. Planificación

1.4.1. Alcance y riesgos del trabajo

La creación de este videojuego está sujeta a un tiempo limitado y no puede ser extendido debido a su naturaleza. Es un proceso de, aproximadamente, 4 meses de desarrollo y se lleva a cabo de forma casi integra por una sola persona.

Los riesgos que pueden darse a relucir son varios y, por este motivo, es necesario identificarlos y encontrar la mejorar manera de mitigarlos. Para ello se utilizará la siguiente matriz de riesgos que muestra cuan probable es que ocurra algo y que impacto puede tener sobre el proyecto.

Figura 1: Matriz de riesgos

Los riesgos son varios y la manera de reducir su impacto dependerá en cada momento de la situación en la que se encuentre el proyecto. El impacto de los riesgos indica cuanto control hay sobre ellos, si son muy graves tenemos poco control, por otro lado, la probabilidad, indica su frecuencia. En los siguientes puntos se describen uno a uno, por orden de prioridad, y la manera en cómo se encararán:

- **Tiempo**:

  El tiempo es un factor que siempre está presente y marcar bien el ritmo de la práctica será de vital importancia. Se hará un control periódico donde se planificará el trabajo teniendo en cuenta las tareas realizadas y las que faltan por realizar y, de esta manera, determinar si se está yendo por el buen camino o, en caso contrario, se buscarán mejoras para encaminarlo de nuevo

- **Imprevistos técnicos**:

Como en cualquier equipo de desarrollo, los imprevistos son comunes y poco bienvenidos. Saber cómo evitarlos no es tarea sencilla y encontrarse con ellos no se puede prever dado que, como su propio nombre indica, son imprevistos. Entonces, la forma de actuar en este riesgo tiene dos variantes: la primera consiste en tener claras las ideas desde un principio, siempre antes de comenzar el desarrollo será necesario construir un plan y analizarlo, es decir, la solución consiste en construir un planteamiento para evitar errores; si de todas maneras no se consigue evitar, entonces, se tratará de comprender el problema y buscar la mejor solución para evitar retrasos, asimismo, habrá que documentar el fallo para poder evitar otro error más adelante. Por lo tanto, el plan consiste en evitar los imprevistos o solucionarlos. Evidentemente, el primer caso lo podemos controlar y es el más deseable.

- **Aumento de alcance**:

  Cuando hay un aumento de alcance, significa que durante el proceso de planteamiento no se tuvieron en cuenta todos los factores y hay que añadir más trabajo a las tareas iniciales. Este riesgo no se puede solucionar suponiendo que no habrá un aumento, sino todo lo contrario. Teniendo en cuenta esto, se dejará un margen de error temporal para poder manejar estas situaciones. Es decir, si el desarrollo dura 4 meses, las últimas 2 o 3 semanas se dejarán disponibles para hacer trabajo extra.

- **Recursos limitados**:

  Tanto el personal, el tiempo, como las herramientas que se van a utilizar en este trabajo, son limitadas. Limitadas por cantidad, presupuesto y experiencia y lidiar con ello será todo un proceso de aprendizaje. La mejor manera de mitigarlo es un buen planteamiento inicial con un buen control de tiempo y una buena elección de las herramientas.

- **Falta de claridad**:

  De los fallos más comunes dentro de un equipo de desarrollo se encuentra la falta de claridad. Si los requerimientos no están claros en el momento de desarrollar algo, el trabajo será más complejo y rebuscado. Hay que mantener un sistema que permita el análisis y entendimiento de todas las partes y en caso de duda repetir el proceso.

  Dado que en este caso el equipo lo forma una sola persona es mucho más relevante este punto, puesto que no se depende de nadie y muchas veces se dan las cosas por hecho. Mantener una documentación adecuada y controlada será de mucha importancia para evitar este riesgo.

- **Costes elevados**:

  Posiblemente el riesgo menos relevante comparado con los anteriores. El problema que puede surgir es no encontrar el dinero para conseguir un recurso importante para el proyecto, no obstante, siempre existen alternativas gratuitas, que son menos atractivas, pero pueden llegar a ser igual de efectivas para este contexto.

1.4.2 Método y Estrategia

La metodología escogida para este proyecto es la de Scrum, un método ágil que permite organizar el proyecto en diferentes iteraciones (s*prints*) de 2 a 4 semanas cada una. Cada iteración es un bloque en el cual se asignan unas tareas y se siguen unas dinámicas establecidas, estas son las diferentes ceremonias que ayudan al equipo de trabajo a cumplir el objetivo.

Es un método que permita mucha flexibilidad a diferencia de los métodos tradicionales. No hace falta asignar todas las tareas desde la planificación inicial (Waterfall) y se pueden ir desarrollando a medida que avanza el proyecto, ahora bien, la idea no puede ir cambiando constantemente. Por otro lado, el Scrum es más estricto en cuanto a tiempos de entrega si lo comparamos con otras metodologías ágiles como el Kanban, que es más conveniente para equipos que no tienen entregas tan limitadas. Por lo tanto, el Scrum está en un punto intermedio que ofrece la libertad suficiente para realizar cambios cuando el proyecto está avanzado y, además, los tiempos están muy marcados.

En el siguiente diagrama se puede observar el proceso típico que se utiliza en Scrum. Lo más destacable para este proyecto son: las *User Stories* (USs), donde se escribirán los requisitos de cada actividad; el *Backlog*, que guarda las USs (u otras tareas como *bugs*) hasta el momento de su desarrollo; *Sprints*, es la iteración con duración determinada que marca el inicio y final de un ciclo; también, mencionar la *Planning*, que es una ceremonia o reunión donde se decide que actividades deben entrar en un *sprint* y se realiza al principio del ciclo; la Demo, una reunión donde se expone el trabajo realizado al final de un *sprint* y; por último, la *Retrospective*, otra reunión donde se hacen comentarios de los puntos que han ido bien y los que han ido mal para llegar a unas conclusiones que pretenden mejorar el proceso de desarrollo y, por lo tanto, es una ceremonia que se hace al final de una iteración. Hay mucho más detrás de esta metodología, no obstante, para simplificar se utilizará solo lo comentado en este apartado.

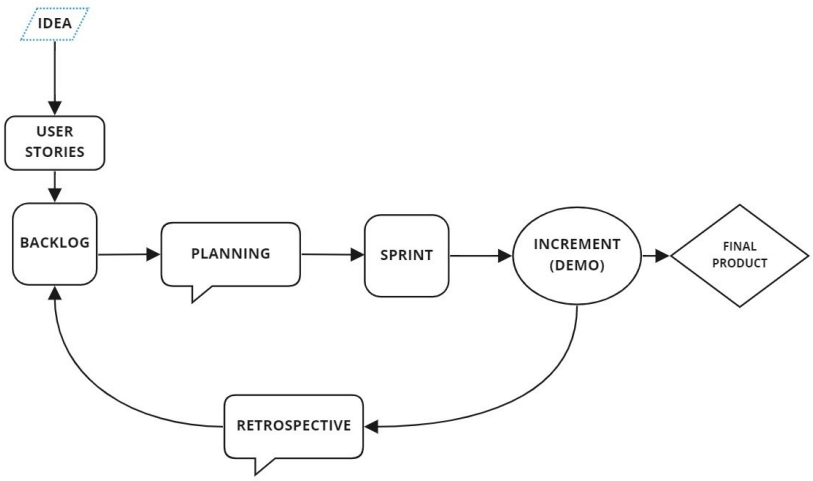


Figura 2: Flujo de Scrum

En cuanto a la herramienta para llevar a cabo el proceso de Scrum se ha escogido Jira. Es una herramienta que se utiliza mucho en el ámbito laboral y tiene una versión gratuita con todo lo necesario para aplicar la metodología que se prefiera.

En Jira se tendrán en cuenta tres espacios diferentes: la hoja de ruta, el *backlog* y el tablero. El primer espacio, será imprescindible para la planificación general, ahí se organizarán las actividades por épicas y cada una tendrá un objetivo diferente, es más, la hoja de ruta es, en realidad, un calendario y se pueden visualizar los *sprints* y las épicas en el tiempo; seguidamente, el *backlog*, que ya se ha hecho referencia en el apartado anterior, muestra todas las tareas existentes y permite asignarlas a los *sprints* que se consideren. Esta asignación se hará al final de un sprint para planificar el siguiente o se puede plantear desde el principio del proyecto, pero eso no significa que sea la asignación definitiva; en último lugar, está el tablero, que es el espacio que más se utilizará puesto que es donde se trabaja diariamente. Ahí se visualizarán las tareas que hay por hacer, las que están en proceso de hacerse y las que se han finalizado, pero, solo aparecen las tareas del ciclo actual.

Además de esto, este *software* permite la creación de las *User Stories* (y otros tipos de tareas o *tickets*) que serán necesarias para saber el trabajo a realizar. El contenido que hay que añadir dentro de estos *tickets* es: una descripción con los resultados que se quieren obtener, unos requisitos con las tareas que hay que hacer y unas instrucciones que especifican las pruebas que permitirán decidir si la US está completa o no. Asimismo, en cada tarea se especificarán los llamados *story points* (SPs), que indican el esfuerzo general que requiere la tarea para llevarse a cabo.

Al final de cada iteración se calcularán los SPs de todas las tareas completadas, la suma de ese valor indicará una aproximación de las tareas que se pueden asumir para el siguiente *sprint*. No obstante, no hay que confundir estos puntos con el tiempo, puesto que es complicado medir cuantas horas exactas se va a tardar en realizar un trabajo. Por lo tanto, es más una sensación del esfuerzo que requiere completar una US y está más asociado a la cantidad de trabajo que hay que hacer, el total de personas que hay que implicar o la complejidad de los requisitos.

1.4.3. Fases del proyecto

El proyecto se dividirá en diferentes fases que determinarán los hitos y el alcance en cada momento para mantener un control más estable de este. Cada fase se divide en varios *sprints* con una duración determinada de dos semanas cada uno.

- **Fase 0**: en esta fase se inicia el proyecto y es el momento de planificar, diseñar y analizar el trabajo que se realizará durante las siguientes semanas.
- **Fase 1**: aquí comienza el desarrollo y se asignan dos *sprints* (4 semanas) con el objetivo de crear un prototipo con los sistemas y mecánicas básicas que más adelante serán necesarias. Este prototipo será el primer reto que determinará el rumbo del proyecto.

- **Fase 2**: durante esta fase el desarrollo continúa siguiendo el objetivo de completar el juego y tener una versión final. Es la fase más importante puesto que es la más larga (4 *sprints*) y contiene los puntos más críticos del desarrollo. Será vital dar un *feedback* continuo para calcular bien los tiempos de trabajo.
- **Fase 3**: etapa de duración corta (1 *sprint*) dedicada a hacer pequeñas mejoras o añadidos en caso de que el alcance haya aumentado o queden algunas tareas por hacer. Es ese margen que se menciona en los riesgos y que no se tendrá en cuenta a la hora de asignar tareas en la fase inicial.
- **Fase 4**: con la etapa de implementación acabada solo queda documentar y presentar el proyecto.

1.4.4. Contenido de los *Sprints*

Definir el contenido de cada iteración sería un error por cómo está concebido el sistema de Scrum, ahora bien, se puede planificar cual es el objetivo o alcance de cada uno de forma general.

- ***Sprint* 1**: Creación y configuración del proyecto en Unity, instalación de recursos básicos, creación de escenario de pruebas, activación de sistemas básicos y primer control del personaje. PEC 1.
- ***Sprint* 2**: Plantear sistema y mecánicas de combate del protagonista y enemigos. Entrega PEC 1.
- ***Sprint* 3**: Aplicar GameManager y estados del personaje. Introducir la lógica de cambio de escena y SpawnSystem. PEC 2.
- ***Sprint* 4**: Activación de sistemas más complejos como misiones e inventario. Entrega PEC 2.
- ***Sprint* 5**: Construcción de escenarios. Música y sonidos de ambiente. PEC 3.
- ***Sprint* 6**: Menú de juego, opciones, pantalla de pausa. Versión Final. Entrega PEC 3.
- ***Sprint* 7**: Mejoras y últimos retoques.
- ***Sprint* 8**: Documentar el proyecto. Entrega PEC 4.

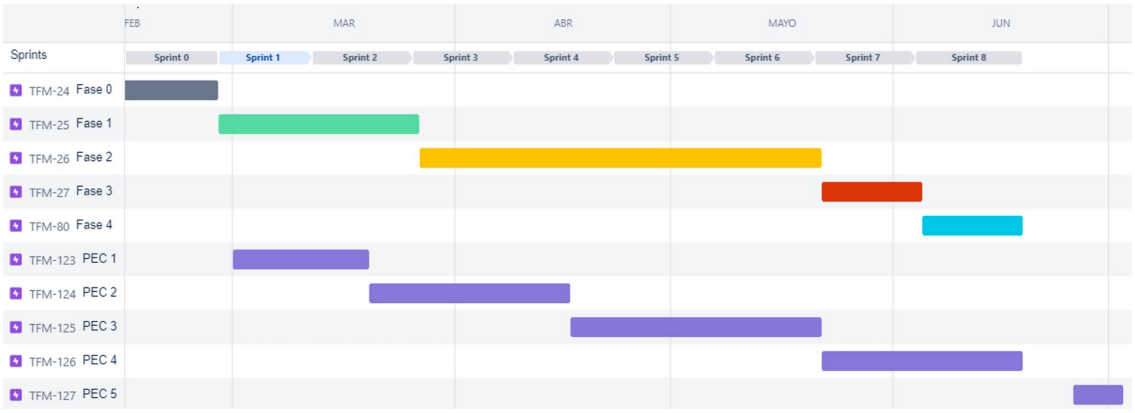


Figura 3: Hoja de ruta

1.4.5. Presupuesto

El siguiente presupuesto es una estimación teniendo en cuenta que este proyecto comienza desde cero, el equipo se conforma por una sola persona y su duración es de aproximadamente 4 meses. Además, se ha tenido en cuenta un gasto extra por si ocurre algún contratiempo.

Tabla 1: Presupuesto material

Estimación del presupuesto: Material		
Ítem	**Descripción**	**Gasto Único (€)**
PC	Intel Core i7 con RTX 3060	1.602,87
Ratón	Tecknet	11,89
Teclado	Redragon	45,99
Gamepad	Switch Controller Pro	69,99
Auriculares		20
Silla		150
Mesa		100
Otros (5% del total en riesgos)		100,04
TOTAL		**2.100,78**

Tabla 2: Presupuesto herramientas y recursos digitales

Estimación del presupuesto: Herramientas y recursos digitales		
Ítem	**Descripción**	**Gasto Único (€)**
Unity	Plataforma de desarrollo	0
Microsoft VS	Editor de código	0
Recursos Asset Store	Recursos de Unity	0
Recursos Asset Store	Recursos externos a Unity	500
Recursos ofimáticos	Word, Excel, Jira, Miro ...	100
Otros (5% del total en riesgos)		30

TOTAL		**630,00**

El salario se calcula teniendo en cuenta un sueldo aproximado de 24.000€ y 40 horas semanales de trabajo. Asimismo, se ha dividido el dinero por etapas para visualizar mejor la cantidad en cada momento del proyecto.

Tabla 3: Presupuesto gastos de personal

Estimación del presupuesto: Gastos de Personal			
Etapas	**Horas**	**Precio (€)**	**Gasto Total (€)**
Planificación	40	8,5	340
Diseño y Análisis	40	8,5	340
Implementación	560	8,5	4.760
Finalización	40	8,5	340
Monitorización y Control	640	4	2.560
Otros (10% del total en riesgos)			834
TOTAL			**9.174,00**

Haciendo el cálculo total, el presupuesto estimado del proyecto es de 14.634,78 €.

Tabla 4: Presupuesto total

Estimación Total	
Estimación	**Gasto Total**
Material	2.100,78
Herramientas y recursos digitales	630,00
Gastos de personal	9.174,00
Gastos mensuales	2.730,00
TOTAL	**14.634,78**

1.5. Estructura del resto del documento

Los siguientes capítulos seguirán la misma estructura que se ha establecido en el planteamiento, de esta manera se mantiene un orden y un sentido que ayudará a entender mejor la memoria en relación con el trabajo realizado.

- **Capítulo 2. Estado del arte**: el siguiente capítulo se dedicará al estado del arte donde se contextualizará el trabajo y se indagará en diferentes aspectos que se utilizarán dentro de este.
- **Capítulo 3. Diseño y Análisis**: la mayor parte de esta sección se utilizará para definir el trabajo a realizar. Esto incluye: la explicación sobre las mecánicas del juego, el

funcionamiento de los diferentes sistemas, la arquitectura y su funcionamiento y el qué, cómo y cuándo se deberá probar.

