

Esta es una parte de un proyecto conjunto de dos estudiantes cuyo objetivo es abordar el punto de partida de dos desarrolladores independientes que incursionan en el ámbito de los videojuegos utilizando la plataforma Unity. Se pretende analizar en profundidad los aspectos que se han identificado como esenciales para el lanzamiento exitoso de cualquier juego al mercado, así como los desafíos inherentes a dicho proceso y la forma en que han sido solventados.

El presente estudio abarca de manera exhaustiva la implementación de sistemas fundamentales, abordando áreas clave como el sistema de audio, la entrada de controladores, la gestión de escenas, el guardado de progresos, el control de la cámara y la interfaz de usuario. Además, se ha desarrollado un framework de navegación de menús que agiliza la creación de interfaces de usuario en Unity.

La implementación y validación de todas estas estructuras y sistemas se han llevado a cabo en la creación de nuestro propio videojuego, Carnem Levare. Este juego no solo consolida los conocimientos adquiridos, sino que también ejemplifica la aplicación práctica de las soluciones propuestas. En él se evidencian las ventajas inherentes a nuestras implementaciones, así como las posibles mejoras a considerar en futuros desarrollos.



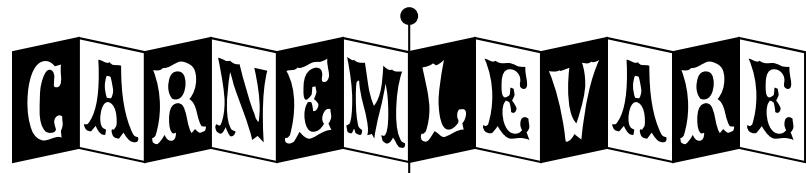


ETSIIT
Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación



GRADO EN
INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado



**“Estructuras y sistemas fundamentales de un
videojuego”**

CURSO ACADÉMICO: 2022/2023

Autor:

Alejandro Cruz Lemos

Tutor:

Pablo García Sanchez



Alejandro Cruz Lemos y Pablo García Sanchez

Este trabajo está bajo la licencia Creative Commons Attribution-ShareAlike 4.0 Internacional (CC BY-SA 4.0). Este es un resumen legible por humanos de (y no un sustituto para) la licencia. Tienes la libertad de:

Compartir Copiar y redistribuir el material en cualquier medio o formato.

Adaptar Adaptar, transformar y construir sobre el material.

El licenciante no puede revocar estas libertades mientras sigas los términos de la licencia:



Atribución: Debes dar el crédito apropiado, proporcionar un enlace a la licencia e indicar si se hicieron cambios. Puedes hacerlo de manera razonable, pero no de ninguna manera que sugiera que el licenciante te respalda a ti o tu uso.



Compartir igual: Si adaptas, transformas o construyes sobre el material, debes distribuir tus contribuciones bajo la misma licencia que el original.

Para ver una copia completa de esta licencia, visita <https://creativecommons.org/licenses/by-sa/4.0>



Este documento ha sido generado utilizando **Alcázar**, una plantilla L^AT_EX gratuita y de código abierto para trabajos académicos creada por [Juan Del Pino Mena](#).

EQUIPO DE DESARROLLO

Alejandro Cruz Lemos



Maestro del arte computacional, tiene una relación peculiar con la guitarra (aunque la guitarra podría no estar de acuerdo), una pasión inusual por el helado de pistacho, una maestría en convertir las pequeñas cosas en situaciones estresantes y un talento innato para cazar y destrozar errores de programación. Se rumorea que es uno de los cerebros detrás de la famosa desarrolladora de videojuegos Parry Mechanics.



Guillermo García Arredondo



Mago de los algoritmos, disfruta devorando un tazón de pasta carbonara a las tres de la mañana, desafiándose con calistenia al fallo, da vida a los relatos de fantasía más envolventes y encuentra deleite en las notas de Paco de Lucía. En sus propias palabras: “¿Parry Mechanics? No tengo ni la más mínima idea de a qué te refieres”.



Paula Cruz Lemos



Maestra del arte conceptual, creadora de portadas para trabajos de fin de grado que podrían inspirar a los envidiosos de la Sixtina. Durante el día, duerme como si estuviera practicando para las Olimpiadas del sueño, y por la noche... bueno, quién sabe si está descubriendo los misterios del cosmos o tejiendo planes para dominar el mundo. Según fuentes secretas, forma parte de Parry Mechanics, aunque incluso ella podría tener dudas al respecto.



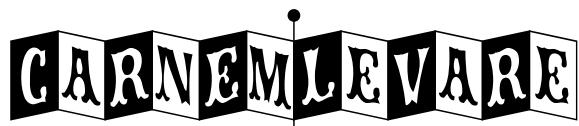
Cita este trabajo:

```
1 @mastersthesis{citeKey,
2   author = "Alejandro Cruz Lemos and Pablo García Sanchez",
3   title = "Estructuras y sistemas fundamentales de un videojuego",
4   school = "Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación",
5   year = "2023",
6   month = "Agosto",
7   address = "C. Periodista Daniel Saucedo Aranda, s/n, 18014 Granada"
8 }
```

D. Pablo García Sanchez, Profesor del departamento de Ingeniería de Computadores, Automática y Robótica de la Universidad de Granada,

Informa:

Que el presente trabajo, titulado:



**Estructuras y sistemas fundamentales de un
videojuego**

ha sido realizado bajo mi supervisión por **Alejandro Cruz Lemos**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a septiembre de 2023.

Fdo. Prof. Pablo García Sanchez

Fdo. Alejandro Cruz Lemos

Estructuras y sistemas fundamentales de un videojuego

PALABRAS CLAVE:

Software libre, Unity, desarrollo de videojuegos, desarrollo de software, marco de trabajo, audio adaptativo, patrón de diseño, interfaz de usuario

RESUMEN:

Este proyecto tiene como objetivo abordar el punto de partida de dos desarrolladores independientes que incursionan en el ámbito de los videojuegos utilizando la plataforma Unity. Se pretende analizar en profundidad los aspectos que se han identificado como esenciales para el lanzamiento exitoso de cualquier juego al mercado, así como los desafíos inherentes a dicho proceso y la forma en que han sido solventados.

El presente estudio abarca de manera exhaustiva la implementación de sistemas fundamentales, abordando áreas clave como el sistema de audio, la entrada de controladores, la gestión de escenas, el guardado de progresos, el control de la cámara y la interfaz de usuario. Estos componentes se presentan en el marco de una arquitectura de software diseñada para simplificar su gestión, adaptación y aplicabilidad en diversos proyectos de desarrollo Unity. Además, se ha desarrollado un *framework* de navegación de menús que agiliza la creación de interfaces de usuario en Unity, introduciendo una capa de abstracción entre la interfaz visual y su lógica subyacente.

La implementación y validación de todas estas estructuras y sistemas se han llevado a cabo en la creación de nuestro propio videojuego, Carnem Levare. Este juego no solo consolida los conocimientos adquiridos, sino que también ejemplifica la aplicación práctica de las soluciones propuestas. A través de esta aplicación, se evidencian las ventajas inherentes a nuestras implementaciones, así como las posibles mejoras a considerar en futuros desarrollos.

La otra cara de este proyecto se manifiesta en el trabajo realizado por mi compañero, cuyo propósito es enfocarse en el diseño intrínseco de Carnem Levare, incluyendo su jugabilidad, gestión de animaciones, inteligencia artificial de los enemigos y otros aspectos que contribuyen a la singularidad de nuestro videojuego.

Foundation and Essential Systems of a Video Game

KEYWORDS:

Open source, Unity, game development, software development, framework, adaptive audio, design pattern, user interface

ABSTRACT:

This project endeavors to explore the initial journey undertaken by two independent developers as they immerse themselves in the realm of video games using the Unity platform. The primary goal is to delve deeply into the critical elements required for the successful launch of any game into the market. This includes a comprehensive examination of the challenges that arise during this process and the innovative solutions devised to conquer them.

The scope of this study encompasses an in-depth exploration of the implementation of foundational systems. These systems encompass crucial domains such as audio mechanics, controller inputs, scene management, progress preservation, camera dynamics, and user interfaces. These components are seamlessly integrated within a thoughtfully designed software architecture, tailored to streamline their management, adaptation, and relevance across a diverse spectrum of Unity-based development projects. Moreover, an intuitive menu navigation *framework* has been crafted to enhance the creation of user interfaces within Unity, introducing an abstraction layer that bridges the visual interface with its underlying logic.

The deployment and validation of these intricate structures and systems have been meticulously executed in the creation of our very own video game, Carnem Levare. This game serves not only to solidify the assimilated knowledge but also as a tangible manifestation of the practical implementation of the proposed solutions. Through this application, the inherent benefits of our devised systems are vividly illustrated, accompanied by insights into potential enhancements for forthcoming developments.

The other side of this project is reflected in the work carried out by my partner, whose purpose is to focus on the intrinsic design of Carnem Levare, including its gameplay, animation management, enemy artificial intelligence, and other aspects that contribute to the uniqueness of our video game.

*“Un mundo crece en
torno a mí. ¿Le estoy
dando forma, o son
sus contornos
predeterminados lo
que guía mi mano?”*

Doctor Manhattan

Alan Moore
Watchmen, 1986



AGRADECIMIENTOS

Quiero dedicar estos párrafos a las personas que han sido fundamentales en la construcción de este trabajo tal y como lo presento hoy.

A mis padres, quienes han sido mi firme apoyo en esta travesía de adentrarme en el mundo del desarrollo de videojuegos, conscientes de que, al igual que cualquier expresión artística, es un terreno complejo en el que iniciar. Mi hermana, con quien debato todas las ideas que me asaltan y que aporta siempre una perspectiva distinta y renovadora, que me lleva a cuestionar si estoy siguiendo el rumbo adecuado.

A mi compañero de trabajo, con quien he mantenido conversaciones prácticamente a diario acerca de este proyecto, y con quien he enfrentado todos los desafíos y obstáculos que el trabajo nos ha presentado. Espero continuar colaborando con él para llevar a cabo el lanzamiento de este videojuego y muchos más en el futuro.

A mis amigos, por brindarme un espacio donde desconectar y relajarme, y a mi tutor de trabajo, por sus valiosos consejos y por señalar los errores en mis argumentos y escritura. Por último, quiero expresar mi agradecimiento a todas las personas en internet que, sin buscar beneficios económicos, comparten información que agiliza el proceso de aprendizaje. Su generosidad ha sido crucial para avanzar más rápidamente en este camino.

ACKNOWLEDGEMENTS

I'd like to extend these paragraphs as a heartfelt tribute to the individuals who have played an indispensable role in shaping this work into what it is today.

To my parents, whose unwavering support has been a guiding light as I ventured into the intricate world of game development; a field that, much like any artistic pursuit, presents its unique set of challenges at the outset. My sister, a constant companion in my creative explorations, brings her own fresh and distinct perspective to every idea we discuss, prompting me to continuously evaluate my chosen path.

My dedicated colleague at work has been my daily sounding board throughout this project, helping me navigate through the myriad issues and hurdles that naturally arise. I look forward to our ongoing collaboration, as we endeavor to not only release this video game but also embark on numerous future projects.

To my friends, for providing a space where I can unwind and disconnect. And to my thesis advisor, for their invaluable advice and for pointing out errors in my arguments and writing. Lastly, I want to extend gratitude to all the individuals online who share information selflessly, speeding up the learning process.

Índice general

Acerca de este trabajo	III
Autorización de la defensa	v
Resumen	vii
Dedicatoria	xii
Agradecimientos	xiii
Tabla de contenidos	xviii
Lista de Figuras	xx
Lista de Tablas	xxi
Lista de Fragmentos de Código	xxiii
Glosario	xxv
1. Introducción	1
1.1. Presentación	1
1.2. Motivación	1
1.3. Objetivos	2
1.4. Estructura de la memoria	3
2. Estado del arte	5
2.1. Audio adaptativo	5
2.2. Sistema de Inputs	6
2.3. Cámara	7
2.4. Interfaz de usuario	8
2.5. Metodologías de desarrollo de videojuegos	8
2.6. Conclusiones	9
3. Planificación	11
3.1. Introducción	11
3.2. Metodología	11
3.2.1. Gestión del Tiempo	11
3.2.2. Comunicación	12
3.2.3. Selección de objetivos	12
3.2.4. Metodología ágil y Scrum	13
3.2.5. Control de versiones	14
3.3. Seguimiento del desarrollo	14
3.3.1. Etapas del proyecto	15
3.3.1.1. Visión global	17
3.3.1.2. Etapas	19
3.3.1.3. Gestión de tareas (<i>Issues</i>)	20

3.4. Análisis	20
4. Estructuras fundamentales	23
4.1. Introducción	23
4.2. Detalles de la implementación	23
4.3. Arquitectura software	23
4.3.1. Patrón de diseño Fachada	24
4.3.2. <i>ScriptableObject</i> como comunicador	26
4.3.3. Breve introducción a eventos de C#	27
4.4. Sistema de Inputs	27
4.4.1. Breve introducción al <i>InputSystem</i> de Unity	30
4.4.2. Definición de eventos	30
4.4.3. Representación de <i>inputs</i>	32
4.4.4. Reasignación de <i>inputs</i>	33
4.4.5. Vibración	34
4.5. Sistema de audio	34
4.5.1. Breve introducción a <i> AudioSource</i> de Unity	37
4.5.2. Estructura del sistema	37
4.5.2.1. <i>Sound</i> y <i>SoundGroup</i>	38
4.5.2.2. Estructura de datos (tabla <i>hash</i>)	39
4.5.3. Controlador del sistema	40
4.5.4. Conclusiones de la implementación, ventajas e inconvenientes	40
4.6. Sistema de guardado	41
4.6.1. Estructura y funcionamiento del sistema	45
4.6.2. Estructura de guardado local	45
4.6.3. Guardado por defecto	47
4.6.4. Serialización XML (guardado permanente)	47
4.7. Gestión de escenas	49
4.7.1. Breve introducción a <i> SceneManager</i> de Unity	51
4.7.2. Configuración de escenas	51
4.7.3. Cargador de escenas y transiciones	51
4.7.3.1. Gestión y Control de Transiciones Animadas	52
4.7.4. Controlador del sistema	54
4.8. Controlador de cámara	54
4.8.1. Uso de <i>MonoBehaviour</i> y <i>Prefabs</i>	57
4.8.2. Breve introducción a <i>Cinemachine</i> de Unity	57
4.8.3. Componentes y funcionamiento	58
4.8.4. Definición de cámaras	58
4.8.5. Efectos de cámara	59
4.8.6. Controlador	60
4.9. Sistema de interfaz de usuario	61
4.9.1. Uso de <i>MonoBehaviour</i> y <i>Prefabs</i>	63
4.9.2. Acercamiento a <i>Unity UI</i>	63
4.9.3. Estructura y funcionamiento	63
4.9.4. MVC para la interconexión de objetos interactivos	64
4.9.5. Acercamiento al framework de navegación de menús	65