- **Capítulo 4. Implementación**: este capítulo documentará el progreso por *sprints*, explicando punto por punto el trabajo y los resultados finales por cada fase.
- **Capítulo 5. Conclusiones y resultados**: aquí se mostrará el resultado global definitivo y se hará una valoración final con los puntos conseguidos y líneas de futuro. Este apartado funcionará como unas conclusiones.

2. Estado del arte

2.1. Contexto

En la ingeniera de *software* siempre se han tenido muy presentes las buenas prácticas y el saber mantener un código ordenado y limpio. Personas destacadas como Robert C. Martin con su ensayo "*Design Principles and Design Patterns*" [3] o los llamados *Gang of Four* con su libro "*Design Patterns*" [4] son posiblemente los más destacados.

Robert comentaba en su escrito que en muchas ocasiones se encontraba con proyectos donde, a pesar de que en un principio todo funcionaba correctamente, a medida que el producto crecía se iba deteriorando poco a poco hasta llegar a un punto donde era muy difícil mantenerlo. Adicionalmente, auguraba que las causas de estos problemas son la falta de anticipación y las consecuentes dependencias que se generan internamente.

A partir de aquí, lo más relevante de su ensayo consiste en explicar unos principios que tratan de establecer unas "reglas" para conseguir evitar todos estos inconvenientes. Estas normas más adelante se convertirían en los llamados principios SOLID que son de sobra conocidos por cualquier programador experimentado.

Por otro lado, *The Gang of Four* (GoF) es el nombre que reciben los cuatro autores del libro "*Design Patterns*" que se publicó en 1994. En este texto se recopilan los patrones más comunes que hasta esa fecha no estaban documentados. Partiendo de ese punto, se proporcionó a cada uno de un nombre, una explicación y ejemplos reales que daban visibilidad a las posibles utilidades. Es un libro anterior a los escritos de Robert, sin embargo, siguen la misma filosofía utilizando otras palabras. Para destacar, se hace mención sobre la anticipación, la simplicidad y se resalta la reutilización de código.

Todo esto fue creado hace ya más de dos décadas y en un sector tan cambiante como es el mundo de la computación, surgen algunas dudas, ¿Siguen aplicando estos conocimientos hoy en día?, ¿Han surgido nuevas técnicas o diseños que sustituyan a los ya existentes? La respuesta rápida es que sí, todos esos conceptos se siguen utilizando y, a pesar de que no se ha reinventado nada, siempre se pueden encontrar nuevas variaciones de lo ya existente.

2.2. Principios SOLID

SOLID es un acrónimo creado con la letra inicial de cada principio y se pensó así para facilitar el aprendizaje de estos.

2.2.1. Principio de responsabilidad única (*Single Responsability Principle)*

Cada clase debería tener una única responsabilidad evitando sobrecargarla de comportamientos muy distintos entre sí. Cuanto más individuales e independientes sean las clases, más control se tendrá sobre ellas y, por lo tanto, se consigue una mejor lectura de los componentes, tanto a nivel especifico de clase como el conjunto de estas. En consecuencia, en caso de error la detección de este será directa porque las clases tienen una única responsabilidad.

2.2.2. Principio de apertura y cierre (*Open Closed Principle)*

Las clases deberían estar abiertas a extensión pero cerradas a modificación. Es decir, la idea es que se pueda cambiar lo que hace una clase sin cambiar el código de esas clases. Básicamente, lo que dice este principio es que hay que utilizar técnicas como las interfaces o la abstracción de clases que permiten justamente extender el comportamiento de una clase sin tocar la clase original.

2.2.3. Principio de sustitución de Liskov (*Liskov's substitution Principle)*

Este principio indica que cualquier subclase debe poder ser substituida por su clase base, en otras palabras, si una clase B es hija de una clase A, cualquier método que requiera de un objeto de la clase A, podrá ser substituido por la clase B. Dado que la clase B hereda todo el contenido de la clase A, se entiende que tienen el mismo comportamiento y, por ese motivo, puede procesar cualquier cosa que se le pida a la clase base.

Si esto no se cumpliera significaría que la clase B a cambiado su comportamiento por completo y podría generar errores muy difíciles de detectar. Al final, el objetivo es conseguir un código consistente para evitar errores.

2.2.4. Principio de segregación de interfaces (I*nterface Segregation Principle)*

La segregación de interfaces está muy relacionada con el principio de responsabilidad única, ya que, es básicamente separar las interfaces en unidades pequeñas para que quien haga uso de ellas no tenga que implementar métodos que después no vaya a utilizar. El objetivo es reducir ruido en las clases con métodos vacíos y mejor la lectura de código.

2.2.5. Principio de inversión de dependencia (D*ependency Inversion Principle)*

En este caso, este principio se relaciona directamente con el de apertura y cierre. En pocas palabras se dice que las clases deben depender de las interfaces o la abstracción y no la implementación. Por ejemplo, si una clase contiene la lógica de una herramienta concreta y otra la lógica para accionar esa herramienta ambas clases no deberían saber una de la otra para conseguir su objetivo. Quien activa la herramienta no le debería interesar como funciona internamente esa herramienta, ni la herramienta debe conocer quien la activa y, todo esto, se

consigue a partir de la abstracción o interfaces. El objetivo, finalmente, es reducir dependencias innecesarias.

2.3. Patrones de diseño

Seguidamente se describirán algunos de los patrones de diseño más populares para el desarrollo de videojuegos, se verá algún ejemplo y se destacarán sus ventajas y desventajas:

- **Singleton**: este patrón es uno de los más utilizados, pero a su vez, uno de los más controvertidos y que muchos intentan evitar. Se puede definir como un patrón que crea una única instancia de una clase para todo el sistema y que, además, permite un acceso global desde el resto de clases.

  Hay muchos usos que se pueden aplicar con este patrón, por ejemplo, uno de los más típicos es crear un *Game Manager* que contiene la lógica más general del juego y, después, el resto de clases pueden acceder fácilmente. Aplicado a Unity este patrón funcionaría de la siguiente manera:

GameManager.cs

```
1  public class GameManager : MonoBehaviour
2  {
3      private static GameManager instance;
4
5      public static GameManager Instance
6      {
7          get
8          {
9              if (instance == null)
10             {
11                 instance = new GameManager();
12             }
13             return instance;
14         }
15     }
```

Figura 4: Patrón Singleton en un MonoBehaviour

La variable estática *instance* contiene la instancia en sí y la propiedad *Instance* se asegurará que solo haya una única instancia que será el punto de acceso. Con esto ya está el patrón establecido, solo falta añadir lógica en el resto del *script* y acceder desde donde se necesite.

Las ventajas más evidentes se han comentado al principio: se crea una única instancia y es de acceso global, sin embargo, el hecho de permitir que se pueda acceder desde cualquier lugar provoca que se haga un mal uso de este diseño, puesto que existe la tendencia a sobrecargar estas clases con variables y métodos muy dispares dificultando así su lectura. Asimismo, se crea una dependencia de esta instancia que siempre es recomendable evitar, ya que no permite generar componentes aislados y dificulta en gran medida las pruebas de estos.

En conclusión, este patrón es recomendable utilizarlo siempre y cuando se trabajé con casos muy específicos. Su simplicidad lo hace muy popular pero hay que saber lo que se hace y quizá otros patrones sean más convenientes.

- **State**: el objetivo de este patrón es que un mismo objeto pueda tener diferentes comportamientos simplemente modificando el estado sin tener que cambiar de objeto. Se basa en el modelo de comportamiento de máquina de estado o FSM (*finite state machine*), que se define como un conjunto de estados donde solo un estado está activo y la salida depende únicamente del propio estado y sus entradas.

  Hay varias maneras de implementarlo, aquí se puede ver el diagrama de una implementación sencilla:

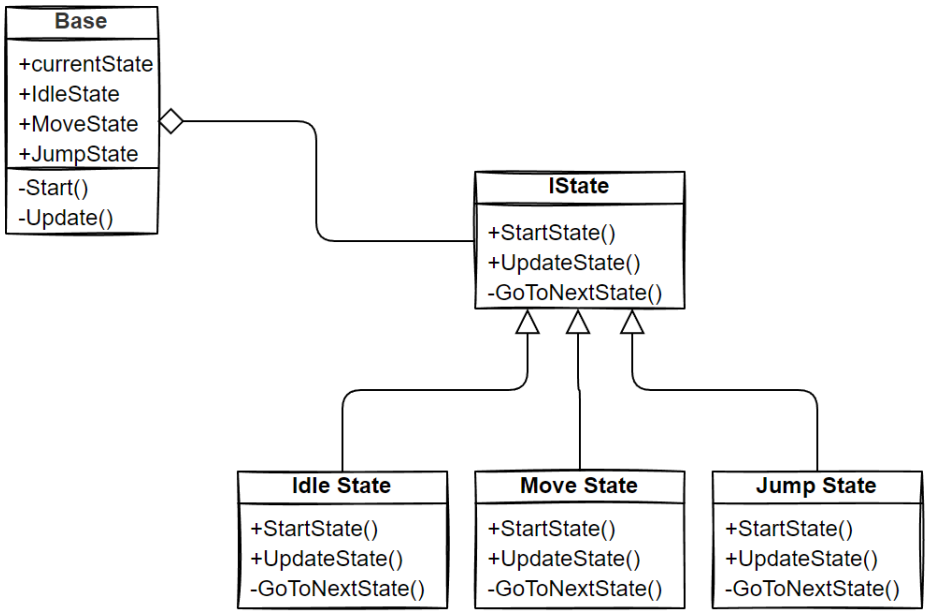


Figura 5: Diagrama de clases del patrón State

Básicamente, cada estado es una clase que implementa una interfaz, de esta manera cada estado funciona igual, pero con comportamiento distintos. Después, hay una clase base que será la que se incluya en el objeto donde se quiere utilizar la FSM. Esta clase base gestionará los estados especificando el estado inicial y el estado actual y, también, accionará los métodos de cada estado en el momento que corresponda.

En el siguiente diagrama se ha creado un flujo de tres estados muy simple simulando el control de un personaje que puede moverse o saltar.

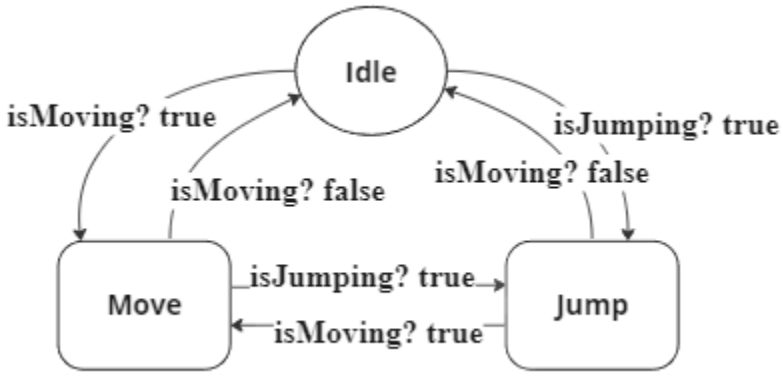


Figura 6: Ejemplo de diagrama de estados

En la clase base se llamará a un estado inicial, en este caso *Idle* y, dentro de cada estado, se aplicará el comportamiento deseado, en el caso de ejemplo, en cada estado hay acciones de entrada (*Start*) y acciones que se ejecutarán en cada *frame* (*Update*), también, en cada *frame* se comprobará si se cumplen ciertas condiciones para moverse al siguiente estado: *Move* o *Jump.*

Su uso más típico en videojuegos es el de la IA porque se permite el control autónomo en tiempo de ejecución, pero hay otros usos como la gestión de animaciones o control del estado del personaje. Este patrón es útil siempre y cuando se tenga en cuenta que solo puede existir un estado activo y que solo depende de la entrada y el estado actual, además, se destaca su escalabilidad puesto que todo el código esta dividido en pequeñas clases que cumplen con una única función, por esta razón, sus estados y decisiones tienen alta probabilidad de ser reutilizados en otras FSMs.

Es un patrón que cumple muy bien su función y si se aplica correctamente es muy recomendable, no obstante, en ocasiones otros modelos de comportamiento serán más óptimos según la situación. En IA cuando se quiere algo más complejo el FSM es más rígido porque solo se puede lanzar un estado a la vez, si se quisiera aplicar dos comportamientos distintos al mismo objetos, se necesitarían dos máquinas al mismo tiempo con lo que comportaría el doble de trabajo y complejidad. En cambio, se suele optar por modelos como el *Behaviour Trees* que es mucho más flexible y permite realizar comportamientos distintos en paralelo. Esto es solo una valoración que se pueden tener en cuenta a la hora de elegir el mejor método, pero eso no resta valor a al patrón de State que según como se aplica tiene mucho potencial.

- **Observer**: este patrón es de los más conocidos y utilizados gracias a sus beneficios. De forma muy simplificada se puede decir que crea un canal de comunicación entre una clase (*Subject*) que quiere enviar un mensaje, y otras clases (*Observer*) que estarán escuchando y actuarán en consecuencia.

Es muy útil por la independencia entre el observador y el sujeto, sencillamente, ninguno sabe de la existencia del otro y esto ayuda a simplificar las clases puesto que se puede separar la lógica de los elementos de la interfaz, entre otras muchas cosas.

En el siguiente ejemplo un sistema de vida requiere de una interfaz para mostrar al jugador los puntos de vida. El sujeto (*HealthSystem*) contendrá toda la lógica necesaria para restar vida al personaje, por otro lado, el observador (*HealthBarUI*), sencillamente, estará a la espera de algún cambio, si lo recibe se refrescará la UI.

HealthSystem.cs

```cs
using UnityEngine;
using UnityEngine.Events;

public class HealthSystem : MonoBehaviour
{

    [SerializeField] private int baseHealth;

    public event UnityAction<int> OnHealthChange = delegate { };

    int currentHealth;

    private void Start()
    {
        currentHealth = baseHealth;
        OnHealthChange?.Invoke(currentHealth);
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        OnHealthChange?.Invoke(currentHealth);
    }
```

Figura 7: Clase que funcionará de Subject que contiene la lógica

Primero hay que crear el *Subject* que se encargará de crear un evento o canal que se deberá invocar cuando se quiera notificar al observador.

HealthBarUI.cs

```
1 using UnityEngine;
2 using UnityEngine.UI;
3
4 public class HealthBarUI : MonoBehaviour
5 {
6     public Text healthText;
7     public HealthSystem healthSystem;
8
9     private void OnEnable()
10    {
11        healthSystem.OnHealthChange += RefreshHealthBar;
12    }
13
14    private void OnDisable()
15    {
16        healthSystem.OnHealthChange -= RefreshHealthBar;
17    }
18
19
20    public void RefreshHealthBar(int newValue)
21    {
22        healthText.text = newValue.ToString();
23    }
24 }
```

Figura 8: Clase Observer que escuchará a un canal

Seguidamente el *Observer* se debe suscribir a ese evento cuando se activa el *script* y eliminar la suscripción cuando se desactiva. Esta suscripción hace referencia al método que se quiera utilizar tras recibir una notificación y cuando es recibida se activa esa parte del código. Con esto se ha conseguido dividir la lógica del sistema de vida de los elementos de la UI, evitando dependencias innecesarias.

- **Object Pool**: el siguiente patrón a diferencia de los anteriores tiene un objetivo bastante especifico que es el de mejorar el rendimiento evitando la fragmentación en memoria. En pocas palabras, se guarda un espacio en memoria al inicio (creación del *pool*) con los objetos necesarios y se desactivan, entonces, a medida que se requieran se activaran hasta que vuelvan a desactivarse de nuevo. Con esto se evita instanciarlos y destruirlos frecuentemente ya que son operaciones muy pesadas y, además, se evita que la memoria quede fragmentada dejando espacios desaprovechados.

  En videojuegos es muy común el uso de este patrón porque constantemente se instancian objetos que requieren ser destruidos al poco tiempo de ser puestos en escena, por ejemplo, las balas de un arma tienen que crearse e inmediatamente, cuando golpean con otro objeto o llegan a una distancia especifica, se tienen que eliminar. Esto es un caso claro donde utilizar este patrón que activará las balas y las desactivará sin la necesidad de destruirlas.

  Una solución que proporciona Unity es la siguiente:

ObjectPool.cs

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

public class ObjectPool : MonoBehaviour
{
    public static ObjectPool SharedInstance;
    public List<GameObject> pooledObjects;
    public GameObject objectToPool;
    public int amountToPool;

    private void Awake() => SharedInstance = this;

    private void Start()
    {
        pooledObjects = new List<GameObject>();
        GameObject tmp;
        for(int i = 0; i < amountToPool; i++)
        {
            tmp = Instantiate(objectToPool);
            tmp.SetActive(false);
            pooledObjects.Add(tmp);
        }
    }

    public GameObject GetPooledObject()
    {
        for (int i = 0; i < amountToPool; i++)
        {
            if (!pooledObjects[i].activeInHierarchy)
            {
                return pooledObjects[i];
            }
        }
        return null;
    }
}

```

Figura 9: Creación de un Object Pool en Unity [7]

Esta clase instanciará al principio de partida una cantidad de objetos dentro de una lista que se proporcionará en el inspector junto al objeto que se quiere activar. Estos objetos comenzarán desactivados y mediante un método público otras clases podrán pedir activar el siguiente objeto de la lista que esté desactivado. Es importante que se desactive de nuevo una vez se cumpla con su función.

2.4. Arquitectura en Unity

En la industria de los videojuegos estos principios, técnicas y patrones aplican exactamente de la misma manera, ahora bien, existen diferencias que dan un enfoque algo distinto respecto a otros productos. Hoy en día, los motores de videojuegos suponen un cambio respecto a otras herramientas que obligan a adaptar los razonamientos que explicaban tanto Robert como el GoF.

Antes de comenzar con la materia es necesario entender las bases y el funcionamiento de Unity. Más adelante, se hará uso de los términos que se introducirán en esta sección.

2.4.1. Ventanas

En general, los motores de videojuegos están pensados para que no solo los desarrolladores sean capaces de entenderlo, por lo que, todo es muy visual. La entrada principal está en sus ventanas de trabajo que son:

- **Proyecto**: ventana donde se guardan todos los archivos de proyectos (*scripts*, audio, texturas, etc.) o, también, llamados *assets*.
- **Escena**: aquí es donde se manipulan todos los objetos del juego en cuanto a su posicionamiento.
- **Jerarquía**: en la jerarquía aparecen todos los objetos (*GameObjects*) que se han colocado en la escena en forma de jerarquía, por lo tanto, entre *GameObjects* existe la relación padre e hijo.
- **Inspector**: la ventana de inspector permite visualizar y configurar los objetos que hay en escena o los *assets* del proyecto.
- **Vista de juego**: finalmente, todo confluye en esta ventana que es la representación de lo que hay en escena, no obstante, la vista estará limitada por una cámara que se debe configurar en el inspector para renderizar la imagen como mejor se adapte al juego.

2.4.2. *GameObjects*

El concepto más básico que hay que entender para cualquier persona que trabaje en Unity son los GameObjects, cada objeto que exista en escena dentro del juego será uno de estos. Los objetos por definición son cascaras vacías que necesitan definirse a partir de los llamados componentes y, por defecto, solo contienen información de su posición y orientación (componente transformada) en escena.

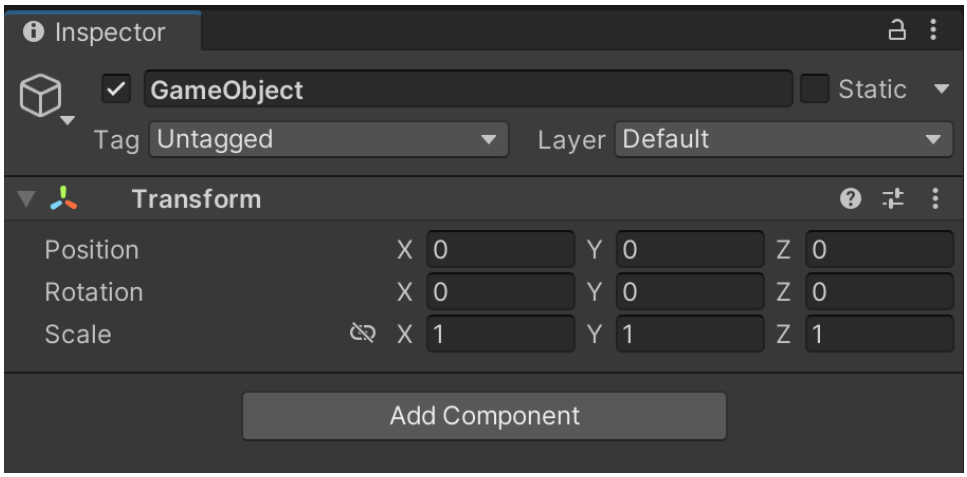


Figura 10: Visualización de un GameObject sin componentes en el inspector

Cada objeto puede contener infinitos componentes y cada componente describirá una característica de ese *GameObject*. Unity tiene muchos componentes incorporados que se pueden

utilizar con tan solo dos clics, algunos son muy utilizados y otras veces son más específicos y menos frecuentes.

2.4.3. *MonoBehaviour*

Unity se preocupa por proporcionar muchos de los comportamientos más típicos de esos componentes, con todo eso, no es suficiente para abarcar la diversidad de funcionalidades que uno se pudiera esperar. Es aquí donde entran los *MonoBehaviours*.

Un *MonoBehaviour* no es más que la clase base del cual todos los *scripts* de Unity derivan. Y es gracias a estos, que se pueden definir nuevos componentes o comportamientos para los *GameObjects*. Por ejemplo, se ha añadido un objeto *PlayerCar* en el inspector y se le ha proporcionado un nuevo componente llamado *Car* que describe su comportamiento a partir de variables y métodos. La idea es que el coche pueda moverse con esa información.

CarMovement.cs

```
1 public class Car : MonoBehaviour
2 {
3     public string brand;
4     public Color color;
5     public float speed;
6
7     public void Drive
8     {
9         // Do some stuff
10    }
11 }
```

Figura 11: Estructura simple de un MonoBehaviour

Cuando se añade al *GameObject* se muestra de la siguiente manera:

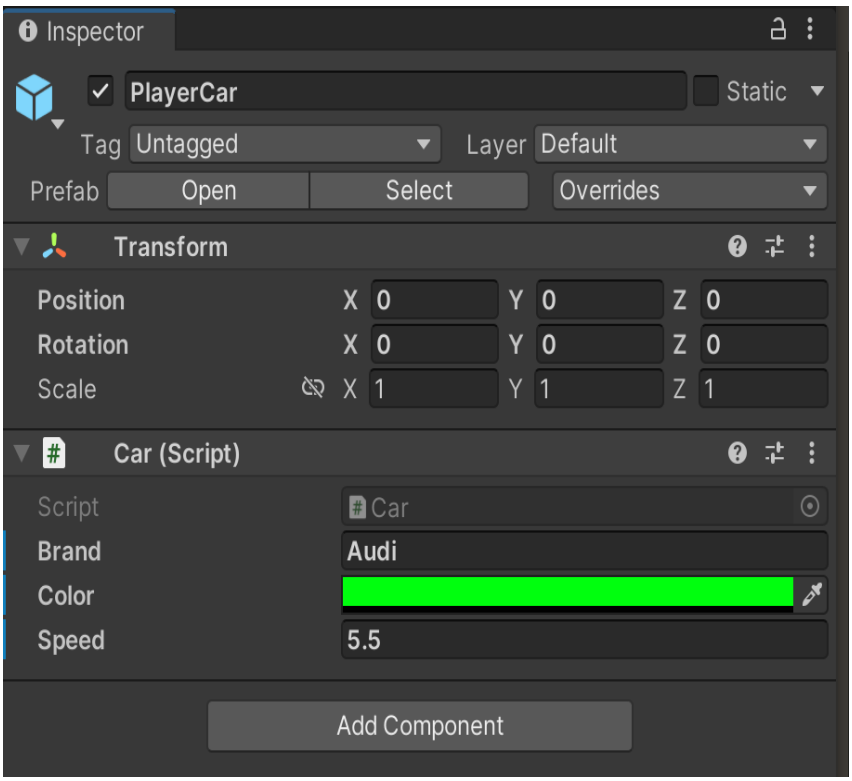


Figura 12: Visualización de un MonoBehaviour en el inspector

El potencial que ofrecen estos *scripts* es mucho mayor al de una simple clase. Con el ejemplo anterior en realidad solo se ha definido una clase con métodos y variables, sin embargo, no se ha especificado el cuándo realizar esa acción. Aquí es donde entra una de las características fundamentales de los *MonoBehaviours,* los *Event Functions* o *callbacks*. Básicamente, el s*cript* ejecuta el código que hay dentro de unas funciones que Unity proporciona y lo más interesante es que lo hace en un orden especifico y de una manera específica. Las funciones más típicas son: A*wake()*, que se llama justo después de invocar un objeto en escena y, es útil para referenciar componentes del inspector; *Start(),* que se llamará antes de la primera actualización de *frame y* se suele utilizar para inicializar variables; y, *Update()*, que se llama una vez por *frame* y suele contener la mayor parte de la lógica del juego.

Aplicando esta teoría al *script* del *GameObject* se le esta proporcionando la información suficiente para moverse o pintarse de un color:

Car.cs

```
public class Car: MonoBehaviour
{
    public string brand;
    public Color color;
    public float speed;

    private MeshRenderer mesh;
    private float currentSpeed;

    private void Awake()
    {
        mesh = GetComponent<MeshRenderer>();
    }

    private void Start()
    {
        currentSpeed = 0;
        mesh.material.color = color;
    }

    private void Update()
    {
        Drive();
    }

    ...
}
```

Figura 13: Uso de los callbacks de Unity

Llegados a este punto ya se puede visualizar las ventajas que ofrece este tipo de *scripts*. Para empezar, son las únicas que pueden interactuar con *GameObjects* y, por lo tanto, definir el comportamiento de estos. De la misma manera, pueden interactuar con otros componentes del mismo objeto o, incluso, con otros *GameObjects* que hay en la escena.

Al mismo tiempo, su gran flexibilidad convierte algunos de sus beneficios en grandes inconvenientes. Como ya se ha dicho, un componente debe describir una característica del G*ameObject*, sin embargo, este concepto se suele interpretar incorrectamente en muchas ocasiones haciendo de los *MonoBehaviours* de grandes contenedores de datos y métodos muy dispares que complican el entendimiento del contenido. En el ejemplo propuesto, ocurre esta misma problemática puesto que se mezcla la lógica del tipo de coche con la lógica de movimiento y es algo que se debería evitar. Por otro lado, el acceso a las funciones de evento puede llegar a convertirse en un caos a medida que el proyecta aumenta, ya que toda la lógica se incluye en estas funciones y se puede alcanzar la situación donde no se sepa cómo se está ejecutando ni en qué orden, porque todo está contenido en las mismas funciones en múltiples *scripts*.

A pesar de todo, estas desventajas no son más que el resultado de una mala interpretación o el desconocimiento de todas las herramientas que Unity proporciona. Seguidamente, se introducirán algunas ideas para mejorar el uso de los *MonoBehaviours* y, entender mejor algunos conceptos sobre arquitectura en videojuegos.

2.4.4. *Prefabs*

Cada *GameObjet* contendrá sus propios componentes y a medida que el proyecto crezca, seguramente, ese mismo objeto será requerido en diferentes escenas o múltiples veces en la misma. Esta situación conduce a un problema cuando se hace una modificación y se quiere propagar ese mismo cambio a los objetos iguales debido a que no existirá relación entre ellos y se tendrá que modificar uno a uno manualmente.

La solución viene dada por los *prefabs*, que son *GameObjects* convertidos en *assets*. Cuando este *asset* se modifica, propaga su cambio a todos los objetos iguales que haya en escena automáticamente, sin embargo, no ocurre lo mismo si se instancia un *prefab* en escena y se realiza la modificación desde la escena.

Esto es bastante conveniente puesto que existe la posibilidad de instanciar múltiples *prefabs* idénticos y modificar algún parámetro individualmente sin necesidad de sobrescribir a los iguales. Por otro lado, un cambio de este estilo conlleva muchas responsabilidades y cualquier error, por pequeño que sea, puede ser muy costoso de identificar.

Una alternativa para evitar esa clase de comportamientos es crear diferentes configuraciones del mismo objeto, continuando el ejemplo del coche se puede hacer lo siguiente:

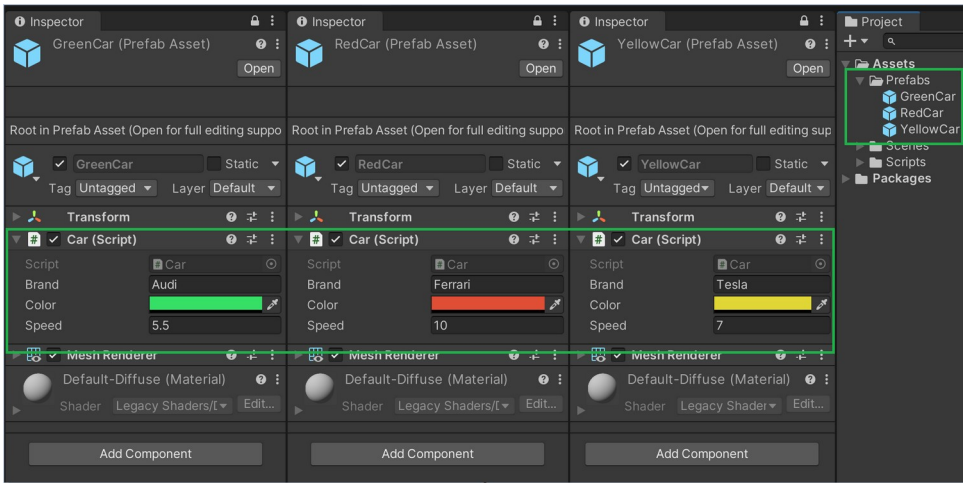


Figura 14: Multiples prefabs iguales con diferentes configuraciones

Como se puede observar, ahora se dispone del mismo objeto con algunas variaciones, ahora bien, por cada *prefab* creado se obtendrá un nuevo *asset*, lo que complicará mucho el mantenimiento de todos estos recursos y puede llegar un punto donde el proyecto sea insostenible en caso de necesitar muchas versiones, no obstante, es una posibilidad más, que bien utilizada se puede adaptar a la perfección.

2.4.5. Estado compartido y no compartido

Con lo visto hasta ahora, se puede concluir que existen dos estados diferentes de datos que residen en los objetos de la escena como los *prefabs* y que habrá que tener en cuenta. Un ejemplo rápido de entender sería el de un enemigo que tiene puntos de vida (*currentHealth*) y un valor

base de vida (*baseHealth*) que se utiliza para inicializar el valor actual de vida. Cuando se instancian varios enemigos iguales en escena, el *baseHealth* es un estado compartido, porque si los enemigos inicializarán su vida con valores distintos, sería confuso para los jugadores, en cambio, el *currentHealth*, es un estado no compartido debido a que no se espera que cuando uno sufra daño el resto también lo reciba.

En el apartado anterior se comentaba que las instancias de los *prefabs* en escena no compartían las modificaciones y esto es justo lo que significa un estado no compartido. Por lo tanto, no hay diferenciación entre estados y, todos son no compartidos si no se tienen en consideración. Esto implica que cada variable se guardará en memoria.

Para diferenciar entre estados compartidos y no compartidos hay varias maneras de conseguirlo, por ejemplo, hacer uso de variables estáticas, utilizar *prefabs* como contenedores de datos o, la más conveniente, ya que es el objetivo de su creación, los *Scriptable Objects.*

2.4.6. *Scriptable Objects*

Los *Scriptable Objects* (SOs) son clases de Unity al igual que los MonoBehaviours con la diferencia que no se pueden acoplar a los *GameObjects y* no puede hacer uso de la mayoría de *callbacks.*

"It's MonoBehaviour,..., but not a component!" [5]

A primera vista, no parece que aporte algo mejor a un *MonoBehaviour,* de hecho, lo empeora puesto que reduce sus funcionalidades más útiles. No obstante, los *Scriptable Object* no se crearon para substituir a los *MonoBehaviours,* sino para complementarlos. Además, una comparación más acertada sería con los *prefabs,* dado que los SOs se deben crear en el proyecto como *assets,* aún así, tampoco es lo mismo porque, por un lado, los *prefabs* son un *GameObject* que puede contener toda clase de componentes y ser instanciados en escena y, por otro lado, los SOs son una clase a la que se puede referenciar desde cualquier *MonoBehaviour* pero, solo existirá una única instancia por *asset* en escena.

Siguiendo el ejemplo del enemigo, en las siguiente figura se va a crear un *Scriptable Object* que más adelante se utilizará para darle salud al enemigo.

HealthConfigSO.cs