4.9.6.	MVC para la navegabilidad del sistema	66
4.9.7.	Vínculo con el Sistema de Inputs	67
4.10.	Conclusiones	68
5.	Framework de creación de menús	69
5.1.	Introducción	69
5.2.	Motivación	69
5.3.	Idea y fundamentos del marco de trabajo	70
5.3.1.	Problema y Desafíos en la Navegabilidad de Menús	70
5.3.2.	Propuesta: Creación de un Editor de Árboles Lógicos	71
5.3.2.1.	Tipos de nodos	72
5.3.2.2.	Funcionamiento	73
5.3.2.3.	Lógica subyacente	74
5.4.	Estructura software	75
5.5.	Estructura del árbol	75
5.5.1.	Estructura abstracta	75
5.5.1.1.	Uso de <i>ScriptableObject</i> y creación de asociaciones	78
5.5.1.2.	Asignación de ID	78
5.5.2.	Árbol de menús	79
5.5.2.1.	Creación de caminos	79
5.6.	Editor de árbol de navegación de menús	80
5.6.1.	Breve introducción a la creación de editores en Unity	82
5.6.2.	Detalles de la implementación	83
5.6.3.	Diagrama de clases y comunicación	83
5.6.4.	Creación de la ventana del editor y acceso directo	85
5.6.5.	Uso de <i>UxmlFactory</i> y <i>UxmlTraits</i>	86
5.6.6.	Creación de la vista del inspector	87
5.6.7.	Creación de la vista del árbol	89
5.6.7.1.	Creación de la vista del nodo	89
5.6.7.2.	Integración de <i>NodeView</i> en la vista del árbol	90
5.6.7.3.	Creación de la vista del enlace	92
5.6.7.4.	Integración de <i>EdgeView</i> en la vista del árbol	92
5.6.7.5.	Comunicación durante la ejecución: Nodo seleccionado	94
5.7.	Conclusiones	95
6.	Caso de uso: Carnem Levare	97
6.1.	Acerca de Carnem Levare	97
6.1.1.	Un Vistazo a la Temática	97
6.1.2.	Explorando el Género	97
6.1.3.	Aspectos de la implementación actual	98
6.2.	Sistema de Inputs	99
6.2.1.	Estructura	99
6.2.2.	<i>InputSystem/InputReader</i>	99
6.2.3.	Vibración/Reasignación de inputs	100
6.3.	Sistema de Audio	100
6.3.1.	Estructura	100
6.3.2.	Controlador y acceso	102

6.3.2.1.	Benchmark y posible solución	103
6.4.	Sistema de Guardado	104
6.4.1.	Estructura y funcionamiento	104
6.4.1.1.	Guardado de Inputs	105
6.4.1.2.	Aplicar los cambios de opciones	106
6.5.	Gestión de escenas	106
6.5.1.	Estructura y funcionamiento	107
6.5.2.	Objetos permanentes entre escenas	108
6.6.	Controlador de cámara	109
6.6.1.	Uso de <i>Cinemachine</i>	110
6.6.2.	Cámara de combate	110
6.6.2.1.	Inicialización de objetivos (Targeting)	110
6.6.2.2.	Apuntado y seguimiento	111
6.6.2.2.1.	Problema de seguimiento	111
6.6.2.2.2.	Apuntado mediante <i>CinemachineTargetGroup</i>	111
6.6.2.3.	Orbital Transposer	112
6.6.2.4.	Efectos de cámara	114
6.6.2.4.1.	Alineación de cámara orbital	114
6.6.2.4.2.	Movimiento Lineal	114
6.6.2.4.3.	Suavizado de seguimiento	115
6.6.2.4.4.	Ruido de cámara	115
6.6.2.4.5.	Vibración de cámara (<i>Camera Shake</i>)	115
6.6.2.4.6.	Cambio de target	115
6.6.2.4.7.	Hitstop	115
6.6.3.	Cámara frontal	116
6.6.4.	Conclusiones y posibles mejoras	116
6.7.	Sistema de interfaz de usuario	117
6.7.1.	Menú principal	117
6.7.1.1.	Diseño	117
6.7.1.2.	Navegabilidad	118
6.7.1.3.	Asignación de funcionalidades	120
6.7.1.3.1.	Initial Menu	120
6.7.1.3.2.	Selectable Options Menu	120
6.7.1.3.3.	Sound Menu	120
6.7.1.3.4.	Controls Menu	121
6.7.1.3.5.	Visuals Menu	121
6.7.1.4.	Asignación de inputs	121
6.7.2.	Menú de Pausa	122
6.7.3.	Conclusiones y posibles mejoras	122
7.	Conclusiones y trabajos futuros	125
7.1.	Conclusiones	125
7.2.	Perspectivas Futuras	125
Bibliografía		127

Lista de Figuras

2.1. Recreación del videojuego Pong, 1972	5
2.2. Super Mario Odissey, 2017	6
2.3. Presentación oficial de Project Leonardo para PlayStation 5	7
2.4. Rumble Pack para Nintendo 64	7
2.5. Menú de reasignación de inputs en The Last of Us Part 2	8
2.6. Esquema de funcionamiento DevOps	9
3.1. Tareas actuales del proyecto de Github	13
3.2. Esquema de metodología ágil y en cascada	14
3.3. Etapas de desarrollo del proyecto	16
3.4. Commits realizados por cada día de desarrollo	17
3.5. Línea de tiempo de ramas y <i>Pull Requests</i>	18
3.6. Issues cerradas durante el proyecto	21
4.1. Estructura software completa	24
4.2. Ejemplo de uso del patrón de diseño Fachada	25
4.3. Ejemplo del principio de Segregación de interfaces	26
4.4. Esquema de funcionamiento del patrón Observador	27
4.5. Diagrama de clases del Sistema de Inputs	29
4.6. Interfaz de <i>Input Action Asset</i>	30
4.7. Esquema de funcionamiento <i>Input Reader</i>	31
4.8. Ejemplo de uso de <i>Player Input</i>	31
4.9. Fuentes usadas para la representación de <i>inputs</i>	32
4.10. Diagrama de comunicación para representación de inputs	33
4.11. Esquema de funcionamiento de funciones async	34
4.12. Diagrama de clases del Sistema de Audio	36
4.13. Estructura del sistema de audio	38
4.14. Gráficas de rendimiento (diccionario vs tabla <i>hash</i>)	39
4.15. Ejemplo de estructura de sistema de audio	42
4.16. Diagrama de clases del sistema de guardado	44
4.17. Diagrama de flujo del sistema de guardado	46
4.18. Ejemplo de Principio de inversión de la dependencia	47
4.19. Diagrama de clases del controlador de escenas	50
4.20. Ejemplo de inspector de SceneLogic	52
4.21. Diagramas de flujo del controlador de escenas	53
4.22. Diagrama de clases del controlador de cámaras	56
4.23. Diagrama de clases del sistema de interfaz de usuario	62
4.24. Esquema de MVC para la interconexión de objetos interactivos	65
4.25. Ejemplo de árbol de navegación de menús	66
4.26. Esquema de MVC para la navegabilidad del sistema	67
5.1. Menú de inventario y equipamiento de Baldur's Gate 3	70

5.2.	Árbol lógico de navegabilidad para menú de inventario y equipamiento de Baldur's Gate 3	71
5.3.	Camino completamente visualizable del menú de Baldur's Gate 3	72
5.4.	Camino no completamente visualizable del menú de Baldur's Gate 3	72
5.5.	Posibles acciones disponibles del menú de Baldur's Gate 3	73
5.6.	Estructura y dependencias en el framework	75
5.7.	Diagrama de clases del la estructura abstracta del árbol	76
5.8.	Ejemplo de aplicación del patrón Composite	77
5.9.	Multifunción de <i>ScriptableObject</i> como estructura del árbol	78
5.10.	Ejemplo de búsqueda en profundidad	79
5.11.	Estructura del árbol de navegabilidad de menús	79
5.12.	Contenido de la pila en una navegabilidad al hijo con nodos estáticos	81
5.13.	Contenido de la pila en una navegabilidad a la derecha	81
5.14.	Diagrama de clases de los <i>scripts</i> del editor	84
5.15.	Jerarquía del editor en <i>UI Builder</i>	86
5.16.	Diagrama de comunicación del botón “UPDATE” de <i>InspectorView</i>	88
5.17.	Diagrama de comunicación tras seleccionar un nodo en el editor.	88
5.18.	Ventana del editor con la vista de árbol por defecto.	89
5.19.	Creación de nodos dentro y fuera del editor	92
5.20.	Comparativa entre enlaces con y sin diferencia de color en asignación	93
5.21.	Representación de nodos seleccionados en ejecución	95
6.1.	Definición de <i>UISoundEffects</i>	101
6.2.	Benchmark de búsqueda mediante etiquetas	103
6.3.	Definición de <i>SceneLogics</i>	107
6.4.	Objeto de seguimiento de jugador	112
6.5.	Posición de <i>CinemachineTargetGroup</i>	112
6.6.	<i>Orbital Transposer</i>	113
6.7.	Uso de Orbital Transposer para encuadre de jugador y enemigo	113
6.8.	Curva cúbica de Bézier	114
6.9.	Cámara frontal	116
6.10.	Menú de inicio de partida	118
6.11.	Menú seleccionable para configuración	119
6.12.	Árbol lógico de menú principal	120

Lista de Tablas

4.1. Ventajas e inconvenientes (Estructura del sistema de audio)	41
4.2. XML vs JSON	48
6.1. Benchmark de búsqueda mediante etiquetas	103

Lista de Fragmentos de Código

4.1. Ejemplo de código del menú de pausa	31
4.2. Ejemplo de equivalencias entre fuente y <i>action</i> (Mando)	32
4.3. Ejemplo de equivalencias entre fuente y <i>action</i> (Teclado)	32
4.4. Carga de escenas inmediata	54
4.5. Carga de escenas con pantalla de carga	54
5.1. Código de la función <i>CreateGUI</i>	85
5.2. Código de la función <i>OnSelectionChange</i> de <i>TreeEditor</i>	85
5.3. Código de la función <i>OnOpenAsset</i> de <i>TreeEditor</i>	86
5.4. Código de la clase <i>SplitView</i>	86
5.5. Código de la función <i>UpdateSelection</i> de <i>InspectorView</i>	87
5.6. Código del constructor de la vista del árbol	89
5.7. Código del constructor de la vista del nodo	89
5.8. Código de creación de nodos en <i>TreeView</i>	90
5.9. Código de la función <i>OnGraphViewChange</i> de <i>TreeView</i>	91
5.10. Código de la función <i>PopulateView</i> de <i>TreeView</i>	91
5.11. Integrar la vista del enlace en <i>PopulateView</i>	92
5.12. Integrar la vista del enlace en <i>OnGraphViewChanged</i>	94
6.1. Archivo de guardado de opciones XML	105
6.2. Guardado de <i>inputs</i> mediante JSON	105

Glosario

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [I](#) | [M](#) | [N](#) | [P](#) | [S](#)

A

AAA Es una clasificación informal utilizada para los videojuegos producidos y distribuidos por una distribuidora importante o editor importante, típicamente teniendo marketing y desarrollo de alto presupuesto [1].

acoplamiento Es el grado en que los módulos de un programa dependen unos de otros.

async Las funciones async en C# tienen muchas utilidades aunque en este proyecto se van a usar sobre todo para esperar un tiempo determinado o a cierta condición antes de ejecutar alguna funcionalidad. Las funciones async como tal están dentro de la main thread del juego, para poder lanzar realmente algo asíncrono o en background debemos usar la palabra clave await.

atributo Son marcadores que se pueden colocar encima de una clase, propiedad o función en un *script* para indicar un comportamiento especial [2].

await Se usa para esperar en background (fuera de la main thread) a alguna task o tiempo determinado.

B

build Es una versión ejecutable en el sistema operativo del código del juego.

C

ciclo del juego Es una secuencia de procesos que se ejecutan continuamente mientras el juego esté en marcha [3].

cohesión En ingeniería del software, algo tiene alta cohesión si tiene un alcance definido, unos límites claros y un contenido delimitado y perfectamente ubicado.

D

diccionario En C# es una colección que nos permite almacenar datos de forma clave-valor.

E

escena En Unity el diseño del juego se divide en archivos llamados escenas para poder separar los

distintos niveles, escenarios o menús de nuestro juego.

F

framework Un entorno de trabajo (del inglés framework) o marco de trabajo es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar [4].

frontend Es el componente del sistema que se encuentra disponible al público.

G

game feel Se trata de la sensación que nos brinda un juego cuando interactuamos con éste (El término fue popularizado por Steve Wink en su libro Game Feel: A Game Designer's Guide to Virtual Sensation).

I

inspector Ventana en Unity que sirve para ver y editar propiedades y configuraciones para casi todo en el Editor, incluidos GameObjects, componentes de Unity, recursos, materiales y configuraciones o preferencias.

M

MonoBehaviour Cualquier script que herede de esta clase entrará en el bucle del juego, es decir, Unity ejecutará automáticamente las funciones de *MonoBehaviour* definidas por el motor gráfico. Funciones que podrán redefinirse en el *script* para optar a un comportamiento específico dentro del bucle.

multiplicidad La multiplicidad de extremo de asociación define el número de instancias de tipo de entidad que puede haber en un extremo de una asociación [5].

N

namespace Palabra clave que se usa para declarar un ámbito que contiene un conjunto de objetos relacionados.

P

prefab tipo de asset que le permite almacenar un objeto GameObject completamente con components y propiedades. El prefab actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un prefab asset será inmediatamente reflejado en todas las instancias producidas por él, pero, también se puede anular components y ajustes para cada instancia individualmente [6]. plural

S

ScriptableObject Es un contenedor de datos que puede usar para guardar grandes cantidades de información, independientemente de las instancias de clase. Uno de los principales casos de uso es reducir el uso de memoria de su proyecto al evitar copias de valores.

StyleSheet Estas se aplican a elementos visuales para controlar el diseño y la apariencia visual de la interfaz de usuario [7].

Introducción

1.1. Presentación

Los videojuegos son actualmente una de las mayores industrias del entretenimiento y desempeñan un papel fundamental en el desarrollo de software[8]. Su creación no es algo trivial, no solo hay que tener en cuenta los aspectos más concretos del ámbito como el diseño, la jugabilidad, el arte o el mercado, sino que estas ideas hay que materializarlas como algo jugable. La creación y gestión de un sistema de audio, cámara, *inputs* (entradas de un teclado/ratón, un mando o cualquier otro dispositivo), niveles, interfaz, generación de enemigos, IA y un largo etcétera son la base que da forma y define un videojuego.

En este proyecto, desarrollaremos algunos de estos sistemas. Además, hemos dedicado mucho tiempo a la creación de un framework para Unity, que nos permite diseñar los menús de nuestro juego con relativa facilidad. Nuestro objetivo es que todos los sistemas desarrollados sean lo más flexibles y escalables posible, de manera que puedan utilizarse en futuros proyectos sin tener que reinventar la rueda.

Buscamos que la implementación de todos estos sistemas consolide nuestro juego como un producto de calidad y que cumpla con los estándares de la industria actual. Deseamos que esta memoria sirva como un resumen autoexplicativo de los problemas y soluciones que hemos encontrado durante el desarrollo, de modo que aquellos que se inician como desarrolladores de videojuegos puedan tener una idea de cómo abordar estos sistemas, que a veces pueden resultar un tanto engorrosos y consumir tiempo que podríamos destinar a la parte más específica del juego que deseamos desarrollar.

Este proyecto es el resultado de la colaboración entre dos estudiantes. Cada uno ha realizado un Trabajo de Fin de Grado con el propósito de dividir la tarea de desarrollo de un videojuego en dos secciones claramente definidas. Una de estas secciones, la que ya se ha mencionado y en la que se centra este proyecto, aborda aspectos generales y cuestiones esenciales que son comunes a cualquier videojuego, independientemente de su género o temática. La otra sección se enfoca en aspectos específicos del videojuego en desarrollo, con un enfoque particular en la creación de un juego de lucha y en todos los elementos asociados a esta temática.

Nota: aunque se utiliza la primera persona del plural, este trabajo engloba mi contribución personal a menos que se indique lo contrario (por ejemplo, herramientas de desarrollo colaborativas).

Hemos decidido utilizar Unity como motor gráfico y herramienta de desarrollo para agilizar nuestro trabajo, junto con Visual Studio Code como editor de código y GitHub para el control de versiones.

1.2. Motivación

El desarrollo de videojuegos actualmente supone una ardua tarea debido al nivel de exigencia que se está llegando en el medio, dando como resultado desarrollos muy extensos de hasta cuatro

o cinco años en videojuegos *AAA*. Las grandes desarrolladoras cuentan con numerosos recursos, como equipos de trabajo, tiempo, financiamiento y, lo más importante, *frameworks* dentro de sus motores que facilitan enormemente la creación de nuevos juegos, especialmente secuelas o precuelas de franquicias existentes. En nuestro caso, motores gratuitos como Unity o Unreal Engine brindan una ayuda significativa, pero no son suficientes por sí solos para satisfacer todas las necesidades de desarrollo.

Con este proyecto se pretende que podamos adentrarnos en la industria del videojuego como desarrolladores independientes, dándonos las herramientas suficientes para el desarrollo de este y posteriores videojuegos. A nivel de implementación la creación de nuestros propios sistemas hace que expandir su contenido y solventar fallos sea algo mucho más cómodo. Además, nos permite adecuar dichos sistemas a videojuegos específicos, haciéndolos más camaleónicos. De hecho, en la tienda oficial de Unity existen numerosos *frameworks* de pago que podrían ahorrarnos mucho tiempo en proyectos pequeños, pero a largo plazo, adoptar este enfoque nos proporciona un mayor bagaje en el desarrollo de videojuegos. A su vez esta memoria se puede utilizar como herramienta de estudio para que otras personas puedan evitar los errores que hemos cometido en el desarrollo y enriquecer las ideas que hemos aportado en nuestra implementación.

En el ámbito personal este proyecto me ha permitido conocer patrones y principios que eliminan diseños deficientes, creando un código legible y extensible. He fortalecido mis habilidades en el campo del desarrollo de software y ampliado mis conocimientos en programación orientada a objetos y diseño de videojuegos. Además, como equipo, nos permite que el desarrollo de futuros videojuegos se enfoque más en los aspectos creativos y de diseño, dejando de lado estos aspectos más generales y, por lo tanto, agilizando su creación.

1.3. Objetivos

El objetivo principal de este proyecto es la creación de la estructura software que cimenta a cualquier videojuego comercial, independientemente del género o la temática que trate. Esta estructura está compuesta por los sistemas o funcionalidades típicas del desarrollo de un videojuego y por un esqueleto que los engloba. Se proponen los siguientes sub-objetivos:

1. Creación de un sistema de interfaz que agilice el trabajo y no limite a los diseñadores.
2. Creación de un sistema de *inputs* que facilite el uso de cualquier periférico y su integración con cualquier juego.
3. Creación de un gestor de escenas y niveles, para poder transitar de un menú, a una cinematográfica o a una parte jugable.
4. Creación de un gestor de audio que permita a los diseñadores de sonido añadir nuevo contenido y trabajar cómodamente.
5. Creación de un sistema de guardado.
6. Creación de un controlador de cámara capaz de adaptarse fácilmente a diversos usos o situaciones.
7. Unir todos los elementos anteriores con un software modular e independiente para su reutilización en posteriores proyectos.

1.4. Estructura de la memoria

La estructura de esta memoria se divide en cinco capítulos, aparte del presente. En el Capítulo 2, se presenta el estado del arte, junto con las fuentes de inspiración y los avances contemporáneos en el ámbito del desarrollo de videojuegos. El Capítulo 3 aborda el uso de GitHub y la metodología de desarrollo empleada para coordinar nuestro trabajo.

Los Capítulos 4 y 5 constituyen el núcleo central del proyecto, donde se detalla el desarrollo e implementación de las estructuras fundamentales del videojuego, así como la creación del *framework* de diseño y navegación de menús. Cada uno de estos capítulos incluirá una introducción para contextualizar su contenido, diagramas de clases tanto parciales como completos para facilitar la comprensión de la implementación, y una conclusión que sintetizará los aspectos clave y recapitulará lo aprendido.

Luego, en el Capítulo 6, se presenta un caso de uso que abarca todo lo implementado: el juego en el que he estado trabajando junto a mi compañero. Aquí se analizan las utilidades, los resultados obtenidos y cómo los sistemas influyen en la experiencia de juego. Finalmente, en el Capítulo 7, se presentan las conclusiones finales y futuros trabajos.

Es importante señalar que las utilidades de Unity, así como las clases y *scripts*, se mantendrán en su idioma original (*cursiva*) para garantizar una mayor claridad. La mayoría de figuras y tablas presentadas en esta memoria han sido realizadas por nosotros, usando la herramienta “Draw.io” [9]; en caso de no ser así, se indicará debidamente. Algunas de las figuras que representan capturas del juego o de Unity han sido adaptadas para mejorar su visualización en este informe, sin que esto afecte a su comprensión.

Estado del arte

En este capítulo se van a comentar algunos ejemplos que nos han servido de inspiración y que como jugadores creemos que son el nivel más alto de desarrollo al que se ha llegado en sistemas y estructuras fundamentales. Es complicado hablar de “lo más avanzado” en este ámbito puesto que la mayoría de grandes empresas de desarrollo tienen oculta al público su implementación, haciendo casi imposible conocer más detalles que los que te proporciona jugar a su juego. Aun así este primer vistazo puede dar una visión global de lo que aportan estos sistemas y como sin ellos el videojuego no podría funcionar.

Previamente es relevante comentar que aunque los siguientes ejemplos pueda parecer que no están tan vinculados con el diseño de software y la implementación, estos no serían posibles sin una estructura modular y muy bien organizada. Por ejemplo un menú de accesibilidad con muchas opciones debe tener acceso prácticamente a todos los ámbitos del juego, cambiando desde el audio, hasta la forma en la que jugamos o aspectos visuales y a su vez debe tener una alta **cohesión** y, por tanto, un bajo **acoplamiento** de software.

2.1. Audio adaptativo

Las estructuras fundamentales de un videojuego han ido evolucionando a lo largo del tiempo adecuándose a las necesidades que surgían y al crecimiento del medio. Lo que no ha cambiado es que de forma general son un producto audiovisual, el sonido y el aspecto visual cobran la misma importancia. No hay más que quitar el *feedback* auditivo a cualquier juego para comprobar lo vacío que se siente.

Ya en 1972 Allan Alcorn usó un generador de sincronización (un circuito presente en la televisión que se usaba para sincronizar la imagen y evitar que se produzcan barridos horizontales) para producir los efectos sonoros en Pong [10], resultando en el primer videojuego con sonido de la historia. A día de hoy las productoras de juegos AAA destinan mucho presupuesto a la creación del sonido, usando equipos similares a los del cine para efectos sonoros y contratando orquestas para la creación de las bandas sonoras.

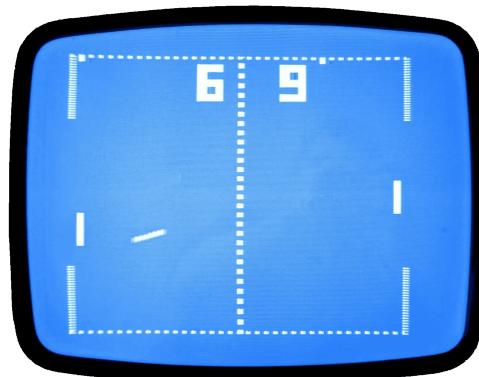


Figura 2.1 – Recreación del videojuego Pong, 1972 - Adaptada de [11]

En una producción cinematográfica la música y el sonido se realizan para un producto que no es dinámico ni va a cambiar en el tiempo que los espectadores lo están viendo, sin embargo, un videojuego se transforma dependiendo de las decisiones del jugador, es por eso que ya desde el año 1978 [12] se pretende que el sonido en los videojuegos sea audio adaptativo.

Nintendo ha sido una de nuestras mayores fuentes de inspiración a este respecto. En sus juegos todos los sonidos están pensados para una completa inmersión y se caracterizan por ser muy dinámicos. Algunos ejemplos son cómo la música de Super Mario Odyssey [13] cambia cuando el personaje se sumerge o se vuelve música que imita los 8 bits cuando llegamos a una zona en 2D. Muchos de estos aspectos nos han hecho pensar en lo importante que es tener un buen sistema de audio con herramientas que permitan a los diseñadores de sonido trabajar cómodamente en el juego y añadir lo que crean más conveniente para la inmersión del jugador.



Figura 2.2 – Super Mario Odissey, 2017 - Adaptada de [14] y [15]

FMOD es probablemente uno de los sistemas de audio adaptativo que más se usan en la industria, teniendo soporte para Unity y Unreal Engine, es usado por compañías como Activision, Capcom, Bethesda, 2K y un largo etcétera [16]. De forma muy resumida FMOD se compone de dos partes, FMOD Studio, una aplicación que te permite crear contenido de audio adaptable para juegos y FMOD Engine, donde mediante unos archivos .Bank podemos cargar el contenido creado en el Studio. FMOD Engine se integra con los motores de Unity y Unreal permitiendo así reproducir audio adaptativo en el juego e independizando el trabajo de los diseñadores (Studio) de los programadores (Engine).

2.2. Sistema de Inputs

Los juegos a su vez son un producto interactivo, es decir, el usuario tiene que tener algún medio para transmitir información al juego. Este aspecto es fundamental y de su gestión se encarga el sistema de *inputs*. En este sistema recae la responsabilidad de tener compatibilidad con la amplia variedad de controladores que hay en el mercado; de ser muy rápido y fiable, puesto que si falla algo el jugador no puede jugar y de tener la mayor adaptabilidad posible para adecuarse a todo tipo de jugadores. A este respecto los desarrollos en plataformas concretas hace que la limitación de hardware restrinja a los usuarios a controladores de la marca o a otros que la marca haya dado compatibilidad, por suerte, cada vez más consolas quieren garantizar la accesibilidad a su público con ideas como Proyect Leonardo para Playstation 5 [17].

Es interesante cuando este mismo dispositivo que utilizamos para transmitir información



Figura 2.3 – Presentación oficial de Project Leonardo para PlayStation 5 - Adaptada de [17]

al juego también lo usamos para obtener una retroalimentación táctil y es que desde que para Nintendo 64 se creó el Rumble Pak [18], la tendencia en la industria de la consola ha sido la creación de mandos con dos motores que permitan obtener vibración al jugador en ambas manos. Pese a que todavía no se ha estandarizado, lo más novedoso a este respecto es el controlador de la PlayStation 5 que usa la tecnología de vibración *háptica*, no es más que una forma avanzada de retroalimentación táctil que permite al jugador sentir una amplia variedad de sensaciones en los dedos y las manos [19]. Por ejemplo en God of War Ragnarok los motores hápticos consiguen generar una vibración que simula la sensación de patinar sobre hielo.