```
1 [CreateAssetMenu(menuName = "Scriptable Objects/HealthConfig", fileName =
  "HealthConfig")]
2 public class HealthConfigSO : ScriptableObject
3 {
4     public int baseHealth;
5 }
```

Figura 15: Creación de un Scriptable Object HealthConfigSO

Como se puede observar, es una clase normal pero hereda de la clase *ScriptableObject*, además, aquí se introduce un concepto nuevo que son los atributos. Los *Attributes* son marcadores que proporcionan acciones especiales sobre clases, variables o métodos. En este caso, se coloca el atributo *[CreateAssetMenu()]* encima de la clase para poder crear un *asset* desde la ventana

proyecto a partir de esa clase. Este es un caso especifico para los SOs, sin embargo, hay muchos más tipos de atributos diferentes para otras muchas situaciones.

Para hacer referencia en un *MonoBehaviour* a*l asset* del SO, se debe referenciar como si de otra clase se tratara y se especificaría como publica, de esta manera, se puede asignar más adelante desde el inspector.

Enemy.cs

```
1 public class Enemy : MonoBehaviour
2 {
3     public HealthConfigSO healthConfigSO;
4     private float currentHealth;
5
6     private void Start()
7     {
8         currentHealth = healthConfigSO.baseHealth;
9     }
10 }
```

Figura 16: Referencia en un MonoBehaviour sobre un SO

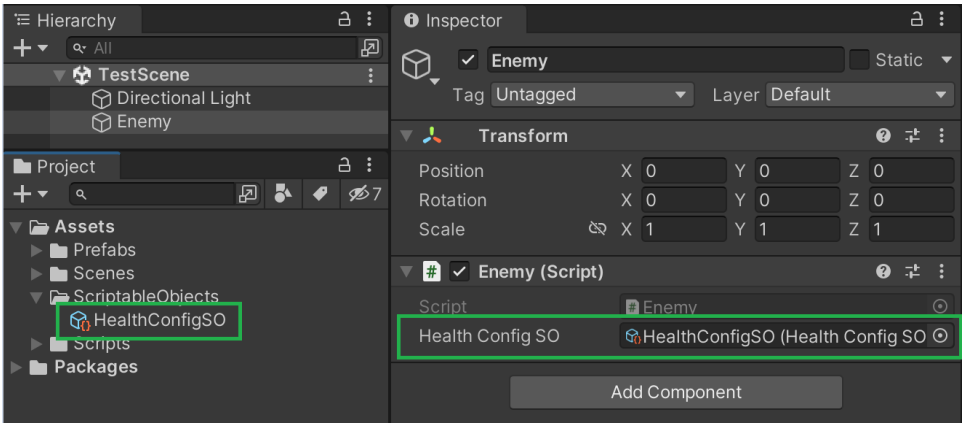


Figura 17: Vista del SO en el proyecto (asset) y en el inspector

Así pues, el objetivo principal de los *ScriptableObjects* es el de contenedores de datos para estados compartidos. Adicionalmente, dado que solo existirá una única instancia en escena, son ideales para la optimización de memoria debido a que evitan múltiples copias del mismo valor. Por esta razón, es vital para la salud de un proyecto entender la diferencia de datos porque a largo plazo puede ser muy perjudicial, especialmente, si se trabaja con *prefabs*.

Los SOs no son tan solo un lugar donde guardar información, también, permiten añadir lógica que potencian su uso en gran medida, al fin de cuentas, siguen siendo clases. En el siguiente apartado se verán algunos de los usos más típicos.

2.5. Patrones de diseño con Scriptable Objects

A medida que aparecen más herramientas más posibilidades van surgiendo. Este es el caso que demostró Ryan Hipple en 2017, en una charla sobre arquitectura de videojuegos, donde

introducía algunas variaciones a los paradigmas ya existentes a partir del concepto de los *Scriptables Objects*.

Según Hipple, todos los sistemas que se crean deben ser modulares e independientes entre sí, asimismo, cada uno de esos sistemas debe ser completamente editable basando su funcionamiento en datos y componentes que solo se centran en un solo problema, en consecuencia, cuanto más editable sea el proyecto (cuanto más control se tenga sobre los datos y componentes que haya disponibles) más fácil será depurarlo. Con esto en mente, Hipple propone varios patrones de diseño que encajan a la perfección con esta idea y, sobre todo, basado en SOs.

2.5.1. Diseño basado en variables

En la siguiente imagen, se puede observar el *prefab* del jugador (izquierda) y tres sistemas (derecha) que se encargan de algún caso específico. En el medio, se encuentra un SO que contiene la vida del jugador. Por lo general, si no se trabajará con SOs, este valor lo contendría el *prefab* del personaje, ahora bien, eso provocaría que cada uno de los componentes dependiera directamente del jugador y, si este desapareciera, entonces, todos los sistemas fallarían. En cambio, al convertir la variable de vida en un SO, los diferentes sistemas dejan de saber de la existencia del jugador y solo escuchan a la variable contenida en el SO.

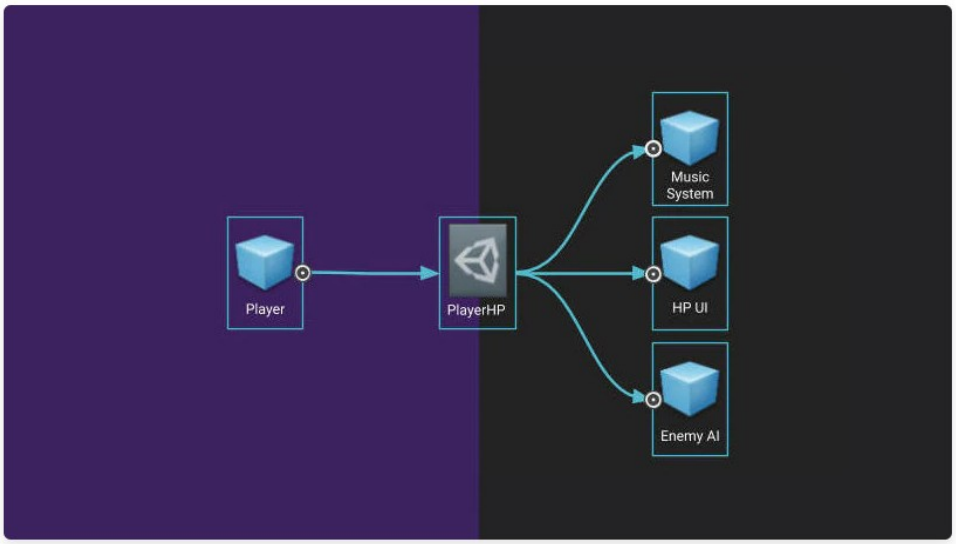


Figura 18: Ejemplo de diseño basado en variables [1, 7]

2.5.2. Diseño basado en eventos

En el siguiente caso, se introducen los eventos, que se pueden entender como canales de comunicación para enviar mensajes entre sistemas. Cada sistema tendrá la capacidad de enviar mensajes cuando ocurre algún evento en el juego y, también, podrá recibirlos y actuar en consecuencia. Así, cada sistema es independiente puesto que nadie sabe nada del otro. En el siguiente ejemplo, el jugador a muerto y, en consecuencia, envía un evento para que los sistemas que estarán escuchando realicen las acciones pertinentes.

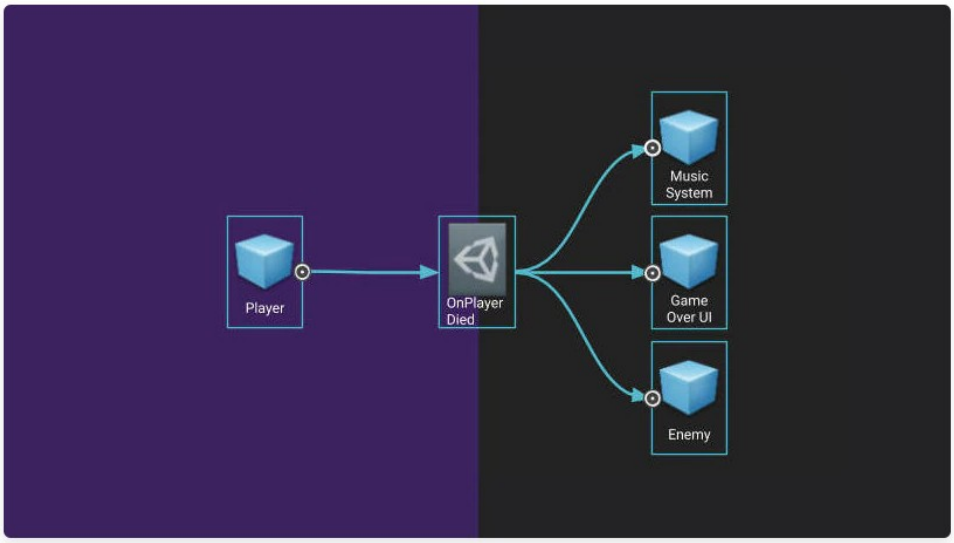


Figura 19: Ejemplo de diseño basado en eventos [1, 7]

2.5.3. Diseño basado en sistemas

Por último, hay un diseño más que habrá que tener en cuenta que se basa en sistemas. Primero de todo, hay que matizar que un *Scriptable Object* no solo contiene datos, sino que se puede añadir lógica como a un *script* normal. Esto permite convertir sistemas enteros en un solo SO. Por ejemplo, un caso sencillo sería el del inventario del jugador donde todas sus funcionalidades están contenidas en ese SO y los objetos en escena harán referencia a ese *asset* para utilizar sus funcionalidades.

3. Diseño y análisis

Antes de empezar a desarrollar el proyecto hay que definir exactamente todo lo necesario para llevarlo a cabo. Hasta ahora se ha hecho más énfasis en el cuándo dejando de lado el qué y el cómo. Es aquí el lugar dedicado a explicar en detalle esos requisitos previos a la implementación. Los puntos más destacados de este apartado son: la definición de herramientas y recursos, el diseño de niveles y mecánicas y, finalmente, la explicación de la arquitectura de forma general.

3.1. Entorno de desarrollo

El conjunto de herramientas que se van a utilizar para el entorno de desarrollo son:

- **Unity (2021.3.16f1)**: el motor de videojuegos Unity es la herramienta principal y la que facilita en gran medida el desarrollo, puesto que contiene la mayoría de recursos para construir un videojuego. La versión escogida es a propósito debido a un error de

visualización en la interfaz del inspector al utilizar *arrays* o similares y esta versión solucionaba el problema.

La elección de este motor no es arbitraría y las razones son varias. Por un lado, la comunidad es una de las más importantes en cuanto a cantidad y se puede comprobar en los datos que se recogen de las plataformas más populares de distribución de videojuegos como son itch.io y Steam, donde a día de hoy, la mayoría de videojuegos son creados en Unity [8, 9]. Esto permite que la comunidad sea muy activa en internet y sea muy fácil encontrar ayuda cuando algún problema se presenta. Asimismo, que existan tantas personas apoyando este motor significa que es una herramienta bastante accesible y fácil de usar. Si se compara con el siguiente motor más popular en Steam (Unreal Engine), se puede notar esta facilidad con tan solo ver el lenguaje de programación, ya que, en Unreal se utiliza C++, en comparación con Unity que utiliza C#. Sin entrar en detalles, el primero es un lenguaje mucho menos atractivo para la mayoría de usuarios, sobre todo, si son nuevos en el mundo de la programación.

Por otro lado, el trabajo se dirige hacia una de las utilidades que proporciona Unity, los *Scriptable Objects*. Esto significa que Unity es la única posibilidad, a no ser que se cambie por completo el tema de este trabajo o se utilicen UDataAssets que son el equivalente a los SOs de Unity.

- **Microsoft Visual Studio 2022 [17.1.32210]**: la siguiente herramienta más importante es el IDE de Visual Studio que es compatible con muchos lenguajes de programación, entre ellos C# que es el que se utiliza en Unity y se utilizará para editar el código, depurarlo y compilarlo.
- **Gitlab**: esta aplicación web es la que se encarga de guardar el proyecto dentro de un repositorio que será responsable de gestionar y compartir el proyecto, además de controlar las diferentes versiones de este.
- **Sourcetree**: finalmente, la conexión entre el proyecto en el ordenador local y el repositorio de Gitlab se realizará con Sourcetree a partir de una interfaz cómoda de utilizar.

3.2. Recursos (*Assets*)

Por lo general, los recursos que se van a escoger se van a decidir en este apartado, si más adelante, es necesario encontrar algún recursos extra se especificará en el apartado de implementación donde se dará una explicación detallada de la necesidad. Por ahora, es importante que todos los *assets* del proyecto estén decididos de antemano, ya que, durante el proceso de desarrollo puede ser prejudicial no disponer de ellos debido a que se deberá interrumpir el trabajo para buscar nuevos recursos y eso puede provocar la falta de continuidad en la implementación.

Entre otras cosas, se ha tenido en cuenta en el momento de escoger un recurso que fuesen paquetes bastante completos, es decir, poniendo de ejemplo los modelos de personajes y

escenarios, se ha intentado encontrar los recursos que más modelos proporcionarán y, así, evitar tener que depender de diferentes fuentes.

Por otra parte, también se ha tenido en consideración la compatibilidad entre recursos de diferentes orígenes puesto que si se detecta más adelante una incompatibilidad eso puede provocar tener que descartar y buscar material nuevo. Por ejemplo, los modelos deben ser compatibles con las animaciones, si no lo son se pueden generar errores innecesarios con la consecuente perdida de tiempo.

3.2.1. Estilo del juego

La temática del juego vendrá influenciada por la elección de estos recursos. Como se ha comentado al principio de esta memoria, el juego tendrá un estilo japonés y esto es debido al *asset* escogido como recurso principal. Además, se apoyará de otros paquetes de recursos que provienen de la misma fuente para complementar todo aquello que se requiera. En este caso la fuente será la compañía Synty [10], que proporciona una gran cantidad de modelos en formato Low Poly y, aunque sean de pago, siempre ofrecen buenas ofertas, de la misma manera, tienen paquetes de recursos gratuitos que ayudan en fases iniciales de proyectos, puesto que se puede comprobar si esos *assets* serán compatibles de verdad.

- **Polygon Samurai**: recurso principal que contiene escenarios y personajes al estilo japonés y otros objetos útiles como armas y accesorios diversos.
- **Polygon Dungeon**: para complementar se utilizarán algunos objetos de este paquete como piedras o minerales.
- **Polygon Nature**: debido al uso de Terrain, herramienta de Unity para crear terrenos o mapeados, es imprescindible el uso de texturas específicas para pintar diferentes tipos de superficies.
- **Polygon Starter**: este *asset* es gratuito y se utilizará en momentos iniciales del proyecto para probar personajes o escenarios que todavía no se han establecido.

3.2.2. Animaciones

En cuanto a las animaciones, se han utilizado las recomendadas por Synty, que aunque puede ser una estrategia comercial para atraer a compradores, sí que ofrecen las animaciones necesarias para la idea de este proyecto. Era relevante encontrar un proveedor único de animaciones porque, normalmente, el movimiento de un personaje queda extraño si se forma con animaciones diversas. Por lo tanto, la compañía Explosive [11] es la encargada de esta tarea, sin embargo, en caso de necesidad se recurrirá a Mixamo (web con animaciones y modelos gratuitos) para animaciones especificas que no se hayan encontrado, pero nunca para conjuntos de animaciones como el movimiento de un personaje.

- **Ninja Warrior Mecanim Animation Pack**: animaciones del personaje principal, como se puede entender, será un ninja. De nuevo, el *asset* es el que define el comportamiento.

- **RPG Character Mecanim Animation Pack FREE**: Explosive también contiene recursos gratuitos para probar sus productos, dado que contiene todas las animaciones básicas de movimientos, este paquete se utilizará tanto para el personaje principal como para los enemigos u otros NPCs.

3.2.3. Interfaz de usuario (UI)

En general, no se va a hacer mucho hincapié en este apartado pero se utilizará un *asset* de la compañía HONETi [12] para aportar una interfaz básica.

- **Dark Flat Fullsceen GUI**: contiene una combinación de elementos en 2D para la UI como fondos, marcos y diferentes iconos.

3.2.4. Música, sonidos y otros efectos

La música y los sonidos se van a tratar de forma diferentes en este proyecto porque es más complicado encontrar recursos de una misma fuente y los que existen son muy caros y no se puede aprovechar todo el conjunto como ocurre con los modelos y animaciones. Por lo tanto, se recurrirá a bibliotecas de sonido gratuitas como OpenGameArt [13] o FreeSound [14]. Aún así, se decidirá que audios utilizar durante la fase de implementación.

Por otro lado, los efectos seguirán la misma línea que el sonido y, aunque en este caso sí que hay recursos de un mismo proveedor, se esperará hasta la fase de implementación para encontrar unos efectos adecuados si corresponde. De todas maneras, Synty contiene algunos efectos que se pueden utilizar con los elementos del escenario como fuego o partículas varias.

3.2.5. Unity

Unity posee una variedad enorme de módulos o funcionalidades que ofrecen muchas posibilidades. Durante el proceso de creación se va a hacer mucho uso de estas ventajas y, ahora, se proporcionará una descripción detallada de cada una y algunos usos que se pueden dar.

- **Cinemachine**: actualmente el estándar de Unity para las cámaras de juego. Tiene múltiples opciones que permitirá ajustar la cámara de juego como mejor se prefiera.
- **Terrain**: se utiliza para generar terrenos o mapas. Es imprescindible en juegos 3D para crear los diferentes escenarios, ya que, de forma muy práctica se pueden crear montañas, pintar texturas, plantar arboles o hierbajos.
- **ProBuilder**: esta herramienta sirve para construir todo tipo de geometrías, desde pequeñas piedras hasta grandes edificios, ahora bien, su uso más extenso es el de crear el prototipo de un escenario. Se utilizará mucho en la fase inicial para construir la escena de pruebas, pudiendo crear paredes, casas o el mismo terreno. ProBuilder se suele conjuntar con otro paquete llamado ProGrids, este da soporte cuando se necesita ajustar la posición de las geometrías a la cuadricula del escenario para proporcionar cierta simetría.
- **(New) Input System**: el nuevo sistema de controles de Unity se ha convertido en el modelo por defecto para esta tarea. Tiene muchas características que combinan a la

perfección con la idea de este proyecto como el mapeado de controles en un mismo lugar o el uso de eventos para enviar la información sin que el código base sepa de la existencia del sistema de entradas.