Figura 2.4 – Rumble Pack para Nintendo 64 - Adaptada de [18]

2.3. Cámara

Tras la llegada de los videojuegos en tres dimensiones, la cámara cobra cada vez más importancia, teniendo que adaptarse a cinemáticas, diálogos, combates o al propio movimiento de los personajes. La cámara es la herramienta que permite al jugador ver el mundo que lo rodea y por tanto es fundamental para transmitir un buen *game feel* y que la jugabilidad no se sienta brusca o artificial.

God of war (2018) hace uso del plano secuencia (recurso típico del cine que consta de una única toma sin cortes) para conseguir una inmersión total en la narrativa del juego[20]. Esto no sería posible sin un software controlador de la cámara capaz de adaptarse a cualquier situación, ya que el plano secuencia se mantiene durante toda la duración del juego.

Otro ejemplo de manejo de todas las técnicas típicas de cámara en videojuegos es el caso de la empresa Naughty Dog, que para su videojuego Uncharted 3 (2011) colaboró con GDC Vault en una charla muy interesante [21] donde explican en profundidad la mayoría de aspectos de la cámara a tener en cuenta en un videojuego de acción aventura como el suyo. Pese a que ya

han pasado más de 10 años es una charla muy recomendada no tanto por la implementación que realizaron sino porque las ideas que se transmiten serán fundamentales en cualquier desarrollo de un controlador de cámara.

2.4. Interfaz de usuario

La interfaz de usuario y los menús han evolucionado mucho hasta tal punto que con la complejidad actual de los videojuegos es necesario diseñadores que optimicen bien el espacio de la interfaz tratando que transmitan la información necesaria pero no sobrecarguen al jugador. Todo juego desde hace ya muchos años tiene como mínimo un menú inicial donde poder configurar apartados que permitan la mejor experiencia al usuario. En nuestra opinión The last of us Part II es el máximo exponente a este respecto dando opciones de accesibilidad prácticamente para todo, llegando así al mayor público posible [22]. Creo que este ejemplo lo deberían seguir todos los juegos AAA, invirtiendo más tiempo en este apartado a veces olvidado.

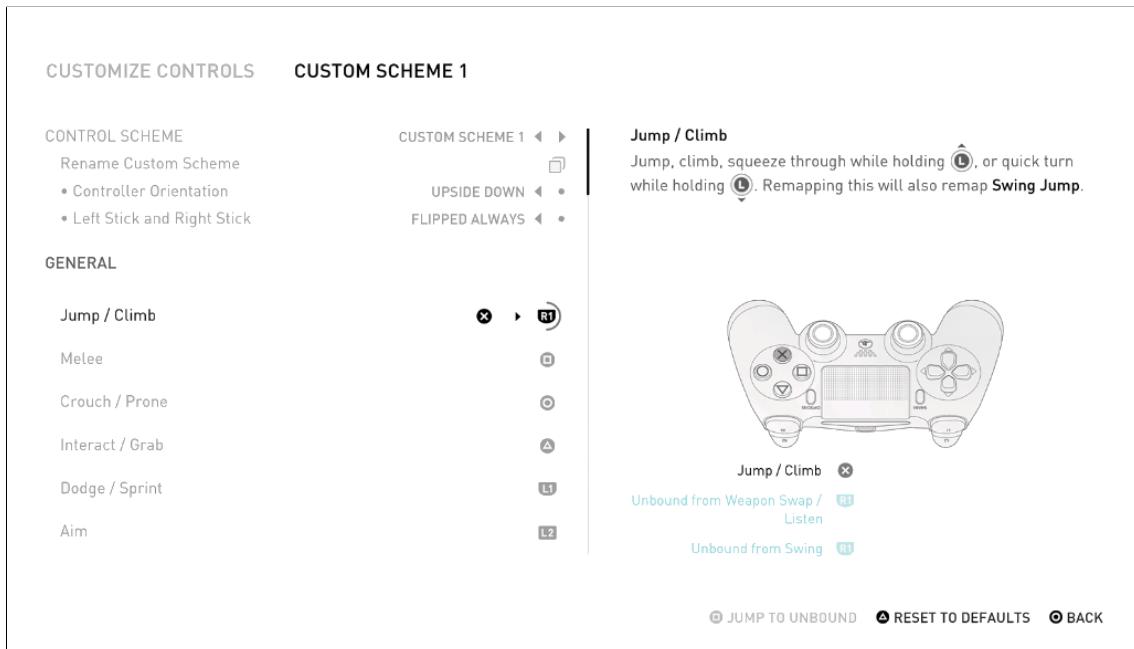


Figura 2.5 – Menú de reasignación de inputs en The Last of Us parte 2 - Adaptada de [22]

2.5. Metodologías de desarrollo de videojuegos

No queremos finalizar esta sección sin antes mencionar algunas de las metodologías que se utilizan actualmente en el desarrollo de videojuegos. Estas metodologías son fundamentales para permitir que grandes equipos de personas trabajen juntos en la creación de los ejemplos que hemos mencionado anteriormente.

- **Documento de diseño del juego.** Este es un componente ampliamente conocido en la industria del desarrollo de videojuegos. Básicamente, implica documentar en detalle todos los aspectos del diseño que se han seguido durante el desarrollo de un videojuego. Este documento es extremadamente útil, ya que proporciona una visión integral de todos los

elementos que rodean al juego. Además, puede servir como recurso de aprendizaje tanto para los miembros del equipo de desarrollo como para otros desarrolladores que deseen aprender de las buenas o malas decisiones tomadas durante el proceso. Es importante destacar que estos documentos suelen ser confidenciales y rara vez se revelan al público en general. Sin embargo, en ocasiones, se pueden encontrar ejemplos de documentos de diseño de juegos más antiguos, como el de Doom [23].

- **Agile y Scrum.** Agile es un enfoque de desarrollo que se centra en la adaptabilidad y la colaboración constante. Scrum es una de las metodologías ágiles más populares. Los equipos de desarrollo de juegos que siguen el enfoque Agile trabajan en iteraciones cortas y se esfuerzan por entregar incrementos de juego funcionales de manera regular [24].
- **DevOps.** Es una metodología que se enfoca en la automatización, la colaboración entre equipos y la entrega continua. En el desarrollo de videojuegos, DevOps puede acelerar el ciclo de desarrollo y garantizar la estabilidad y la calidad del juego [25].

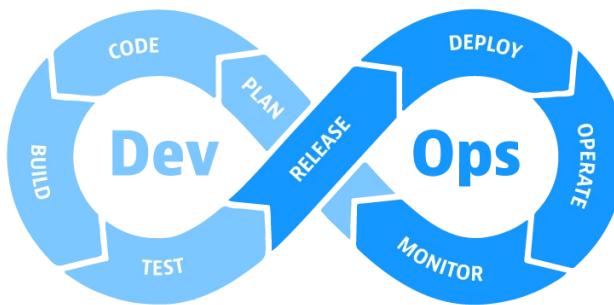


Figura 2.6 – Esquema de funcionamiento DevOps - Adaptada de [26]

- **Diseño Centrado en el Usuario (UCD).** El Diseño Centrado en el Usuario es una metodología que se enfoca en comprender las necesidades y deseos de los jugadores desde las primeras etapas del desarrollo. Esto implica una investigación exhaustiva de usuarios, pruebas de usabilidad y diseño iterativo [27].

2.6. Conclusiones

Desde la música adaptativa de Super Mario Odyssey hasta la cámara en plano secuencia de God of War, nos encontramos frente a características que serían inviables de implementar sin las herramientas adecuadas. Al observar estas características en funcionamiento como usuarios o jugadores, podemos apreciar no solo lo positivo, sino también lo que podría haberse mejorado: ¿Por qué no se ha implementado cierta característica en esta área específica? ¿Por qué la cámara no cambia en esta situación particular? Estas reflexiones nos ayudan a comprender mejor las decisiones de diseño y a apreciar aún más los aspectos destacados de la industria del videojuego.

Este capítulo nos ha permitido adentrarnos en algunas de las mejores aplicaciones de estructuras y sistemas fundamentales que hemos conocido, y nos ha inspirado a considerar cómo podemos aplicar a menor escala lo que hemos observado en sus prácticas. Nuestras implementaciones se basan no solo en nuestros conocimientos de software, sino también en la descomposición y el análisis de todas estas ideas.

Planificación

En este capítulo vamos a abordar la organización, metodología y planificación que hemos empleado en el desarrollo de este proyecto, específicamente en la creación del videojuego Carnem Levare. Abordaremos diversos aspectos que incluyen nuestra forma de comunicación, la gestión de horarios, nuestra metodología de trabajo, la coherencia y compatibilidad del código, entre otros.

3.1. Introducción

Desde el primer momento en que comenzamos con el desarrollo de Carnem Levare, sabíamos que queríamos convertir este proyecto en un trabajo serio. Reconocíamos la necesidad de organizarnos de manera efectiva y adoptar una metodología de trabajo sólida para avanzar al máximo y simultáneamente aprender y corregir errores mientras desarrollábamos un videojuego con el potencial de generar beneficios en el futuro.

Asimismo, no podíamos dedicar tiempo completo al proyecto debido a nuestras otras responsabilidades, como otras asignaturas y tareas. Por lo tanto, consideramos que la forma más adecuada de trabajar no era seguir una metodología de desarrollo de videojuegos convencional, sino más bien definir nuestra propia planificación, establecer nuestras propias reglas y determinar nuestra manera de trabajar.

No obstante, esto no implica que vayamos a descartar por completo algunas ideas típicas de las metodologías de desarrollo de videojuegos, ya que son útiles y se utilizan por una razón. Sin embargo, nuestra intención es adaptarlas y agregar nuevas para que el proceso de trabajo sea más cómodo y productivo.

3.2. Metodología

Nuestra metodología de trabajo puede describirse como una fusión entre la metodología Scrum y los Tableros Kanban. Cada semana establecemos objetivos para la siguiente y llevamos a cabo revisiones regulares del progreso del juego. A diferencia de un horario fijo de reuniones, adoptamos un enfoque más flexible y mantenemos una comunicación constante para abordar cualquier consulta que pueda surgir.

Para implementar esta metodología, hemos empleado principalmente dos herramientas de software: Discord [28] y GitHub [29]. A lo largo de esta sección, exploraremos sus funciones y cómo respaldan diferentes aspectos de la planificación y gestión del proyecto.

3.2.1. Gestión del Tiempo

Desde el inicio del proyecto, nos propusimos adherirnos a un horario establecido de seis horas diarias, cuatro días a la semana. La ventaja de trabajar en un equipo de dos personas hasta el momento nos ha permitido mantener cierta flexibilidad para ajustar estas horas según convenga

en una semana determinada. Aunque este horario se ha respetado en la medida de lo posible para maximizar nuestro compromiso con el proyecto, hemos tenido que realizar ajustes debido a períodos de exámenes u otras circunstancias imprevistas.

3.2.2. Comunicación

En el ámbito del trabajo en equipo, la comunicación desempeña un papel fundamental. En nuestro caso, hemos optado por utilizar un servidor de Discord dedicado al proyecto, con un canal de voz que nos permite estar disponibles mientras trabajamos juntos. A diferencia de enfoques más estructurados como Scrum, donde las reuniones diarias son esenciales, Discord nos brinda la ventaja de mantener una comunicación constante y en tiempo real sin restricciones de horario. Esto nos permitió debatir ideas, compartir avances y resolver problemas de manera inmediata, agilizando la toma de decisiones y previniendo posibles obstáculos en el proceso de desarrollo.

Además, en el servidor hemos creado varios canales de texto específicos para diversos aspectos del desarrollo, como:

- **Notas** - Un espacio para anotaciones e información que deseamos tener a mano y evitar olvidos.
- **Costes** - Para rastrear los gastos económicos relacionados con el proyecto, ya sean adquisiciones de arte para el videojuego o software relevante.
- **Inspiración** - Aquí compartimos contenido audiovisual que nos ha inspirado en el proyecto, permitiendo a futuros colaboradores conocer nuestras fuentes de inspiración.
- **Concept Art** - Almacenamos posibles diseños de personajes que hemos creado, ya sean propios o de terceros.
- **Música** - Este canal alberga las piezas musicales que he compuesto personalmente, proporcionando una dirección para el apartado musical del juego.
- **Assets** - Una recopilación de recursos gratuitos que utilizamos en la versión actual del juego (modelos 3D, animaciones, sprites, etc.).
- **Tutoriales** - Aquí compartimos vídeos explicativos que consideramos útiles para el desarrollo.
- **Enlaces** - Proporcionamos enlaces a sitios web que resultan relevantes para el desarrollo del proyecto.

3.2.3. Selección de objetivos

La elección de objetivos es una piedra angular para evitar cualquier estancamiento en el proceso de desarrollo y asegurarnos de que siempre tengamos tareas que abordar. Adoptar un enfoque Kanban para gestionar estos objetivos nos proporciona una manera eficaz de centralizar la información y obtener una visión rápida del progreso del proyecto [30].

En nuestro caso, hemos optado por utilizar la funcionalidad de gestión de proyectos de Github como nuestro "tablero Kanban", ya que cumple una función muy similar. A través de esta herramienta, podemos asignar objetivos a los diferentes colaboradores, etiquetarlos y brindar descripciones breves que los caractericen. Además, podemos categorizar los objetivos en términos de si se trata de errores, mejoras, características adicionales, entre otros. Asimismo, esta

funcionalidad nos permite priorizar y ordenar los objetivos según su importancia o complejidad, permitiendo una gestión más efectiva ([Figura 3.1](#)).

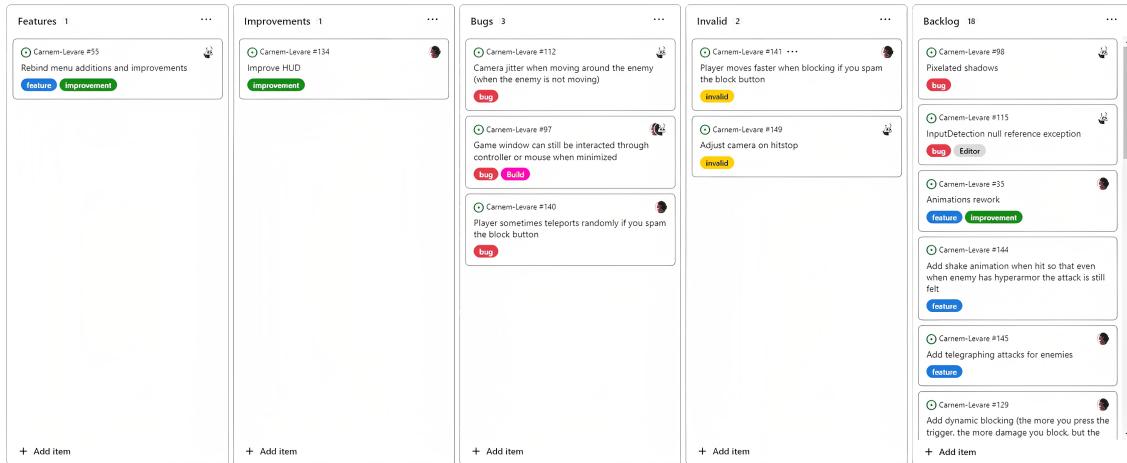


Figura 3.1 – Tareas actuales del proyecto de Github

La gestión de tareas en Github se realiza a través de una funcionalidad denominada *draft issues* [31]. Esta característica nos permite añadir rápidamente una breve descripción de la tarea al tablero de tareas. Aunque esta función está disponible únicamente en la gestión de proyectos, su utilidad puede ser limitada, a menos que se aplique a casos específicos.

En lugar de mantener las tareas como *draft issues*, optamos por convertirlas en *issues* convencionales de Github [32]. Esta conversión nos brinda la capacidad de añadir etiquetas a las tareas y, lo que es más importante, nos permite marcarlas como completadas, asignarles un estado de resolución y proporcionar una traza detallada de todas las tareas que hemos llevado a cabo a lo largo del proyecto. Esta práctica facilita la supervisión de nuestro progreso y proporciona un historial completo de las actividades realizadas.

La selección de tareas y objetivos es un proceso constante y dinámico. Dado que mantenemos una comunicación continua entre nosotros, discutimos las implementaciones necesarias, compartimos obstáculos y celebramos logros. Esto se traduce en una actualización regular de nuestro tablero de tareas en función de las necesidades y el progreso del proyecto.

3.2.4. Metodología ágil y Scrum

Nuestro enfoque de trabajo se asemeja en ciertos aspectos a la metodología Scrum, pero no es una adhesión total a esta metodología. Reconocemos que en el desarrollo de software, especialmente en la creación de videojuegos, una metodología en cascada resulta poco práctica. La rigidez con la que ordena las etapas del proceso no es viable en un proyecto caracterizado por cambios constantes y un componente artístico, donde es común que el programador deba esperar al diseñador o artista y viceversa. Además, pueden surgir fallos o decisiones de diseño que requieran rehacer partes del código [33]. En este contexto, adoptar una metodología ágil se vuelve esencial para abordar con eficacia los desafíos inherentes al desarrollo de videojuegos [34].

Nuestra implementación de Scrum se diferencia en que no asignamos roles específicos, ya que consideramos que, dado que somos solo dos personas, ambos actuamos como el *Scrum Master*, los *Stakeholders* y el equipo de desarrollo. Adaptamos la aplicación de la metodología según sea

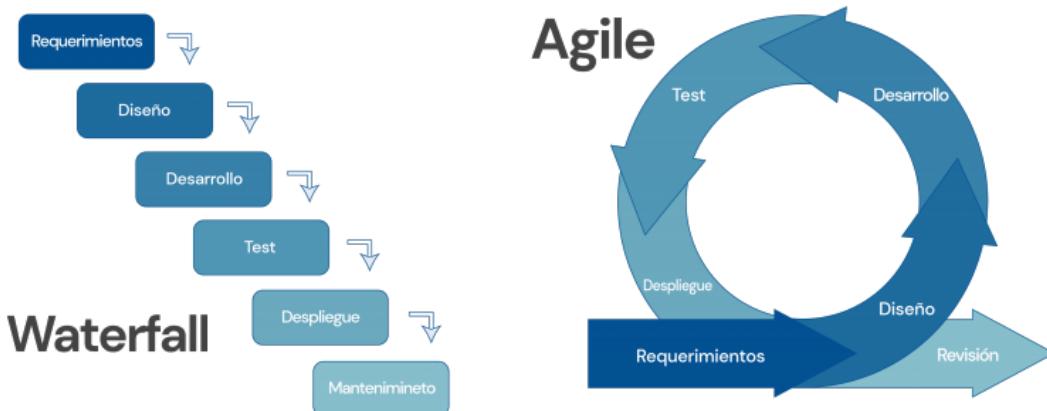


Figura 3.2 – Esquema de metodología ágil y en cascada. Adaptada de [35]

necesario, retrasando o adelantando ciertas implementaciones en función de imprevistos o de la resolución más rápida de problemas.

En términos generales, empleamos *sprints*, estableciendo objetivos para las próximas una a cuatro semanas. Una vez completados, realizamos una revisión exhaustiva de nuestro progreso y construimos una versión del proyecto para verificar si existen errores en la compilación.

3.2.5. Control de versiones

El control de versiones en un proyecto de software es fundamental para manejar los numerosos cambios realizados en los elementos y configuraciones del proyecto. En nuestro caso, hemos optado por utilizar Git para esta tarea. A pesar de que nuestro proyecto de videojuego involucra una cantidad considerable de archivos de software, Git sigue siendo una elección eficiente, fiable y compatible. Para alojar nuestro proyecto, como mencionamos previamente, hemos decidido emplear GitHub. Aunque GitHub ofrece opciones para integrar su funcionalidad directamente en Unity, no nos ha convencido esta práctica. En su lugar, hemos optado por utilizar la aplicación de escritorio GitHub Desktop [36]. Esta herramienta nos permite gestionar todos los aspectos comunes del control de versiones, como commits, ramas, issues, revertir cambios y regresar a versiones anteriores, todo ello a través de una interfaz intuitiva y amigable.

Es recomendable utilizar el archivo `.gitignore` específico para Unity [37] con el fin de evitar que archivos locales innecesarios se guarden en GitHub. Esto contribuirá significativamente a reducir el tamaño del proyecto y a simplificar su gestión en el sistema de control de versiones.

3.3. Seguimiento del desarrollo

En esta sección realizaremos un seguimiento completo de todo el proyecto, desde su inicio hasta su conclusión. El propósito es reunir en una única sección todas las etapas de desarrollo, analizar cómo hemos llevado a cabo nuestro trabajo y detallar en qué fases se divide el proceso.

3.3.1. Etapas del proyecto

Con el término “etapas”, nos referimos a los períodos de tiempo que hemos dedicado a funciones específicas del juego. En cada etapa, puede haber varios *sprints*, generalmente uno por semana, con objetivos muy definidos sobre lo que se está desarrollando en ese momento, como solucionar un error, crear una clase o implementar una funcionalidad, entre otros.

La Figura 3.3 muestra una línea de tiempo con todas las etapas lógicas de desarrollo. Estas se han dividido de manera comprensible, considerando que en un videojuego siempre se están corrigiendo errores o mejorando funciones de otras etapas. También habrá etapas solapadas ya que es posible cambiar entre diferentes etapas si nos encontramos bloqueados en alguna.



Figura 3.3 – Etapas de desarrollo del proyecto. Creado con [TimeGraphics](#).

3.3.1.1. Visión global

Para explicar adecuadamente todas las etapas del proyecto, primero mostraremos una serie de gráficos que hemos creado para facilitar la comprensión. En la [Figura 3.5](#) se presentan las ramas que hemos creado en el proyecto, analizando las fechas de inicio de cada rama y las fechas de los últimos *commits* en Github, lo que nos permite construir una línea de tiempo. Además, en la [Figura 3.4](#) se muestra una gráfica que representa el número de *commits* realizados por día en los que hubo actualizaciones en Github. Esta gráfica y las de la [Figura 3.6](#) se han generado utilizando un *script* de Python que recopiló la información relevante del repositorio de Github y la representó en un gráfico de barras.



Figura 3.4 – Commits realizados por cada día de desarrollo

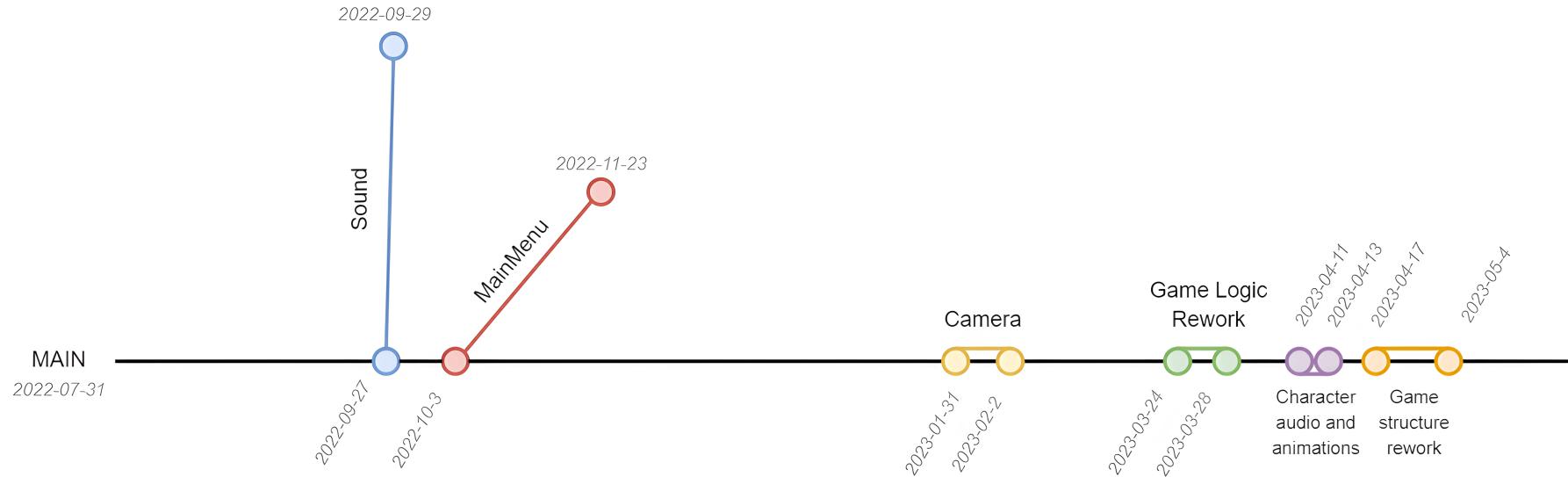


Figura 3.5 – Línea de tiempo de ramas y *Pull Requests*

En líneas generales, no hemos encontrado necesario crear un gran número de ramas en nuestro repositorio, ya que nuestro código se ha complementado de manera fluida en la mayoría de los casos. Cada uno de nosotros trabajó en su área específica sin mayores conflictos. Únicamente utilizamos ramas en situaciones donde vimos una necesidad estricta, como para retroceder a una versión anterior del código de manera segura o cuando queríamos implementar una función sobre la cual no estábamos seguros de su correcto funcionamiento.

Las dos primeras ramas en la línea de tiempo representan casos en los que optamos por no fusionar (*merge*) automáticamente, sino que realizamos la fusión manualmente, transfiriendo el código creado a nuestro proyecto. Esta elección se tomó debido a nuestra inexperiencia previa en trabajar con ramas en proyectos de Unity. No estábamos seguros de cómo se fusionarían todos los *scripts* implicados. En contraste, para las demás ramas del proyecto, procedimos con la fusión y la resolución de conflictos de manera regular.

En relación a la [Figura 3.4](#), es relevante señalar que existen días en los cuales se evidencian cambios realizados por mi compañero y no por mí. Esta disparidad se debe en parte a que en las dos primeras ramas, cuya fusión no quedó registrada, los commits no aparecen en la rama principal. Además, en un par de ocasiones, trabajé en mi propio proyecto de Unity con el fin de probar algunas utilidades sin interferir en el proyecto de mi compañero.

Por último, antes de adentrarnos en la explicación de cada una de las etapas del desarrollo, es necesario aclarar que mientras algunas de ellas representan el trabajo conjunto del equipo, otras reflejan únicamente mi contribución al proyecto.

3.3.1.2. Etapas

Basándonos en la [Figura 3.3](#), vamos a detallar las etapas que consideramos más relevantes o de mayor importancia en el proyecto.

1. **Etapa Inicial.** Esta fase involucró discusiones y exploración para definir la mejor manera de materializar nuestra idea. El proyecto evolucionó y cambió desde este punto inicial. La meta en esta etapa fue crear una demo básica del juego para evaluar su viabilidad ante el público.
2. **Sistema de audio / Interfaz de usuario.** Durante esta etapa, que abarca las dos primeras ramas del repositorio, desarrollamos un borrador inicial del código para el sistema de audio y la interfaz de usuario. Aunque funcional, este código no fue diseñado para ser altamente reutilizable.
3. **Sonido Adaptativo.** Esta etapa marcó el inicio de las dos fases más experimentales del proyecto. Creé un proyecto separado en Unity para experimentar con sistemas de audio, lo cual resultó en la definición del sistema de audio actual en el proyecto.
4. **Procedural Enemies.** Otra fase experimental en la que trabajé en el desarrollo de enemigos procedurales basados en algoritmos genéticos. A pesar de su atractivo, decidimos apartar esta idea para centrarnos en otros aspectos más fundamentales del juego.
5. **Cámara.** En esta etapa, aprendimos a utilizar Cinemachine y definimos casi por completo las funcionalidades de la cámara para el juego.
6. **Gestión de escenas / Pantalla de carga:** A medida que varios aspectos del juego estaban tomando forma, fue necesario unificar las escenas que hasta entonces se desarrollaban por

separado en un sistema de gestión de escenas funcional. La pantalla de carga también se volvió necesaria.