- **Text Mesh Pro**: otra de las funcionalidades que se han convertido en estándar en el motor de Unity, en este caso, su uso principal es el de configurar los componentes de texto de la UI ofreciendo múltiples opciones de configuración.
- **Localization**: paquete para configurar todos los textos en diferentes idiomas a partir de una tabla que utilizará la UI para colocar el texto adecuado en cada momento.
- **ScriptableObject Architecture** [15]: esta utilidad es muy conveniente para la idea del proyecto puesto que facilita el uso de eventos a partir de SOs permitiendo la generación de eventos con unas pocas instrucciones. Es el único paquete de esta lista que no es parte de Unity, pero es totalmente compatible.

3.3. Diseño del videojuego

Se va a crear un prototipo de videojuego en 3D con las mecánicas más básicas típicas de un ARPG, ejemplos destacados de este género son la serie Monster Hunter, Kingdom Hearts o Crisis Core. Evidentemente, la intención no es hacer un videojuego tan completo, sencillamente replicar algunas de sus mecánicas más típicas.

Seguidamente se explicará el diseño del juego donde se comentarán los diferentes aspectos referentes tanto a temas artísticos como más técnicos.

3.3.1. Diseño de niveles

Como en MH, habrá dos escenas diferenciadas. La primera es una zona de descanso donde los jugadores pueden aceptar misiones y hablar con los diferentes NPCs, por otro lado, estará la escena donde ocurre toda la acción y es ahí donde los jugadores deberán completar misiones y luchar contra enemigos, por lo que, solo en esta escena se permite el sistema de combate.

- **Escena *Village* (aldea)**: esta es la escena inicial donde empezará el personaje y será el punto de referencia donde siempre hay que volver después de una aventura. Aquí habrá un panel de misiones donde poder aceptar peticiones y completarlas.

  En tamaño, se debe procurar no hacer un escenario muy grande debido a que no hay demasiado contenido y espacios muy grandes pueden frustrar la experiencia del usuario. El escenario se puede crear libremente pero debe coincidir con las siguientes características: diversas casas agrupadas en una calle principal ancha que servirá de camino de transito hacia la escena de acción, un rio y un mar que sirvan de limitación del escenario y, por último, debe parecer una escena japonesa incluyendo elementos que den esa sensación.

  Finalmente, se debe colocar una especie te tablón que servirá de elemento para activar el sistema de misiones. Deberá estar posicionado en un sitio estratégico como en mitad del pueblo o en la salida hacia la escena del bosque para que sea fácil visualizarlo.

- **Escena *Forest* (bosque)**: el bosque será la escena donde ocurra toda la acción y donde se pueden completar misiones. La interacción en esta ocasión se realizará a través de los combates con diferentes enemigos y la recolección de materiales por parte del personaje.

  La idea es conseguir un mapa situado dentro de un bosque con una sola entrada y salida, que corresponde con la salida de la aldea en la escena anterior. Asimismo, el tamaño del escenario será medio, limitado por alguna especie de obstáculo en el horizonte. Al ser un bosque se espera que la mayor parte del mapeado se rellene con arboles, con la posibilidad de incluir un rio o alguna especie de estanque. También, se requiere de una ubicación especial para romper con la monotonía del bosque, como podría ser un templo o castillo de estilo japonés.

Ambas escenas tienen que crearse con los *assets* proporcionados por Synty, sin embargo, la base del escenario debe moldearse con la herramienta de Terrain. Ahora bien, en fases iniciales no se hará uso de estas escenas y se creará un escenario de pruebas donde, por facilidad de uso, se usará la herramienta Probuilder que permite crear escenarios u objetos de forma muy rápida y sencilla.

3.3.2. Mecánicas

Entre todas las acciones que un jugador puede realizar en un juego ARPG se ha decidido añadir un sistema de combate básico, un inventario para poder recolectar objetos y un sistema de misiones para ofrecer un objetivo que cumplir. En más detalle:

- **Sistema de combate**: el combate no será algo espectacular en cuanto a físicas o efectos, más bien se centrará en la interacción entre sistemas como la salud o el estado del juego. Además, la IA y el control del personaje principal serán relevantes aquí debido a su funcionamiento por estados.

- **Inventario**: el inventario es lo más típico dentro de los videojuegos y en los juegos de rol aún más. En primera instancia, la idea es crear la posibilidad de guardar objetos que se pueden recoger en el escenario y dar visibilidad desde una interfaz, más adelante si el tiempo es favorable se puede añadir la posibilidad de interactuar con algunos objetos.

  Este sistema es conveniente para el proyecto porque interactúa con muchos sistemas dentro del juego. Por un lado, se deberá recoger un objeto, que es totalmente independiente al inventario, guardarlo en él y darle visibilidad por pantalla, además, toda esta información que se guarda en un inventario deberá interactuar con el sistema de misiones de alguna manera, y todo esa información, al final, deberá viajar entre escenas sin perder nada por el camino. Con todo esto, el inventario es uno de los componentes más completos en este videojuego internamente.

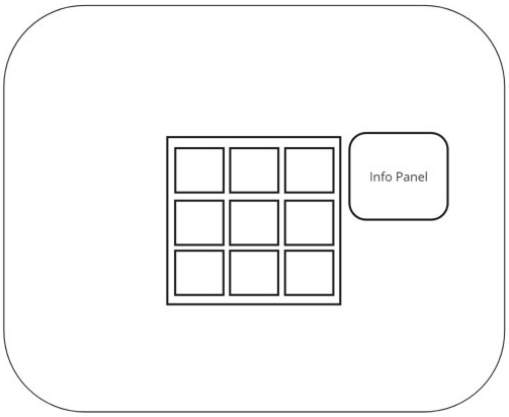
- **Misiones**: para que el juego sea menos contemplativo y haya algo que hacer se crearán misiones. Al igual que en los MH, estas se obtendrán en un tablón y consistirán en tareas como recolectar objetos o matar enemigos.

Para visualizar las misiones se abrirá una pantalla mostrando los diferentes objetivos que podrán ser aceptados. Las misiones se completarán en la escena de acción y, al volver al tablón, será posible finalizarlas a partir de una interacción con la interfaz.

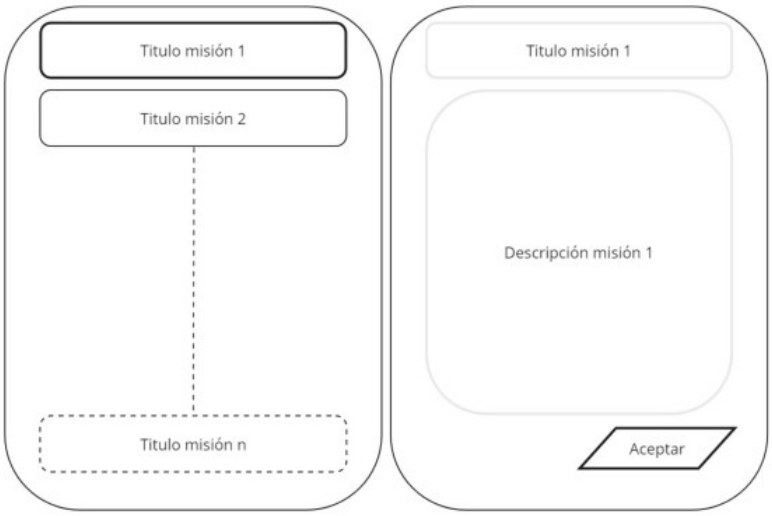
3.3.3. Interfaz de usuario

La interfaz o UI, por sus siglas en inglés, no pretende destacar ni ofrecer la mejor experiencia de usuario. Sencillamente, se va a mostrar por pantalla lo más necesario para que el juego pueda fluir en sus objetivos.

- **Inventario**: Los objetos que se recolecten en la escena serán guardados en el inventario y se mostrarán en una interfaz. Esta mostrará en una cuadricula los ítems adquiridos junto con su cantidad, de la misma manera, si se pasa el ratón por encima de los objetos se abrirá un panel con la descripción de cada uno.



- **Panel de misiones**: desde un tablón físico dentro del juego, en la escena de la aldea, se podrá abrir un menú donde aceptar o completar misiones. Este menú mostrará las misiones disponibles y su estado, además, cuando son seleccionadas se mostrará en un panel adicional toda la información relacionada (título, descripción y objetivo). Por otra parte, un botón, que cambiará dependiendo del estado de la misión, se mostrará debajo de ese panel.



3.3.4. Arquitectura

Cambiando de tercio, ahora hay que hablar sobre el diseño de la arquitectura. La arquitectura va a ser responsable de comunicar los diferentes componentes y sistemas que existen en el juego. Evidentemente, esto es algo que ocurrirá de forma transparente al usuario final, sin embargo, es donde recaerá toda la carga de trabajo.

En primer lugar, habrá dos escenas principales que se pueden generalizar llamándolas escena de menú y escena de juego. Son principales porque solo una puede estar activa al mismo tiempo y son las únicas interactuables por el jugador. Asimismo, habrá dos escenas más que darán soporte a las principales activándose en paralelo, estas son: *Gameplay Scene*, que incluirá toda la lógica para dar soporte a la interfaz teniendo en cuenta que solo afecta al *gameplay*; y *Managers Scene*, por su lado, contendrá lógica relacionada con lo más técnico como el cambio de escena o el sistema de sonido. Dado que el *gameplay* no puede ocurrir dentro de un menú, esa escena no será necesaria en ese momento. Todo esto es gestionado por un objeto que hay en las escenas principales llamado *SceneInitializer* que cargará las escenas en paralelo al iniciar una escena principal.

Con todo esto, ahora se puede entender mejor la siguiente figura. Hay cuatro zonas diferenciadas que representan cada una de las escenas comentadas más arriba y dentro de cada cuadrado se especifican los elementos más importantes que se pueden encontrar. Esta división es muy conveniente para aislar esos componentes que se reutilizan constantemente en las escenas principales y, al existir una escena especifica esos recursos son siempre los mismos y no hace es necesario cargarlos constantemente.

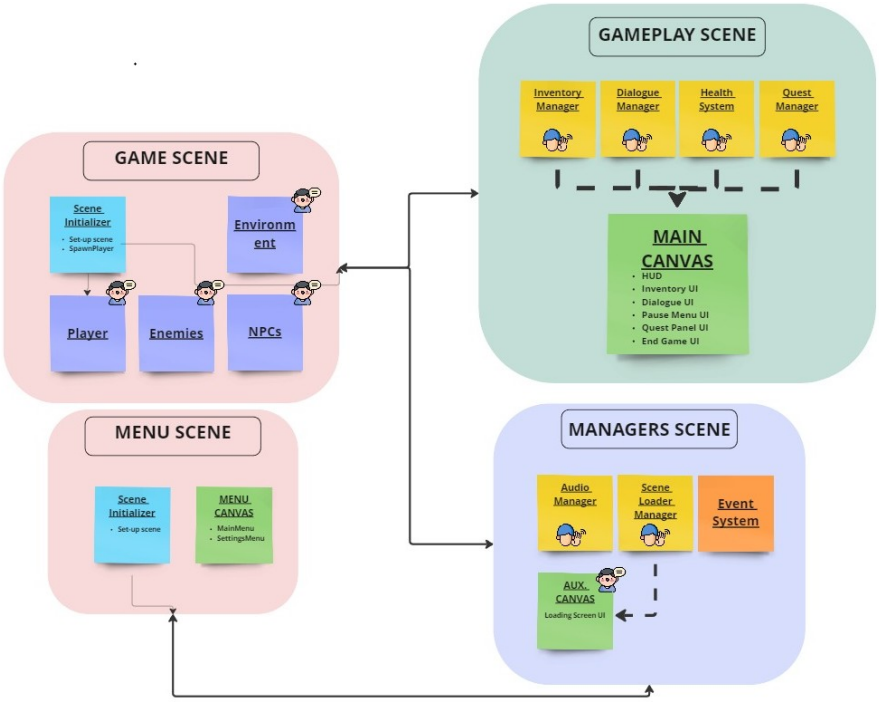


Figura 20: Arquitectura por escenas del videojuego

A partir de aquí, cualquier componente será capaz de comunicarse a través de unos canales de comunicación que en el diagrama están representados por las flechas que hay entre escenas. Cuando un componente de un objeto quiere enviar una señal para que otro sistema reaccione, simplemente, hará uso de este mecanismo, además, tanto el emisor como el receptor no deben saber el uno del otro para evitar dependencias innecesarias.

4. Implementación

4.1. Introducción

En este apartado se seguirá la subdivisión del proyecto en *sprints* para explicar la implementación de este trabajo. En cada parte, se comentará el contenido realizado durante ese periodo y los resultados conseguidos, así como una explicación de la parte técnica más relevante.

4.2. *Sprints*

Cada *sprint* corresponde con un periodo de dos semanas de trabajo y hay un total de 8, también, existe el *sprint* 0, sin embargo, corresponde con todo el contenido incluido en los apartados de planificación, análisis y diseño, por lo tanto, no se incluirán en esta sección y, de igual manera, el 8 es una fase de documentación y no habrá necesidad de comentarlo.

4.2.1. *Sprint* 1 (27 Feb – 12 Mar)

La primera iteración sirve de primer contacto con las diferentes herramientas ya mencionadas, por lo que, inicialmente, el objetivo es configurarlas para su puesta en marcha. En este punto, se

generará un escenario de pruebas y los primeros sistemas que ayudarán a crear el personaje principal.

En una primera instancia, se genera el repositorio en Gitlab que se puede encontrar en este enlace. Seguidamente, con Sourcetree se clona el repositorio en el ordenador local y, finalmente, crear el proyecto de Unity dentro de ese repositorio. A partir de aquí, cada vez que se quiera subir una nueva versión al repositorio de Gitlab, se generará una branca de la carpeta principal que contendrá todos los cambios realizados y, una vez los cambios se hayan subido poderlo combinar con el proyecto principal.

Dos apuntes antes de continuar con la parte de proyecto. Cada una de las brancas creadas deben tener un único motivo, es decir, si se esta trabajando en el modo de combate, no es aconsejable empezar a trabajar, por poner un ejemplo, con el sistema de misiones sin haber subido antes los primeros cambios, mezclar diferentes conceptos en una misma branca puede ser perjudicial, sobre todo, cuando ocurren errores y hay que revisar versiones anteriores. Por otro lado, las brancas se nombrarán según su finalidad, es decir, si es una nueva funcionalidad se nombrara *feature* acompañado de una barra y su premisa, así pues, el resultado sería *feature/character_movement*. En cambio, cuando no es una función sino algo que hay que arreglar se cambiará fe*ature* por *fix*.

En cuanto a la configuración en Unity, una vez el proyecto está creado, se crearan las carpetas necesarias en la ventana de proyecto, estás serán necesarias para ordenar correctamente cada tipo de archivo que se añada, entre estos se encontrarán los *scripts*, imágenes, audios, etc. Cada carpeta debería ser auto explicativa para que no sea necesario tener que dar una descripción de su contenido. También, hay que remarcar que el conjunto de carpetas están dentro de otra con el nombre de *_Game.* Esto es así para evitar que cuando se instalen paquetes externos, se evite hacerlo dentro de la carpeta principal, ya que, _G*ame* es la carpeta que se envía al repositorio de Gitlab y, por norma general, si se instala un paquete externo, primero hay que probarlo antes de saber si se quiere incluir en el proyecto. En caso afirmativo se incluirá en la carpeta *AssetStore* y en caso negativo, se eliminará, es más, si por error no se elimina, no pasaría nada porque estaría fuera de la carpeta del repositorio y, en consecuencia, es como si no existiera.

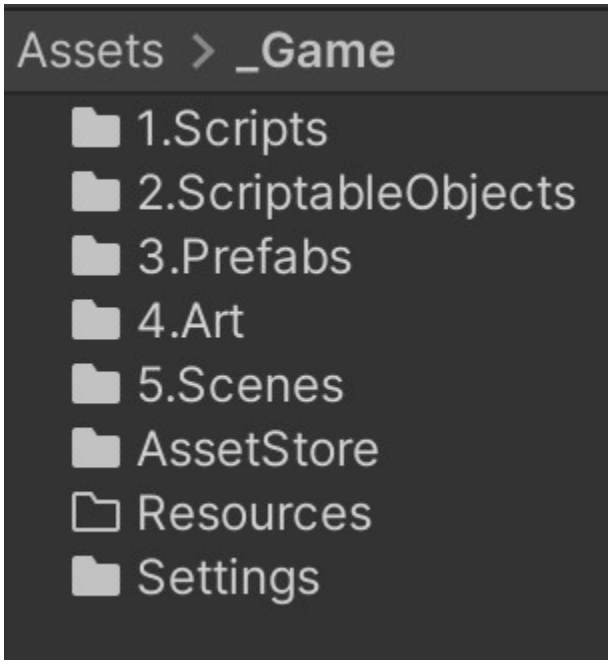


Figura 21: Organización de carpetas en la ventana proyecto

Una vez todo se ha configurado correctamente, es el momento de crear un escenario de pruebas haciendo uso de ProBuilder. Primero, se crea la escena y, a continuación, a partir de diferentes geometrías se obtiene un mapa básico donde poder probar los personajes que se creen, los objetos u otras posibles interacciones.

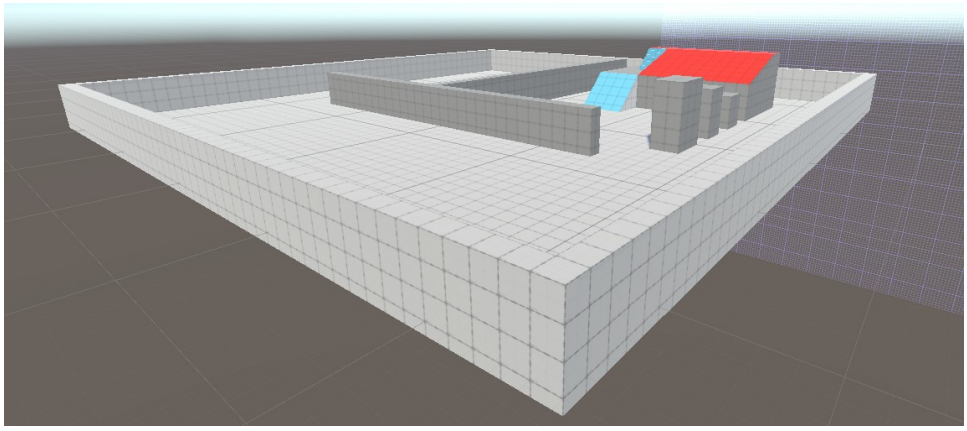


Figura 22: Escena de pruebas

Por lo que respecta al personaje, en una primera versión se escogerá un modelo de Synty genérico para empezar a construir la base de su lógica. Esta se centra, por el momento, en dar movimiento al personaje junto a sus animaciones. No hay mucho que destacar en el código, incluso al no disponer de un sistema de estados, se combinará toda la lógica en un solo *script* hasta que sea posible separarla. Hay que recordar que una buena estructuración de arquitectura pasa por crear unidades lógicas que no mezclen diferentes funcionalidades (principio de responsabilidad única), sin embargo, en este caso es recomendable puesto que hay que controlar

el orden de ejecución entre las físicas y animaciones. Más adelante se verá como separar estas dos cosas para cumplir con las buenas prácticas.

Al utilizar el nuevo Input System, es posible separar los controles del código mapeando en un *asset* externo todas las acciones posibles. También, este nuevo sistema genera automáticamente una i*nterface* que hace referencia a cada una de las acciones del mapa de controles para dar acceso al *input* del jugador. Para acceder a estas entradas se precisa de un elemento adicional que será la parte más interesante de analizar. Remontando a la teoría sobre *Scriptable Objects* y sistemas basados en ellos se va a crear lo que se nombrará *Input Reader*.

InputReader.cs