7. **Rework**. Una de las fases más críticas del desarrollo. Tras un periodo prolongado de trabajo, nos dimos cuenta de que debíamos reconsiderar ciertos principios y patrones de diseño de software. Esto llevó a la revisión y mejora de partes importantes del proyecto. Todas las implementaciones que se describen en este proyecto son el resultado directo de esta etapa.
8. **Framework**. Simultáneamente con la etapa de Rework, reconocimos que la forma estándar de navegación de menús en Unity no era la más adecuada. Surgió la idea de desarrollar un Framework interno en Unity para gestionar la navegación de menús.
9. **Game Logic / Moves Menu**. Esta etapa se enfoca en funciones específicas del juego, como la personalización de movimientos de personajes. Esto implica la creación de un menú que permita a los usuarios ajustar los movimientos o mostrar un menú emergente con los movimientos obtenidos tras derrotar a un enemigo.

Es importante destacar la etapa de **Mejora / Corrección de Errores**. Esta etapa abarca casi todo el desarrollo y refleja cómo el proceso de crear un videojuego implica un ciclo continuo de retroceso. Tras pruebas y evaluaciones internas o externas, surgen mejoras y correcciones de errores que pueden no haber sido evidentes inicialmente. Por lo tanto, esta fase es una constante durante el desarrollo, en la que se detiene momentáneamente lo que se está haciendo para resolver problemas y mejorar aspectos detectados.

3.3.1.3. Gestión de tareas (*Issues*)

En la [Figura 3.6](#) se presenta un análisis detallado de la gestión de tareas en todo el repositorio de Github. Desde esta visualización, podemos extraer dos puntos de análisis cruciales.

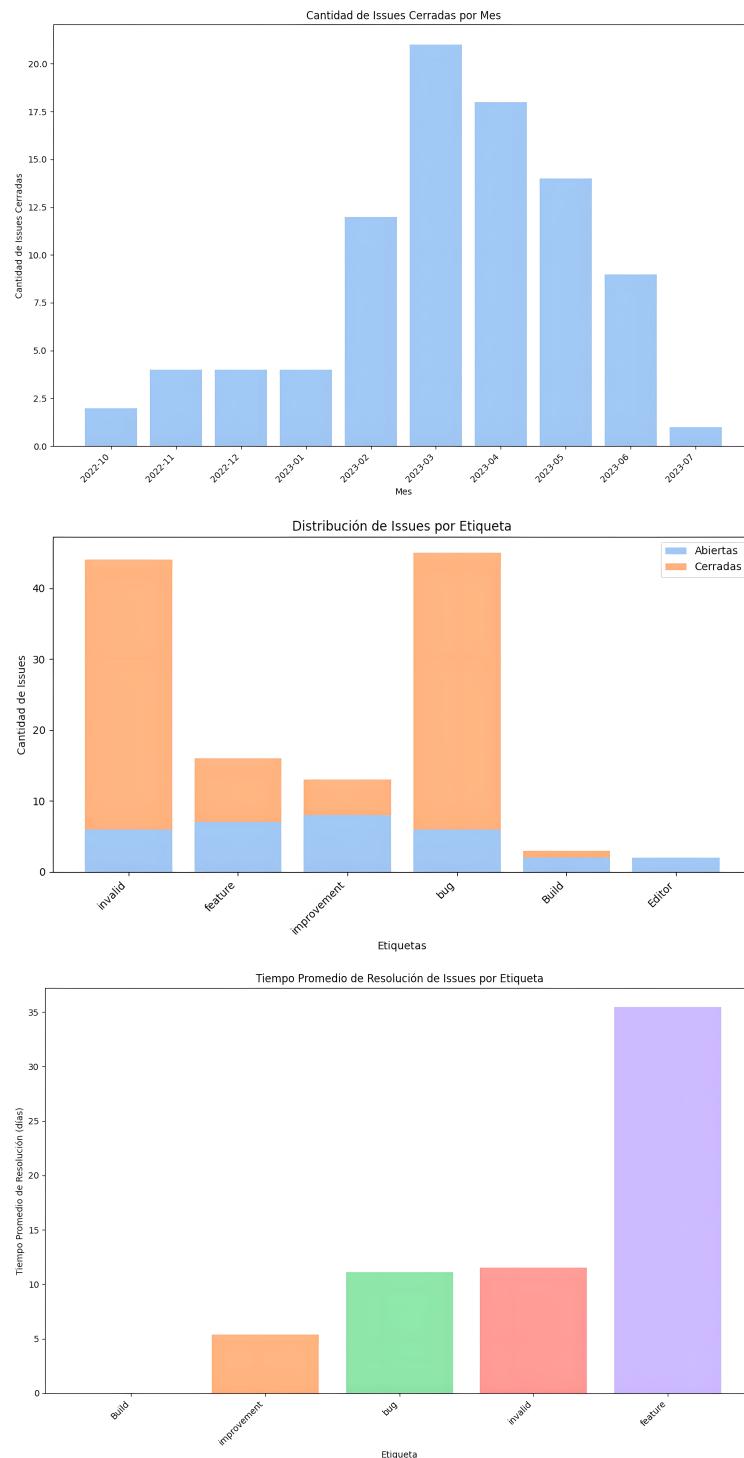
En primer lugar, se observa claramente que a partir de febrero de 2023, fuimos capaces de mantener un ritmo de resolución de tareas más constante y efectivo. Este cambio en el ritmo es indicativo de la mejora en la eficiencia y el enfoque del desarrollo durante esa fase.

Por otro lado, también se evidencia que las tareas de corrección de errores (etiquetadas como *bug*) y las tareas marcadas como *invalid* (indicando que algo no parece correcto o puede mejorarse) han sido las más frecuentes a lo largo del proyecto. Esto refleja la importancia de la detección temprana y la solución de problemas para mantener la calidad del proyecto. En contraste, las tareas relacionadas con la implementación de nuevas características (*feature*) son menos comunes. Esto es en parte porque estas funcionalidades suelen ser más complejas y requieren un mayor tiempo de desarrollo en comparación con las correcciones puntuales de errores.

3.4. Análisis

Para concluir este capítulo de planificación, deseamos compartir nuestras reflexiones después de casi un año de trabajo en este proyecto y explicar los motivos que respaldan las decisiones presentadas en este capítulo.

En primer lugar, es fundamental destacar que todo el proceso de desarrollo ha sido un continuo proceso de aprendizaje. Si bien habíamos trabajado en el desarrollo de videojuegos ante-

**Figura 3.6 – Issues cerradas durante el proyecto**

riormente, nunca habíamos abordado un proyecto con esta profundidad y nivel de detalle. Este factor también ha influido en nuestra curva de aprendizaje, ya que hemos enfrentado desafíos inesperados y hemos tenido que revisar y rehacer varias partes del proyecto.

Observando la distribución temporal de las distintas etapas, podría parecer que la mayor parte del trabajo se realizó en los últimos meses. Sin embargo, esta impresión no refleja completamente la realidad. Si bien al comienzo del proyecto teníamos otras responsabilidades que requerían atención semanal, la mayor parte del aprendizaje y la consolidación de conceptos clave tuvieron lugar en la primera fase del proyecto. Esta etapa definitivamente definió y moldeó el juego en su forma actual.

Estructuras fundamentales de un videojuego

4.1. Introducción

En este capítulo, abordaremos el desarrollo e implementación de las estructuras o sistemas que consideramos esenciales para cualquier software de videojuegos, con el objetivo de adaptarse a la industria actual. En primer lugar, describiremos el esqueleto o la estructura que englobará dichos sistemas, con el fin de comprender todo lo que necesitamos desarrollar y la forma en que se modularizan. Luego, abordaremos cada sistema en detalle, explicando paso a paso todo lo que se ha implementado.

Se ha optado por separar el contenido de este capítulo del videojuego en desarrollo ([Capítulo 6](#)) debido a su extensión y con el propósito de brindar una mayor flexibilidad a los lectores. No obstante, esta separación no implica una desvinculación total, y es perfectamente válido y enriquecedor consultar ambos capítulos simultáneamente y así tener una visión tanto práctica como teórica de cada uno de los sistemas.

4.2. Detalles de la implementación

Antes de pasar a la parte más técnica y centrada exclusivamente en la implementación, quiero comentar algunas cuestiones a tener en cuenta durante todo el capítulo.

Es importante recordar que esta implementación se realiza en Unity en todo momento. Por lo tanto, el funcionamiento a bajo nivel de estos sistemas ya está proporcionado por la herramienta. Nosotros nos centraremos en la capa intermedia existente entre el motor gráfico y el diseñador. Nuestro objetivo es facilitar al máximo la escalabilidad y la variedad de opciones, para que el diseñador pueda trabajar de manera cómoda. Dentro del contexto de esta implementación, utilizamos el término “scripts” para referirnos a las clases implementadas. Esta nomenclatura se debe a que en Unity es necesario crear un archivo separado por cada clase que deseamos manejar. De hecho, el nombre del archivo debe coincidir con el nombre de la clase para que el código compile correctamente [38].

En cuanto a los diagramas de clases que presentamos, es importante mencionar que no contienen la totalidad de métodos y variables que tienen en realidad; hemos seleccionado solo aquellos que consideramos esenciales para una comprensión clara de la explicación. Asimismo, nos enfocamos en explicar las herramientas de Unity que consideramos fundamentales para comprender la idea detrás de la implementación. No obstante, si se desea obtener más información, se puede consultar el glosario de términos clave que proporcionaremos o acceder directamente a la documentación oficial de Unity [39].

4.3. Arquitectura software

La clave de esta sección es establecer los cimientos que nos permitan interconectar los distintos módulos o sistemas, fomentando un bajo [acoplamiento](#) e intentando reducir al mínimo el

número de dependencias. Para lograr esto, introduciremos en las siguientes secciones dos herramientas que facilitan la comunicación entre los sistemas y el programador o diseñador, así como con el resto del código de nuestro juego.

En la [Figura 4.1](#), se muestra un esquema del resultado final. La representación de los sistemas es equivalente a los *namespace* de C#, es decir, estas cajas contienen todos los *scripts* cuya funcionalidad está relacionada con un sistema específico. Si algún elemento accede a estas cajas, significa que necesita acceder a alguno de sus *scripts*. Del mismo modo, si alguna caja hereda de otro elemento, significa que uno o varios de sus *scripts* necesitan heredar de él.

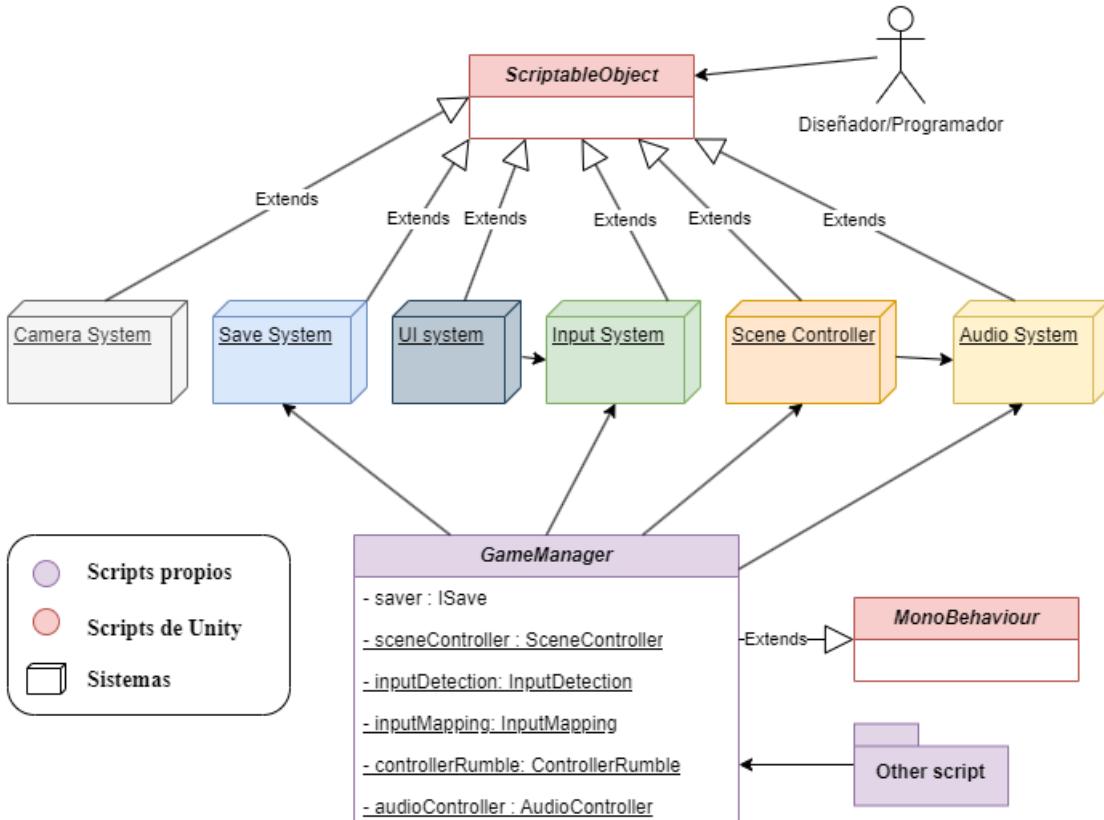


Figura 4.1 – Estructura software completa

4.3.1. Patrón de diseño Fachada

El cometido del patrón Facade o Fachada es hacer de intermediario entre los consumidores y los distintos sistemas [40]. En nuestro caso la “fachada” es el *script* *GameManager* ([Figura 4.1](#)) que contiene una instancia estática de los *scripts* de cada sistema que necesiten un acceso global. Decidimos hacer estáticas las instancias a los sistemas por que a lo largo de toda la ejecución del juego solo hay un componente *GameManager* y debido a que estos sistemas son independientes al funcionamiento específico, solo necesitamos una instancia de cada uno de ellos. Como podemos comprobar ahora cualquier consumidor que quiera acceder a alguna funcionalidad de estos sistemas solo tiene que hacerlo a través de alguna función pública definida en *GameManager*, de hecho, como las instancias son estáticas, estas funciones lo pueden ser también, facilitando aún más su acceso.

Para dejar más claro las ventajas que proporciona este patrón de diseño quiero poner un

ejemplo ([Figura 4.2](#)) que ilustra como podemos reducir las dependencias y disminuir el **acoplamiento**.

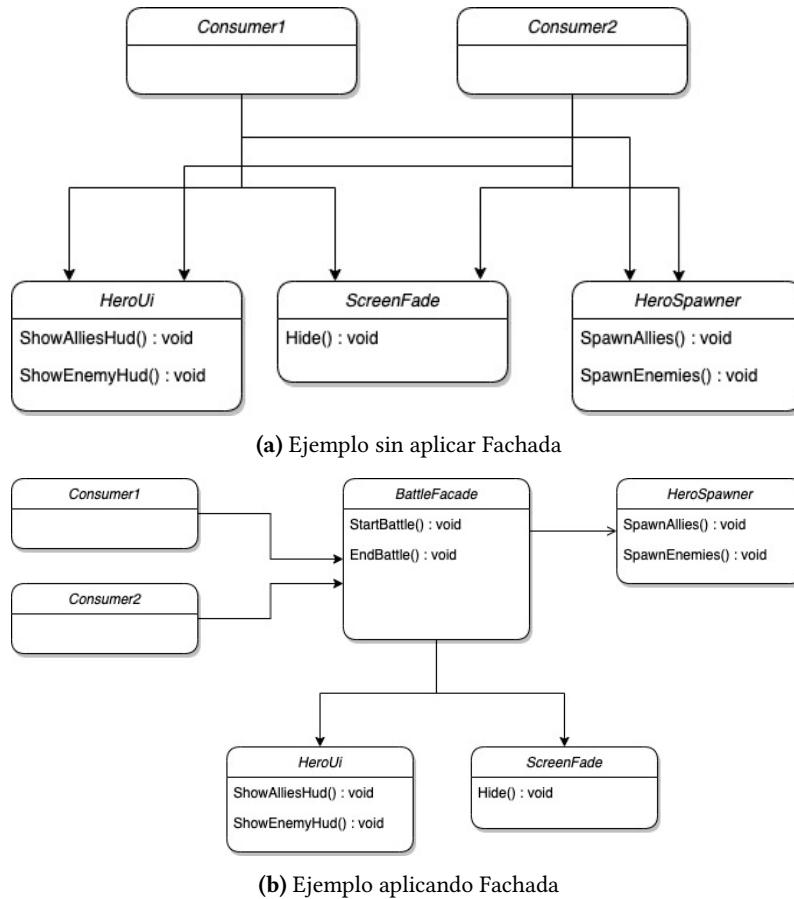


Figura 4.2 – Ejemplo de uso del patrón de diseño Fachada - Obtenido de [41]

Si nos fijamos el único *script* de la estructura que hereda de *MonoBehaviour* es *GameManager*, esto nos va a proporcionar dos grandes ventajas:

1. Código más organizado, dividido en módulos independientes capaces de trabajar por sí mismos y un único *script* que maneja su inicialización y ejecución dentro del ciclo del juego (*game loop*)[\[3\]](#).
2. La eliminación de condiciones de carrera debido a que desde *GameManager* podemos definir exactamente qué orden de inicialización van a tener los distintos sistemas, por ejemplo en la [Figura 4.1](#) vemos como *Scene Controller* va a acceder en algún momento a *Audio System*, por lo que solo debemos asegurarnos de inicializar antes el sistema al que se va a acceder.

Un problema que puede acarrear este patrón de diseño es que la fachada va a proporcionar a los consumidores más funciones de las que realmente necesitan. Por ejemplo si la clase *Player-Movement* necesita acceder solo al *Save System* para guardar el estado actual del jugador, va a tener acceso a todos los demás sistemas que no necesita. Una forma de solventarlo es la definición de interfaces que encapsulen los comportamientos por separado y hacer que *GameManager* las implemente (Segregación de Interfaces (ISP) de SOLID)[\[42\]](#).

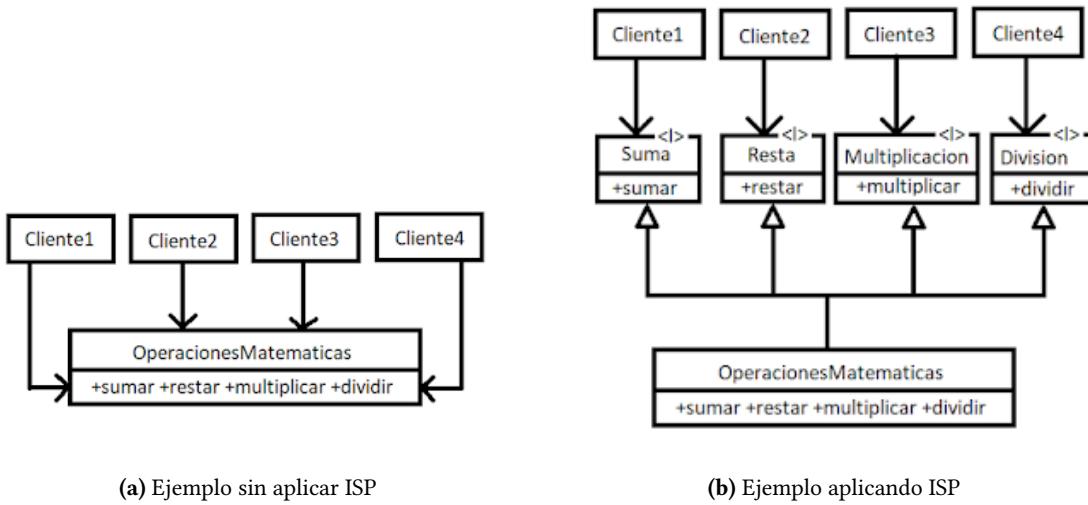


Figura 4.3 – Ejemplo del principio de Segregación de interfaces - Obtenido de [43]

Camera System es también un sistema fundamental, pero no es recomendable una única instancia del mismo ya que la cámara puede variar o cambiar completamente entre *escenas*. La cámara es completamente independiente al resto de sistemas fundamentales y normalmente solo necesita acceder a los objetivos que mire o apunte, generalmente jugador o enemigo. De manera similar, el sistema de interfaz de usuario también presenta esta característica, ya que cada escena puede tener sus propios menús interactivos y solo requiere acceso al sistema de *inputs* para definir su comportamiento.

4.3.2. *ScriptableObject* como comunicador

Durante la implementación de los sistemas descritos en las secciones siguientes, nos encontramos ante la necesidad de encontrar la forma más efectiva de definir y ajustar los parámetros sin tener que sumergirnos en el código. En Unity, la opción más común para serializar parámetros es heredar de la clase *MonoBehaviour*, la cual automáticamente serializa los parámetros públicos y privados (mediante el uso de *SerializeField* [44]) en el *inspector*. Sin embargo, ¿qué sucede si no deseamos que nuestro *script* forme parte del *ciclo del juego* (*game loop*)?

Aquí es donde entra en juego la clase *ScriptableObject*, proporcionando las siguientes ventajas:

- Las instancias de esta clase se definen en el árbol de directorios, lo cual resulta muy conveniente para crearlas y modificar sus parámetros o propiedades.
- Estas instancias existen fuera del *ciclo del juego*, lo que significa que los parámetros se guardan incluso cuando se modifican durante la ejecución.
- Los *ScriptableObjects* pueden ser utilizados como propiedades de un *script MonoBehaviour*, lo que nos permite cambiar un conjunto completo de parámetros simplemente arrastrando un nuevo *ScriptableObject* a dicha propiedad.

Ejemplo: Imaginemos que en nuestro juego los enemigos tienen una serie de parámetros (vida, daño, velocidad, *sprite*, etc.) y el diseñador desea crear diferentes tipos de enemigos basados en estos parámetros. Podemos crear una clase *ScriptableObject* que

almacene todas estas propiedades, permitiendo que el diseñador cree instancias de enemigos fuera de la ejecución del juego. Una vez creadas, solo necesitamos asignarlas a los enemigos en la [escena](#).

- También es una herramienta muy útil como manejador (*Handler*) de eventos, como explicaremos en secciones posteriores.

Podemos apreciar la utilidad de esta característica para el almacenamiento y modificación de datos en los sistemas que estamos desarrollando, especialmente considerando que solo hay un elemento estático de estos sistemas en ejecución.

4.3.3. Breve introducción a eventos de C#

Creo que es conveniente definir el funcionamiento de los eventos de C# ya que es una utilidad que estaremos usando mucho en este capítulo y a lo largo del proyecto. Usando la definición de Microsoft [45] “Un evento es un mensaje que envía un objeto cuando ocurre una acción. La acción podría deberse a la interacción del usuario, como hacer clic en un botón, o podría derivarse de cualquier otra lógica del programa, como el cambio del valor de una propiedad”. Por analogía sería algo así como suscribirse a una revista, es decir, hay mucha gente suscrita a una revista y a todas ellas les llega una copia de la misma sin tener que hacer nada. De la misma forma a un evento podemos suscribirle funciones del programa y este llamará a todas esas funciones una vez sea invocado. La ventaja de los eventos es que el invocador o emisor no tiene porque saber nada acerca de los suscriptores siendo muy útil para independizar dos sistemas diferenciados. Los eventos no son más que la aplicación en C# del patrón Observador [46].

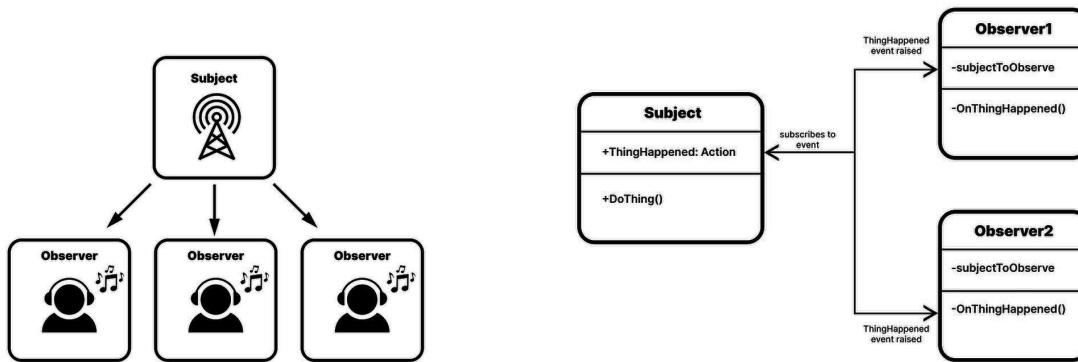


Figura 4.4 – Esquema de funcionamiento del patrón Observador - Adaptado de [47]

4.4. Sistema de Inputs

El sistema de *inputs* consiste en un conjunto de estructuras software que nos permiten reconocer e identificar los distintos periféricos que controlan nuestro juego, además debemos ser capaces de detectar cuando y cómo se ha pulsado alguna de las teclas o botones del periférico. Sumado a esto también se tienen que poder asignar eventos o funciones a las distintas entradas de los periféricos para poder obtener una reacción lógica a la acción que esté llevando a cabo el usuario.

La idea es conseguir que el manejo de los eventos asignados a cada entrada sea lo más escalable e independiente posible al código específico. Otro reto a conseguir es que el jugador pueda cambiar en cualquier momento los *inputs* o *bindings* asignados a cada entrada y la forma en que los vamos a representar en la interfaz.

En la figura [Figura 4.5](#) se muestra el diagrama de clases del sistema de *inputs* para poder seguir la explicación con una ayuda visual.

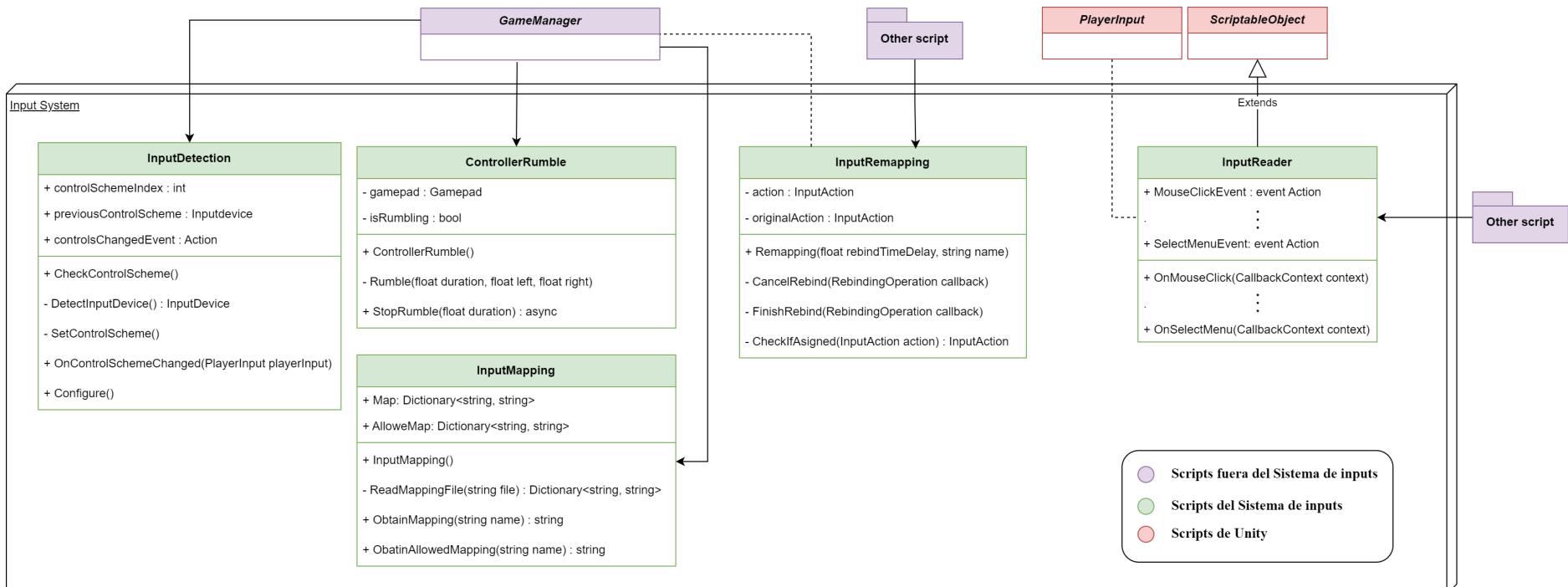


Figura 4.5 – Diagrama de clases del Sistema de Inputs

4.4.1. Breve introducción al *InputSystem* de Unity

Gran parte de este trabajo se realiza utilizando el *plugin InputSystem* desarrollado por Unity [48]. Este plugin simplifica mucho las tareas, ya que, gracias a su estructura basada en acciones (*actions*), proporciona una capa de abstracción entre los periféricos y el código específico del juego. En resumen, tenemos la capacidad de definir conjuntos de acciones (*action maps*) que nos permiten crear acciones y asignarles las entradas (*bindings*) que deseemos. De esta manera, una misma acción, como por ejemplo “PlayerMovement”, puede contener tanto las teclas de flecha del teclado como el joystick izquierdo de un controlador. Estas acciones tienen eventos definidos que se activan cuando se detecta algún cambio en los periféricos. Por ejemplo, cuando movemos el joystick izquierdo de nuestro controlador, se activarán los eventos asignados a la acción “PlayerMovement”, lo que resulta en un código altamente reactivo, algo crucial en un videojuego.