```cs
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.InputSystem;

[CreateAssetMenu(fileName = "InputReader", menuName = "Scriptable Objects/InputReader")]
public class InputReader : ScriptableObject, PlayerInput.IGameplayActions
{
    // Definición de Unity Actions
    public event UnityAction<Vector2> MoveEvent = delegate { };
    public event UnityAction<Vector2> LookEvent = delegate { };
    public event UnityAction StartRunningEvent = delegate { };
    public event UnityAction StopRunningEvent = delegate { };

    private PlayerInput playerInput;

    private void OnEnable()
    {
        // Se crea un única instancia del PlayerInput
        if (playerInput == null)
        {
            playerInput = new PlayerInput();
            // PlayerInput contiene eventos por cada acción que habrá que escuchar
            playerInput.Gameplay.SetCallbacks(this);
        }
    }

    // Cuando Player Input envíe la señal de entrada se activará la función
    // correspondiente e invocará (enviará) un evento con la información que pertoque
    public void OnMove(InputAction.CallbackContext context)
    {
        MoveEvent.Invoke(context.ReadValue<Vector2>().normalized);
    }

    public void OnLook(InputAction.CallbackContext context)
    {
        LookEvent.Invoke(context.ReadValue<Vector2>().normalized);
    }

    public void OnRun(InputAction.CallbackContext context)
    {
        switch (context.phase)
        {
            case InputActionPhase.Performed:
                StartRunningEvent.Invoke();
                break;
            case InputActionPhase.Canceled:
                StopRunningEvent.Invoke();
                break;
        }
    }
}
```

Figura 23: Scriptable Object Input Reader

El *Input Reader* es un SO que va a funcionar como un intermediario entre la entrada del jugador (teclado, *gamepad*, …) y cualquier componente del juego que necesite actuar en consecuencia. Para informar de las entradas, el SO enviará eventos (para la transmisión se utiliza el método *Invoke*) que contendrán la información necesaria, quien quiera recibir la información deberá suscribirse a estos eventos y, así, obtener los datos que requiera.

En este caso los eventos a los que se hace referencia son UnityActions. Con este tipo de eventos se puede activar dinámicamente diversas funciones que estén suscritas y hagan referencia al SO. En el caso que se está analizando, es decir, el movimiento del personaje que se encuentra en el

script PlayerMovement, se hará una referencia hacía el *Input Reader* y se suscribirá a la acción que corresponda con el movimiento (ver método *OnEnable*). Sin entrar en detalles de lógica, el código se vería de la siguiente manera:

PlayerMovement.cs

```
1  public class PlayerMovement : MonoBehaviour
2  {
3      // Dependencia con el input reader
4      [SerializeField] private InputReader inputReader = default;
5
6      // En el callback OnEnable hay que suscribirse a los eventos
7      private void OnEnable()
8      {
9          inputReader.MoveEvent += OnMove;
10         inputReader.LookEvent += OnLook;
11         inputReader.StartRunningEvent += OnStartRun;
12         inputReader.StopRunningEvent += OnStopRun;
13     }
14
15     // Es necesario darse de baja una vez se deshabilite el componente
16     private void OnDisable()
17     {
18         inputReader.MoveEvent -= OnMove;
19         inputReader.LookEvent -= OnLook;
20         inputReader.StartRunningEvent -= OnStartRun;
21         inputReader.StopRunningEvent -= OnStopRun;
22     }
23
24     private void OnMove(Vector2 input)
25     {
26         // lógica de movimiento
27     }
28
29     private void OnLook(Vector2 input)
30     {
31         // lógica de cámara
32     }
33
34     private void OnStartRun()
35     {
36         // lógica de correr
37     }
38
39     private void OnStopRun()
40     {
41         // lógica de parar de correr
42     }
43
44
```

Figura 24: Script PlayerMovement

4.2.2. *Sprint* 2 (13 Mar – 26 Mar)

Ahora que se ha proporcionado al personaje de movimiento y se ha activado el *Input System*, es el momento de añadir el sistema de combate. Este consiste en: crear dos acciones de ataque para el personaje principal, crear un enemigo y asignarle una IA y, finalmente, crear todo el sistema de salud. En el anterior *sprint* ya se explicó cómo se creaban las acciones y no hay necesidad de comentar como funciona de nuevo esa interacción entre jugador y personaje cuando se ataca, por consiguiente, este apartado se focalizará en el enemigo y el sistema de salud.

El modelo del enemigo se obtendrá de Synty y se le añadirá poco a poco todos los componentes que se necesiten para darle físicas e inteligencia. Para la IA se va a utilizar el patrón State a través de un FSM, hay que recordar que este patrón pretende proporcionar a un objeto con diferentes lógicas pero solo una está activa al mismo tiempo. Entonces, en el diagrama que se muestra en la figura siguiente se puede ver como funciona exactamente:

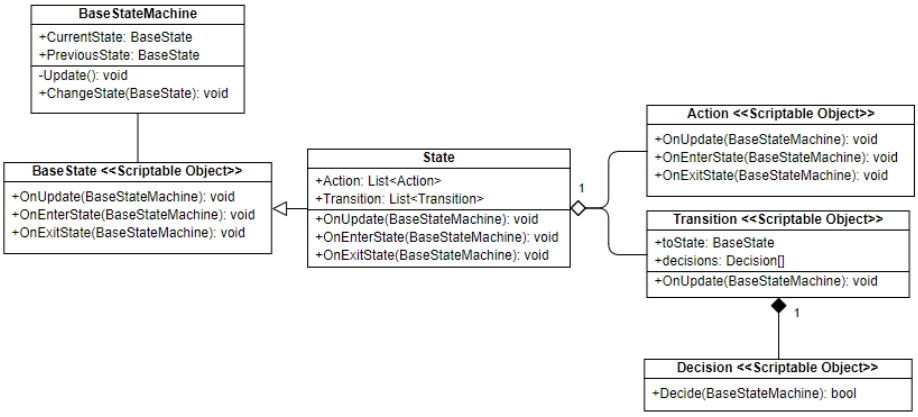


Figura 25: Diagrama del patrón State

El enemigo va a requerir de un componente que funcione de cerebro y, en el diagrama, esta función la cumple el *BaseStateMachine*. Esta clase controla el estado del enemigo a partir de una propiedad *CurrentState* que es del tipo *BaseState* y la irá actualizando dependiendo de lo que ocurra dentro de cada uno de los estados que ejecutarán toda su lógica dentro del método *Update*. Cada estado contiene una lista de acciones y transiciones y, al mismo tiempo, cada transición contendrá decisiones que son las encargadas de comprobar si hay que cambiar de estado o mantenerse en el actual.

La implementación mediante *Scriptable Objects* aporta varias ventajas en este diseño, entre ellas, mecaniza el proceso de crear estados y transiciones sin necesidad de crear nuevos *scripts* y divide toda la lógica en la unidad más pequeña posible con la creación de acciones y decisiones. El único inconveniente es que cuanto más compleja sea la máquina de estados, más SOs se necesitarán y, mantener un buen orden, será crucial para un buen mantenimiento de los *assets* en la ventana de proyecto.

El FSM del enemigo tiene el siguiente aspecto:

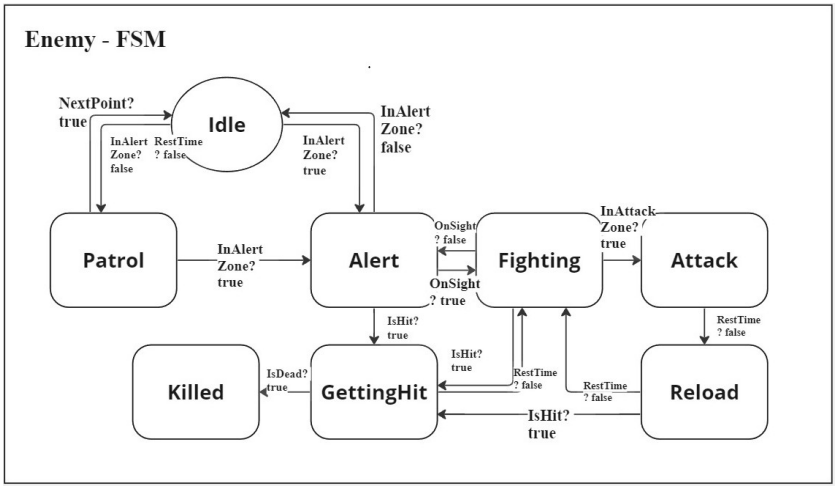


Figura 26: FSM del enemigo

En general, desde cualquier acción y decisión se tendrá acceso a cualquier componente que tenga el enemigo en su jerarquía, uno de estos componentes será el *Damageable*. Este *script* es muy básico y su función es aplicar daño o establecer el estado de muerte cuando la vida llegue a cero, ahora bien, esto permite introducir dos nuevos conceptos en este proyecto. El diseño basado en variables y eventos.

En primer lugar, el diseño basado en variables se basa en contener una o varias variables en un *Scriptable Object*, en este caso, la vida del personaje se asignará al SO *HealthSO*. Hay que tener en cuenta que la vida es una variable no compartida y al asignarla a un SO se genera una nueva problemática puesto que el objetivo es asignarla a varios personajes (enemigos y protagonista), evidentemente, cada uno tiene su propia vida, entonces, habrá que crear una única instancia por cada personaje para que no compartan el valor de la vida en caso de reutilizar el mismo H*ealthSO* (en Unity es posible crear desde un *script* instancias únicas a partir de SOs). Este caso puede llegar a ser confuso porque la solución fácil es añadir la variable directamente en *Damageable* sin la necesidad de un SO y, aunque eso es cierto, este método permite un plus de configuración que una simple variable dentro del *MonoBehaviour* no otorga.

Gracias al SO ahora es posible compartir la información de vida sin depender de la clase Damageable, no obstante, sin esa dependencia hay que encontrar la manera de comunicarse con quien solicite ese valor de vida. Para ello, se utilizará el diseño basado en eventos con *Scriptable Objects* que es básicamente el patrón Observer. Hay que decir que este patrón ya se ha utilizado en el *sprint* anterior con el InputSystem, ahora bien, en esta ocasión se utilizará otro método que simplifica el código necesario para establecer todo el sistema.

El primer paso es saber quien son los implicados, estos son: Damageable (*Subject*) y HUD (*Observer*). Este último puede ser cualquiera que necesite saber la vida de un personaje, pero por ahora, solo se contemplará en el HUD que contendrá una barra de vida que ira variando dependiendo del daño recibido. Además de todo esto, hay que abrir un canal de comunicación y aquí es donde entra el paquete mencionado en el apartado de recursos ScriptableObjects-Arquitecture.

Con esta herramienta se pueden generar SOs que funcionen como eventos o receptores de eventos (*listeners*) y sean capaces de enviar cualquier tipo de dato. Esto significa que los eventos se crearán directamente en el proyecto como un *asset* sin necesidad de código propio. Solo existe una excepción debido a que este método solo puede enviar un tipo de dato al mismo tiempo, como solución se puede crear una clase o un SO que contenga diversos datos y funcione como un objeto único. En este caso, se va a crear un evento y un *listener* a partir del *Scriptable Object HealthSO*, entonces, el *Subject*, dependerá del evento que enviará con la información de salud y, el Observer, a partir de un *listener*, recibirá la señal que activará el método al que haga referencia. En las siguientes figuras se puede observar el funcionamiento:

Damageable.cs

```
using ScriptableObjectArchitecture;

public class Damageable : MonoBehaviour
{
    [Header("Configuration")]
    [SerializeField] private HealthSO healthSO;
    [Header("Broadcasting on channels")]
    [SerializeField] private HealthSOGameEvent refreshHealthUI;

    public void ReceiveDamage(int damage)
    {
        healthSO.TakeDamage(damage);
        refreshHealthUI?.Raise(healthSO);
    }
}
```

Figura 27: Script Damageable

En *Damageable* se hace referencia al canal de comunicación *HeathSOGameEvent*. Cuando se recibe daño se activa este canal con el método *Raise* al que se le pasa el SO *HealthSO*, de esta manera, el evento se habrá enviado correctamente si *healthSO* no esta vació.

HUDManager.cs

```
public class HUDManager : MonoBehaviour
{
    [Header("Dependencies")]
    public HealthUI healthUI;

    public void UpdateHealthBar(HealthSO healthSO)
    {
        healthUI.RefreshHealthBarUI(healthSO.CurrentHealthNormalized);
    }
}

```

Figura 28: Script HUDManager

La ventaja en el *Observer* (*HUDManager*) es que no se hace referencia al evento directamente en el código, a diferencia de lo que ocurría con las *UnityActions* donde había que subscribirse a los eventos para poder utilizarlos. Esto es así porque los *listeners* que se crean se añaden como componentes (siguiente figura), entonces, a partir de esos componentes se hace referencia a los métodos que se quieren activar en el *Observer*, en este caso *UpdateHealthBar* que pide el HealthSO como parámetro de entrada.

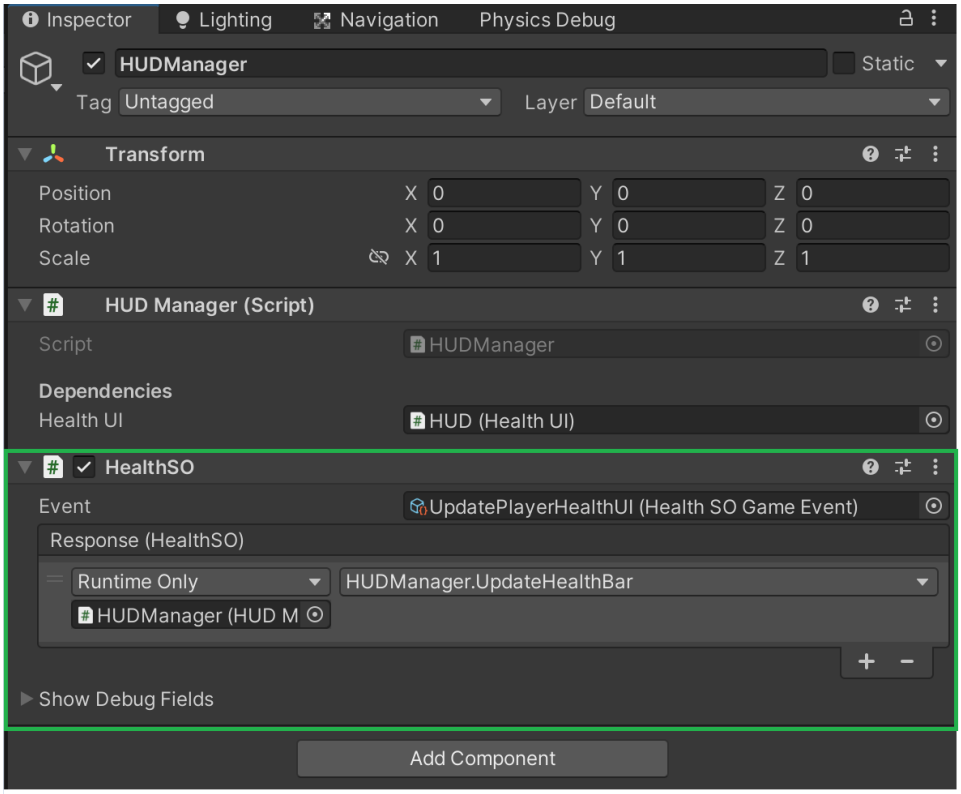


Figura 29: Componente HudManager junto al listener del evento

4.2.3. *Sprint* 3 (27 Mar – 09 Abr)

El siguiente paso es acabar de pulir el personaje principal y eso pasa por controlar el estado del juego. También, con el personaje ya creado, se puede empezar a probar el cambio de escenas y la consecuente invocación del protagonista en escena cuando es necesario.

Dado que se ha levantado una máquina de estados con el enemigo, se va a aprovechar la misma estructura de código para controlar el estado del protagonista. Otra ventaja de haber creado ya la FSM es que ahora no hay que tocar código a excepción de las nuevas acciones y decisiones que se requieran, pero de las clases que forman la estructura no hay que hacer ningún retoque, puesto que funcionan de la misma manera. Por otro lado, en el primer *sprint* se juntaron animaciones y físicas, ya que, en ese momento no había ningún sistema operativo para separarlas, aunque se podían separar por componentes, la idea final era incluirlo en las acciones de cada uno de los estados, a consecuencia de eso, era mucho mejor hacerlo sencillo al principio y, en este punto, separarlo definitivamente utilizando acciones individuales para ejecutar las animaciones u otro tipo de acción. En definitiva, el diagrama del personaje se ve tal que así:

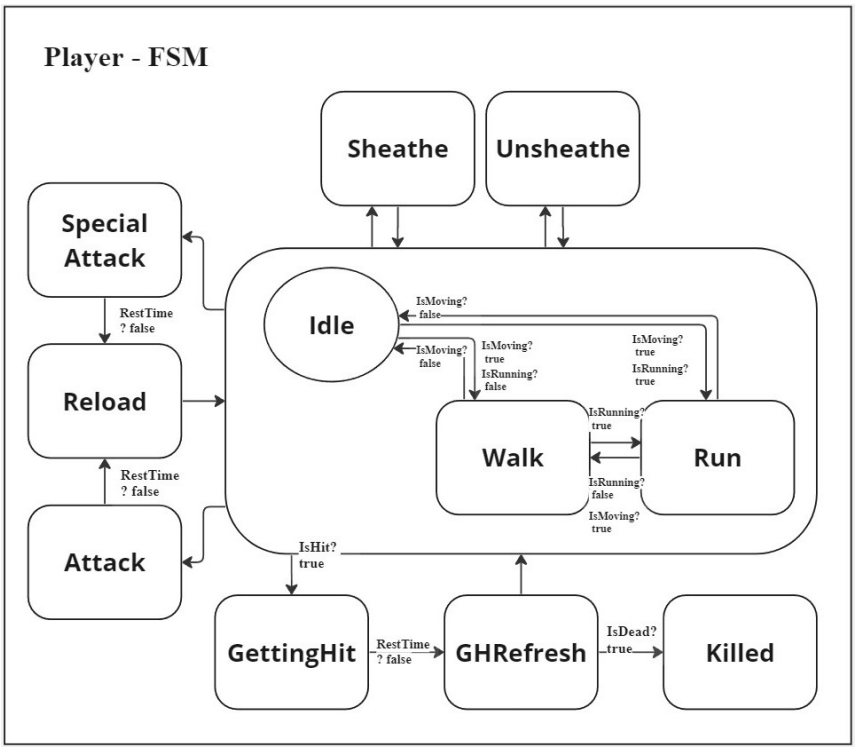


Figura 30: FSM del personaje

En la figura se puede ver como hay tres estados principales: *Idle*, *Walk* y *Run*. Se supone que dependiendo de las entradas del jugador se estará en uno u otro, por esta razón, siempre se parte de uno de estos tres. Son estados que cuando están activos son constantes y, la mayoría de estados partirán siempre de uno de estos tres.

En otro orden de cosas, otra manera de controlar el estado del juego y en consecuencia ciertos aspectos que afectan directamente al personaje, es el *GameManager*, que será un SO y se encargará de guardar el estado actual y de enviar eventos al *GameStateListener* (*MonoBehaviour*) para que actúe en conveniencia. Los estados están relacionados con la situación dentro del juego, por ejemplo, el estado *Menu* significa que el jugador está en la pantalla de menú o, el estado *InGame* significa que estará en la pantalla de juego. Las acciones que se pueden realizar en cada estado estarán relacionadas sobre todo con el control del jugador y la visualización de la UI. Por otro lado, para pedir un cambio de estado se debe utilizar el *GameStateChanger* que a través del *GameManager* se solicitará el cambio de estado.

Un estado es un simple SO vacío, funciona como un *enum* con la diferencia que no hay que crearlo desde código cada vez que se quiera añadir uno nuevo.

GameStateSO.cs

```
1 [CreateAssetMenu(fileName = "GameState", menuName = "Scriptable Objects/Game State")]
2 public class GameStateSO : ScriptableObject
3 {
4 }
```

Figura 31: ScriptableObject GameStateSO

Una variación al sistema de eventos visto en el anterior *sprint* es añadir el *listener* directamente en el código como si se estuviera subscribiendo al evento. En este caso, la modificación es para probar diferentes versiones pero no aplica una ventaja a lo ya visto. Asimismo, cuando se recibe un evento se mira que estado es el actual y, a través de eventos de Unity, se invocan las acciones que se quieran.

GameStateListener.cs

```
public class GameStateListener : MonoBehaviour
{
    [Header("Listening to Events")]
    public GameStateSOGameEvent gameStateChangedEvent;

    [Header("Actions")]
    public UnityEvent onMainMenuState = default;
    public UnityEvent onGameState = default;

    private void OnEnable()
    {
        gameStateChangedEvent.AddListener(GameStateChanged);
    }

    private void OnDisable()
    {
        gameStateChangedEvent.RemoveListener(GameStateChanged);
    }


    private void GameStateChanged(GameStateSO newGameState)
    {
        switch (newGameState.name)
        {
            case "MainMenu":
                onMainMenuState.Invoke();
                break;
            case "InGame":
                onGameState.Invoke();
                break;
            default:
                Debug.LogError("State not recognized");
                break;
        }
    }
}
```

Figura 32: Script GameStateListener

En el inspector se podrá visualizar de la siguiente manera, donde en cada caso se aplica una acción, en este ejemplo, se pasa la UI del juego por secciones y se especifica que interfaz estará activada o desactivada según el estado actual.

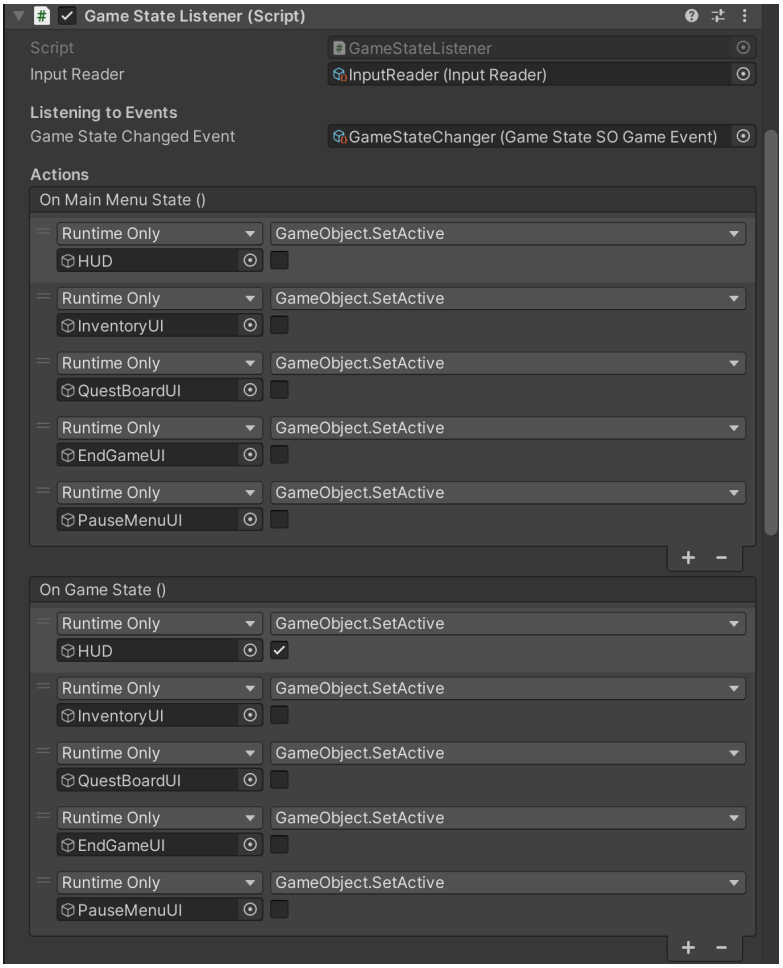


Figura 33: Componente GameStateListener en el inspector de Unity

A continuación, es el momento de probar con el cambio de escenas y se va a aprovechar para crear la división de escenas vista en la arquitectura. Sin entrar mucho en detalles de código se va a explicar las partes más importantes.

Para empezar se crean las escenas implicadas: *MainMenu*, *Gameplay* y *Managers*, asimismo, con la idea de hacer pruebas de cambio de escenas de juego, se clonará la escena de *test*. Por ahora, la escena de menú se incluirá un botón para hacer el cambio, el *Gameplay* incluirá el *canvas* que se mostrará durante el juego y *Managers* será el responsable del cambio de sostener la lógica del cambio de escenas.

En otro orden de cosas, cuando una escena de menú o juego es cargada, habrá un objeto responsable de inicializar ciertos elementos en dos tiempos, este es el *SceneInitializer*. Primero, se cargarán las escenas auxiliares si todavía no lo están y, una vez cargadas, la escena principal estará lista para realizar cualquier otra acción, por ejemplo, invocar al personaje en la escena. Así se vería este componente en el inspector:

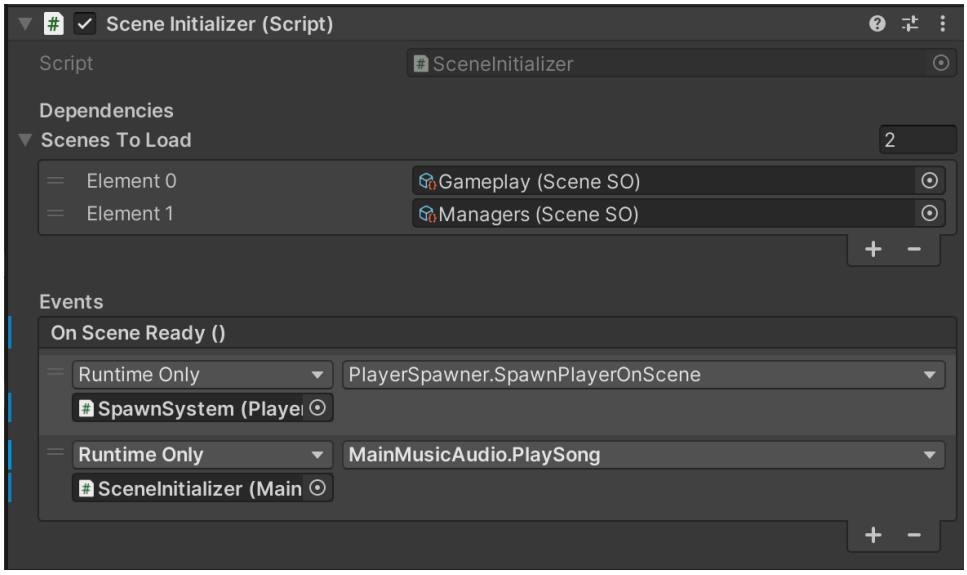


Figura 34: Componente SceneInitializer en el inspector de Unity

4.2.4. *Sprint* 4 (10 Mar – 23 Abr)

Posiblemente la cuarta iteración sea la más compleja en cuanto a código aunque, en realidad, se va a hacer uso de los patrones ya utilizados en anteriores semanas y, por lo tanto, algunos procesos serán más ágiles. El resultado esperado al final de *sprint* es un sistema de inventario y otro de misiones.

En cada uno de los sistemas hay dos frentes distintos que se pueden simplificar en lógica y UI. El juego va a funcionar con el sistema de lógica sin problema y no depende de una UI para subsistir, en cambio, aunque la UI tampoco requiere de la lógica para existir, necesita la inyección de algún tipo de dato. En el caso del inventario se crea el *InventorySO* y en el de las misiones el *BaseQuest*, ambos son SOs. Estos contienen la información necesaria para que la UI pueda mostrar datos, sin embargo, no se muestran hasta que no se envía una señal que active el sistema, por lo que en ningún caso se hace uso de métodos internos de Unity como el *Update* para actualizar información y, así, evitar sobrecargar el juego continuamente, ahora bien, la manera de realizar los envíos de información difiere entre cada uno.

En el inventario, primero de todo, el jugador debe recoger algún objeto de la escena, a continuación, se añade al *InventorySO* que guardará la información en una lista. El siguiente paso consiste en enviar un evento con la información del inventario hacia el *InventoryManager* en caso de necesitar ver los datos en la UI, pero no en otro caso. Este tipo de eventos es el mismo utilizado en el sistema de vida, la ventaja que proporciona este método es que *InventoryManager* no va a depender o no va a saber de la existencia del *InventorySO*. A partir de aquí, simplemente se irán mostrando los datos en pantalla. Otra manera posible de realizar esta conexión entre lógica y UI, sería añadiendo como variable el *InventorySO* en el *InventoryManager*, de esta manera no hace falta enviar ningún evento porque la información ya está ahí, sin embargo, de esa manera solo se puede utilizar un solo inventario, en el caso de querer tener varios inventarios es más conveniente enviar un evento con el inventario que se quiere mostrar en pantalla.

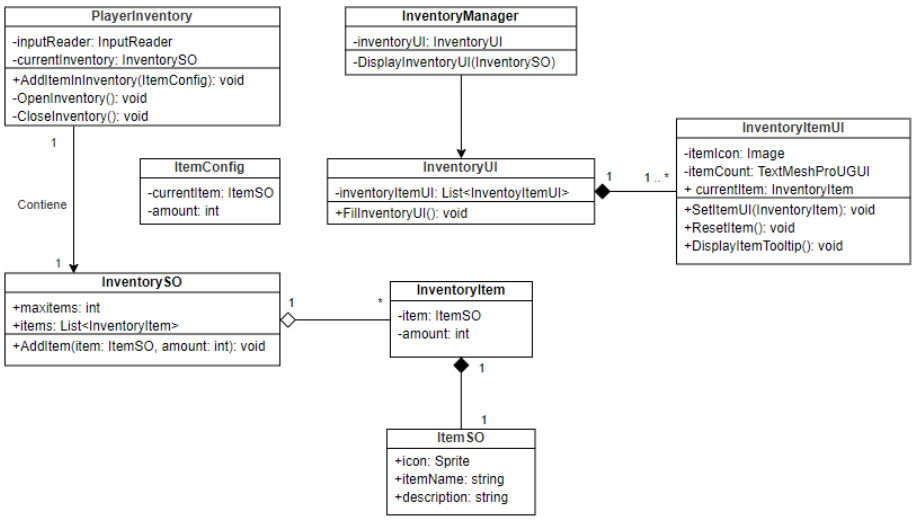


Figura 35: Diagrama de clases del sistema de inventario

En cuanto a las misiones, la característica diferencial es el uso de canales de comunicación como *Scriptable Objects* para enviar eventos por medio de *UnityActions* al igual que se hizo con el *InputSystem*. En esta ocasión, todo empieza en el *BaseQuest*, en realidad, esta es una clase abstracta porque se quieren crear diferentes tipos de misiones (recolección, matar, seguimiento …), por ahora, solo hay del tipo recolección, pero se podrían incluir más tipos más adelante gracias a la abstracción. También, es interesante mencionar que *BaseQuest* es un SO y, por lo tanto, cualquier cosa que herede de esa clase podrá ser un SO, en consecuencia, las misiones se podrán crear como *assets* en la ventana de proyecto y cualquier persona podrá crear misiones sin necesidad de tocar código.

Por otro lado, los eventos (*BaseQuestChannel* y *ItemCollectedChannel*) serán escuchados por el hijo del *BaseQuest* por dos motivos diferentes: cuando haya una interacción del personaje con la escena que actualice el estado de la misión (en el caso de recolección, cuando el jugador coja un objeto de la escena) y cuando se acepte o complete una misión en la interfaz. De nuevo, es importante que cada elemento haga solo lo que le corresponda y los eventos permiten justamente ese objetivo, enviando la información requerida en cada momento pero no dando más de lo que se necesita. A partir de aquí, sería posible añadir más tipos de misiones creando otra subclase de *BaseQuest* y otro canal de comunicación específico para esas misiones.

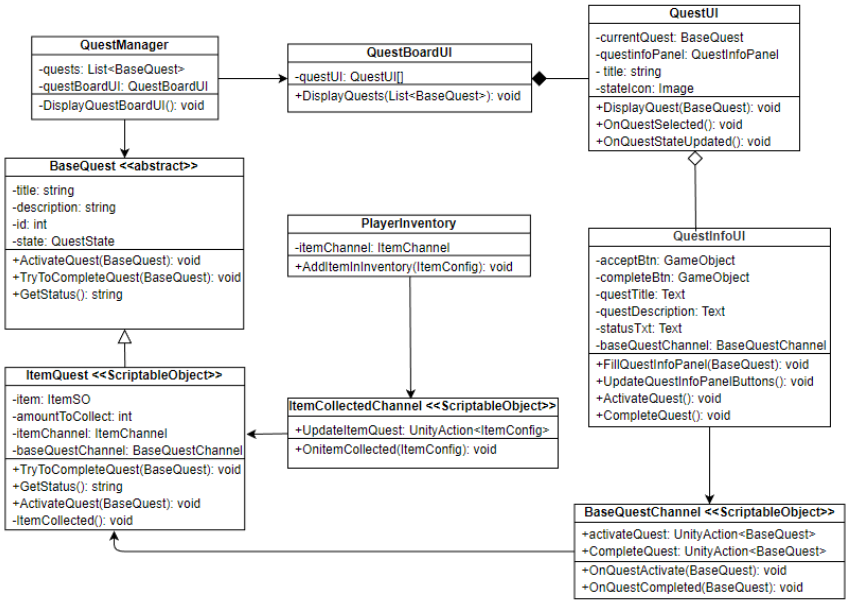


Figura 36: Diagrama de clases del sistema de misiones

4.2.5. *Sprint* 5 (24 Abr – 07 Abr)

Después de conseguir el objetivo en el anterior *sprint* es el momento de algo más relajado. Crear los escenarios y añadir música es el siguiente resultado esperado.

Teniendo en cuenta todo lo mencionado en el apartado de diseño de niveles, se empezará a dibujar un terreno con la herramienta Terrain y, a medida que va tomando forma se van a incluir modelos dentro del mapeado. La primera escena en ser construida es la aldea que tiene el siguiente aspecto una vez finalizada:

Figura 37: Escena de la aldea

De igual manera, la escena del bosque tendrá el siguiente aspecto:

Figura 38: Escena del bosque

Con todo, ahora es el momento de añadir un poco de ambiente con música y sonidos. Aunque parezca trivial la idea es, con un Object Pool, crear un sistema donde haya un límite de sonidos y estos se vayan activando cuando se requieran, además, hay que tener en cuenta la diferencia entre música y sonidos. El funcionamiento se puede apreciar en el siguiente diagrama:

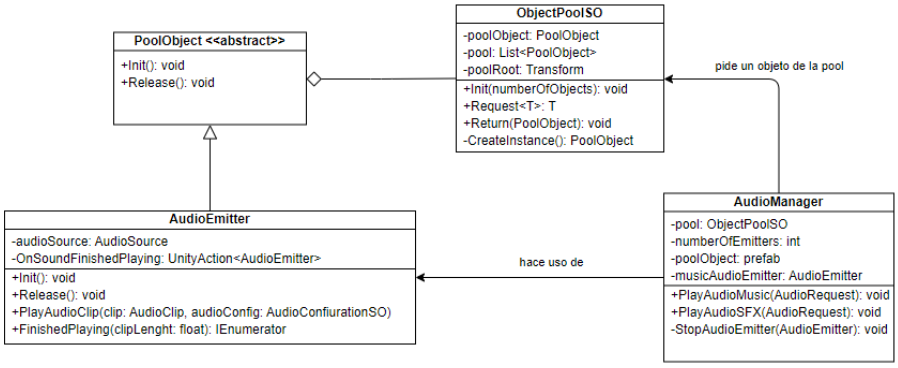


Figura 39: Diagrama de clases del patrón ObjectPool

Todo parte del *AudioManager*, que funciona de intermediario entre lo que ocurre en escena y la salida de sonido. Lo primero que hay que saber es que hay dos tipos de audios: la música y los efectos de sonido. Con esto en mente, cualquier elemento en escena que quiera hacer sonar un clip de audio deberá hacer uso de un canal de comunicación que avisará al *AudioManager* de la necesidad. Ese canal de comunicación es un evento que envía un *AudioRequest* y, la petición incluye el clip que se quiere hacer sonar y la configuración de audio.

Una vez recibida la señal, el *AudioManager* hará uso del *ScriptableObject* generador de *pools*, y solicitará que le proporcione de un *AudioEmitter*. *ObjectPoolSO* es un *script* genérico que permite crear *ObjectPools* con objetos del tipo *PoolObject*, que al ser una clase abstracta, es posible crear *pools* de cualquier tipo de objeto, en esta ocasión ese objeto es un *AudioEmitter* que

contiene toda la lógica para hacer sonar un *clip* o pararlo, además de colocar en escena un *AudioSource* que es el componente de Unity para reproducir *clips*.

Finalmente, todo el sistema está listo para hacer sonar tanto la música como los efectos de sonidos sin la necesidad de depender de múltiples *AudioSources* colocados en los objetos donde se quiere reproducir el sonido, así pues, todo queda concentrado en un mismo lugar y se puede configurar a base de eventos con el *AudioRequest*. Asimismo, se ha utilizado el patrón *object pool* que tiene como objetivo la reutilización de objetos para mejorar el rendimiento del videojuego, puesto que se reserva un espacio de memoria fijo que se reaprovecha constantemente. Visualmente se puede ver en la figura siguiente como en la jerarquía *AudioManager* aparecen *AudioEmitters* que se irán activando y desactivando según vayan entrando y saliendo los sonidos.

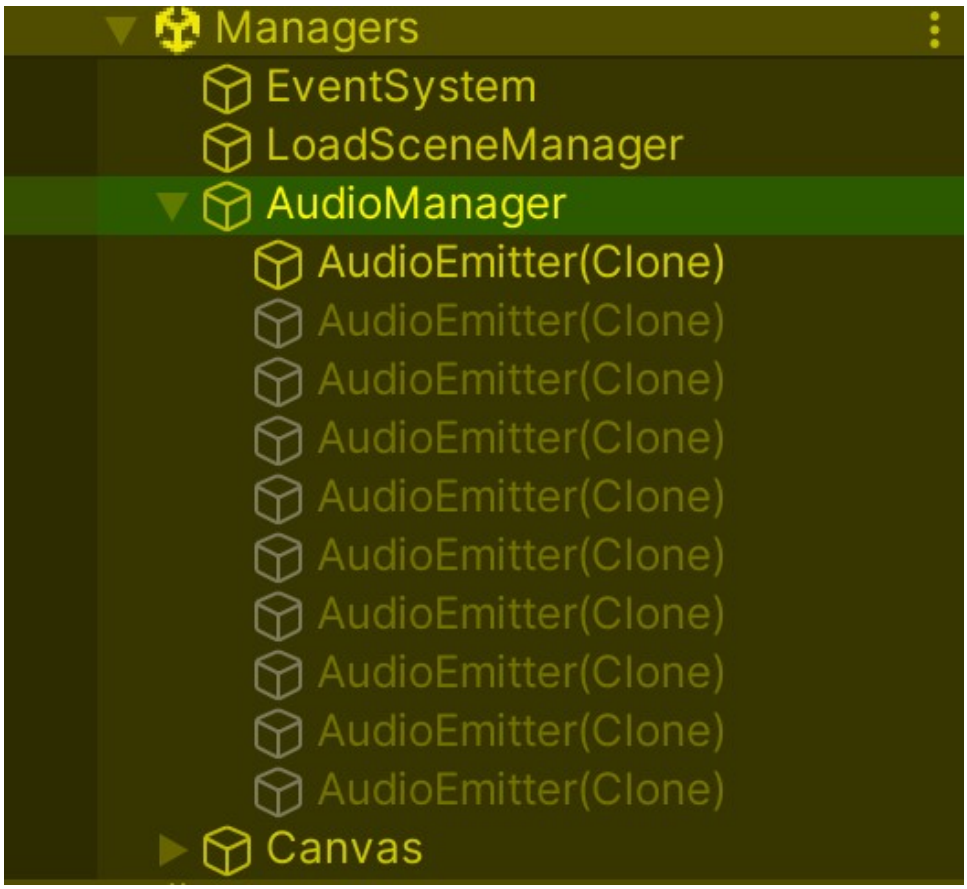


Figura 40: Jerarquía en la escena Managers mostrando los PoolObjects generados desde la pool

4.2.6. *Sprint* 6 (08 Abr – 21 Abr)

El juego ya es funcional en cuanto a mecánicas y escenarios, sin embargo, todavía no hay un menú de opciones o menú de pausa para poder configurar el juego cuando se desee o, simplemente, salir del juego. Este es el propósito en esta iteración, mejorar los menús y añadir opciones de configuración generales.

En primer lugar, la idea es mejorar la interfaz del menú. Esta interfaz se puede reaprovechar para el menú de pausa. Después, a partir de diferentes elementos de la UI, como botones, *dropdowns* o *sliders,* se podrán configurar las opciones, por ahora, serán cuatro elementos: vídeo, con la posibilidad de modificar la resolución; audio, pudiendo cambiar el nivel de los diferentes canales de sonido; controles, para mostrar los controles del juego y; por último, el idioma, con tres opciones disponibles.

Figura 41: Interfaz de opciones (vídeo)

Figura 42: Interfaz de opciones (controles)

Figura 43: Interfaz de opciones desde menú de pausa (sonido)

En la escena actual, donde se realiza la modificación, se enviaran los eventos correspondientes a los diferentes sistemas para que cambien la configuración. Para compartir los cambios entre escenas, se utiliza un *Scriptable Object* que guardará los datos a la espera de que una nueva se cargue y los inicialice.

4.2.7. *Sprint* 7 (22 Abr – 04 Jun)

Este es el último *sprint* y, como ya se anticipaba en la parte de alcance y riesgos, esta reservado a arreglar o mejorar aquellas partes que han quedado por pulir, esto quiere decir que no hay nada nuevo perteneciente a los objetivos marcados al principio. A continuación se van a listar todo el trabajo realizado durante este periodo:

- Se ha corregido un error de animación en la transición del estado Idle-Walk-Run en el que el personaje se mantenía en la animación Idle.
- Se ha arreglado un error donde la cámara comenzaba desde el suelo al iniciar una escena.
- Se ha corregido un error donde la resolución en el menú de opciones aparecía desplazada impidiendo su correcta visualización.
- Se ha arreglado un fallo en las misiones donde no era posible cambiar el estado de estas.
- Se ha creado un nuevo tipo de misiones que consisten en matar enemigos.
- Se han añadido los títulos y descripciones de las misiones.
- Se ha creado un filtro en la cámara para darle un toque más cinemático.
- Se ha substituido la tipografía de los textos que había por defecto para darle un toque más original.
- Se han pulido todos los efectos de sonido substituyendo o añadiendo nuevos sonidos al personaje y los enemigos.

5. Conclusión y resultados

Llegados al final se puede concluir que se han completado todos los objetivos propuestos desde un principio. Se ha conseguido crear un videojuego a partir una arquitectura que se basa en *Scriptable Objects* y no tan solo eso, sino que se han utilizado buenas prácticas que han permitido obtener un mejor resultado en cuanto a la programación.

Por otro lado, se puede destacar que al haber enfocado todo el trabajo en la programación, se han descuidado otros campos como la narrativa o el diseño de niveles y, en consecuencia, el juego no es entretenido o no tiene el objetivo de entretener. Entonces, mirando hacia el futuro, se podría construir el juego desde las bases del diseño o coger todas esas mecánicas y reutilizarlas en otros juegos puesto que son totalmente modulares y no existen dependencias entre diferentes sistemas gracias a la metodología utilizada.

Para concluir, este trabajo ha sido todo un desafío dado que se partía desde un conocimiento escaso sobre buenas prácticas y *Scriptable Objects*. Por esta razón, aunque el resultado final no hubiera sido el esperado, haber adquirido el suficiente conocimiento ya se podría considerar como un éxito.

6. Bibliografía y webgrafía

[1] Unity (11/2017). Unite Austin 2017 – Game Architecture with Scriptable Objects [Video]. Youtube. Disponible en: Link.

[2] Nystrom, R. Game Programming Patterns. 1ª edición. Genever Benning. Noviembre, 2014.

[3] Martin, R. Design Principles and Design Patterns [Internet]. 2000 [Consultado 04/2023]; Disponible en: Link.

[4] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. Octubre, 1994.

[5] Unity (12/2016). Unite 2016 – Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution [Video]. Youtube. Disponible en: Link.

[6] Unity. Introduction to Object Pooling [Internet]. 2016 [Consultado 04/2023]; Disponible en: Link.

[7] Unity. Tres Maneras Geniales de Diseñar Tu Juego con Scriptable Objects [Internet]. 2020 [Consultado 02/2023]; Disponible en: Link.

[8] Itch.io. Most Used Engines [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[9] Gamedatacruch. A.Doucet L. [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[10] Synty [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[11] Explosive [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[12] HONETi [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[13] OpenGameArt [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[14] Freesound [Internet]. 2023 [Consultado 05/2023]; Disponible en: Link.

[15] Unity AssetStore ScriptableObjectes-Architecture [Internet]. 2020 [Consultado 05/2023]; Disponible en: Link.