A esta ventaja se suma el hecho de que el *InputSystem* es capaz de gestionar todos los periféricos que deseemos utilizando una única estructura. Esto significa que, independientemente del controlador que se esté utilizando, el juego debería funcionar perfectamente. Al definir diferentes conjuntos de acciones (*action maps*), podemos tener comportamientos distintos para nuestro juego de forma individualizada, lo que permite que al cambiar el conjunto de acciones que se está utilizando en un momento dado, el *Gameplay* pueda cambiar por completo. Podríamos profundizar más en el funcionamiento del *InputSystem* de Unity, pero creo que con esta información es suficiente para adentrarnos en el tema que nos interesa.

4.4.2. Definición de eventos

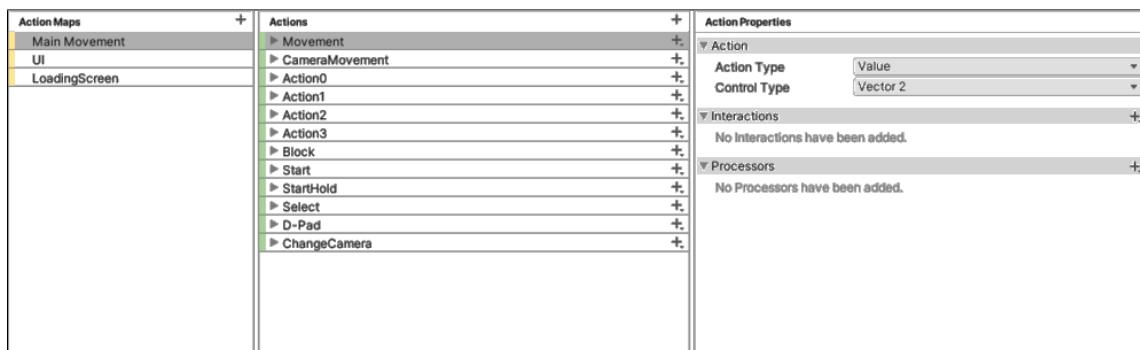


Figura 4.6 – Interfaz de *Input Action Asset*

Lo primero que debemos hacer para definir nuestro *action map* es crear un *Input Action Asset* (Figura Figura 4.6) donde podremos manejar todos las *actions* y *action maps* de nuestro juego. También debemos crear un *Player Input* que es el *script* donde suscribiremos funcionalidades al evento de cada *action* que tengamos definida.

Una vez hecho esto, debemos definir cómo enlazaremos los comportamientos del juego con estos eventos del *Player Input*. La primera idea es suscribir directamente funciones específicas al evento de cada acción. Sin embargo, esto tiene varios problemas, ya que el *Player Input* es un *script* con muchas razones de cambio entre *escenas*. Además, para acceder a este *script* desde cualquier otra parte del proyecto y, al mismo tiempo, mantener el sistema de *inputs* independiente, tendríamos que utilizar algún tipo de búsqueda basada en etiquetas [49], lo cual puede ser costoso para el ciclo del juego. La segunda opción fue utilizar un evento anidado, es decir, nuestro evento de acción está suscrito a una función que a su vez llama a otro evento. Esta im-

plementación nos permite tener una capa adicional de abstracción (*Input Reader*) entre el *Input System* y el código específico. Podemos hacer que el *Input Reader* sea mucho más accesible de forma global en todo el proyecto, utilizando el *Player Input* como emisor del *Input Reader* y el resto del código como suscriptores del mismo (Figura 4.7).

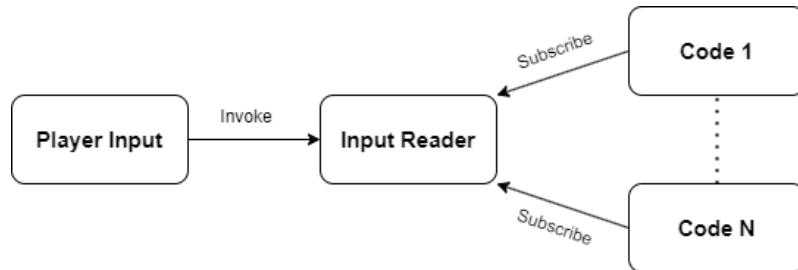


Figura 4.7 – Esquema de funcionamiento *Input Reader*

Para hacer que el *Input Reader* sea accesible en todo el proyecto, inicialmente utilizamos el patrón de diseño Singleton. Sin embargo, a medida que avanzamos en el desarrollo, nos dimos cuenta de que este patrón puede ser menos útil de lo que parece a primera vista, ya que existen alternativas más sencillas y eficaces, como explica Robert Nystrom en su libro Game Programming Patterns [50]. La alternativa que mejor se ajustaba a las necesidades fue utilizar la clase *ScriptableObject*. Al ser una clase abstracta, podemos hacer que el *Input Reader* herede de ella y así crear nuestro propio *ScriptableObject*. Con esta estructura, logramos que todos los eventos sean una única instancia en el proyecto, invariable en el tiempo y accesible públicamente para todos los *scripts* del juego [51].

En la Figura 4.8 y Código 4.1 se muestra cómo podemos agregar las funciones definidas en el *Input Reader* a nuestro *Player Input*, las cuales a su vez llaman a nuevos eventos. Desde cualquier otro *script*, como la gestión del menú de pausa, solo debemos suscribirnos o cancelar la suscripción a dichos eventos según sea conveniente. En este ejemplo, las funciones “MoveRight” y “MoveLeft” se llamarán cuando presionemos los botones del control asignados a las acciones “ChangeRightMenu” y “ChangeLeftMenu”.

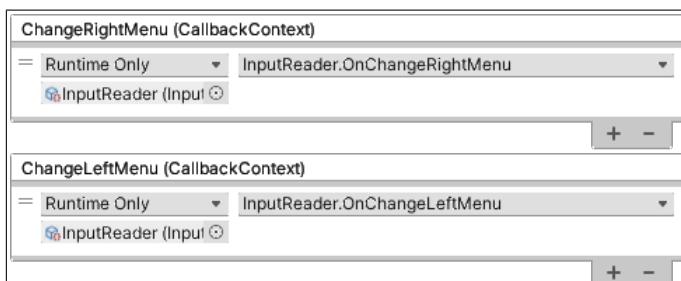


Figura 4.8 – Ejemplo de uso de *Player Input*

Código 4.1 – Ejemplo de código del menú de pausa

```

1  private void Awake()
2  {
3      input.ChangeRightMenuEvent += MoveRight;
4      input.ChangeLeftMenuEvent += MoveLeft;
5      input.MenuBackEvent += Return;
6  }
7
8  private void OnDestroy()
  
```

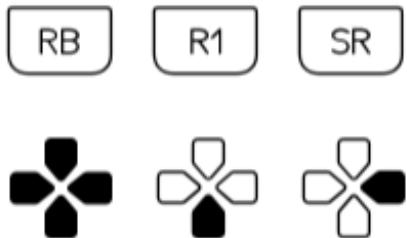
```

9     {
10    input.ChangeRightMenuEvent -= MoveRight;
11    input.ChangeLeftMenuEvent -= MoveLeft;
12    input.MenuBackEvent -= Return;
13 }

```

4.4.3. Representación de inputs

Si se desea que la representación visual de los *inputs* cambie según el controlador utilizado o varíe en función de las opciones elegidas por el usuario, necesitaremos establecer equivalencias entre lo que el jugador ve y la información que tenemos como programadores. Una opción inicial podría ser tener una imagen por cada input distinto (tecla del teclado, botón de un mando, etc.) y asignar la imagen correspondiente a la *action* que queremos representar. Sin embargo, esto resultaría en una gran cantidad de imágenes, ya que las *actions* pueden tener múltiples *bindings* asociados para teclado/ratón u otros tipos de controladores.



(a) Fuente para mando



(b) Fuente para teclado

Figura 4.9 – Fuentes usadas para la representación de *inputs*

Una opción más factible que consideramos fue utilizar una fuente de letra en la que cada entrada estuviera asociada a un símbolo que representara el input deseado (Figura 4.9). Una vez que tenemos la representación visual, necesitamos asegurarnos de que estas equivalencias sean accesibles a nivel de código. Para ello, optamos por crear un **diccionario** que asocie a cada *binding* la letra correspondiente en la representación visual. Elegimos utilizar diccionarios porque, al estar implementados como tablas *hash*, recuperar un valor mediante su clave es muy rápido, cerca de O(1) [52].

Para tener acceso continuo a las equivalencias y poder cambiarlas si es necesario (por ejemplo, si se cambia la fuente u otro motivo), creamos un archivo .txt que almacena las equivalencias con la nomenclatura mostrada en el Código 4.2 y Código 4.3. En resumen, tenemos un archivo de equivalencias .txt y un *script* llamado *Input Mapping* que, al inicio de su ejecución, lee ese archivo y almacena las equivalencias en un **diccionario** para poder consultarlas en la interfaz de usuario.

Código 4.2 – Ejemplo de equivalencias entre fuente y *action* (Mando)

```

1 <Gamepad>/rightShoulder : z
2 <Gamepad>/leftTrigger : k
3 <Gamepad>/rightTrigger : b
4 <Gamepad>/leftStickPress : e

```

Código 4.3 – Ejemplo de equivalencias entre fuente y *action* (Teclado)

```

1 <Keyboard>/leftShift : c
2 <Keyboard>/rightShift : g
3 <Keyboard>/leftAlt : b
4 <Keyboard>/rightAlt : g

```

Es importante tener en cuenta el dispositivo que estamos utilizando, ya que las fuentes de letra deben cambiar según si estamos utilizando un teclado, un ratón o un mando. Para esto, creamos el *script Input Detection*, donde podemos consultar el último dispositivo utilizado y la ID asociada a ese dispositivo. Esta ID simplemente representa el índice en un vector donde tengo almacenadas las fuentes de letra. De esta manera, si estamos utilizando el teclado o el ratón, la fuente a utilizar será la almacenada en el índice 1, y si estamos utilizando un mando, la fuente almacenada en el índice 0.

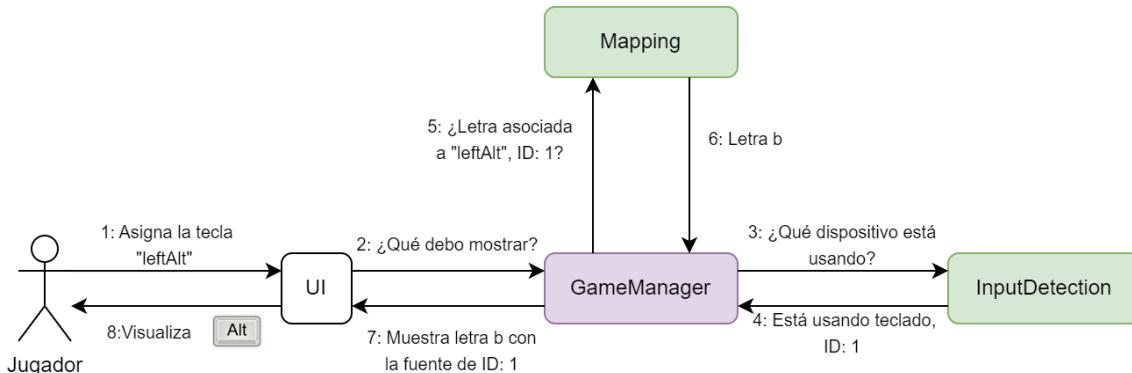


Figura 4.10 – Diagrama de comunicación para representación de inputs

Con el [diccionario](#) y este acceso al índice de la fuente de letra, tenemos una interfaz que cambia de forma dinámica la visualización al usuario, asegurando que siempre tenga la representación adecuada independientemente del dispositivo utilizado. Por ahora, solo se consideran el teclado/ratón y el mando, que son los dispositivos más comunes, pero solo es necesario añadir nuevas equivalencias al .txt y nuevas fuentes al vector para adaptar la interfaz a otros dispositivos.

4.4.4. Reasignación de *inputs*

La reasignación o *remapping* de *inputs* consiste en que el usuario pueda asignar un *binding* distinto a cada *action*, permitiendo así una mayor personalización de los controles. Dado que ya tenemos la representación visual de los *inputs* automatizada, solo necesitamos encontrar una forma de cambiar los *bindings* en tiempo de ejecución. Antes que nada, debemos tener en cuenta que, como se aprecia en la [Figura 4.6](#), se pueden almacenar varios bindings para una misma *action* y se puede acceder a ellos mediante un vector [53]. En nuestro *action map*, hemos decidido que el primer *binding* siempre sea el del mando y el segundo el del teclado/ratón. De esta manera, podemos utilizar el ID definido anteriormente en *Input Detection*.

Para llevar a cabo el cambio de *binding* en ejecución, utilizamos la función *PerformInteractiveRebinding* definida en la clase *Action* [54], a la cual le pasamos como parámetro el índice del *binding* que deseamos cambiar. Esta función pausa el *Input System* momentáneamente y espera

a que el usuario pulse la nueva tecla o botón que desea asignar. Una vez que se pulsa, se asigna ese *binding* a la *action* que llamó a la función. Hemos añadido funcionalidades adicionales, como excluir al ratón de los *inputs* disponibles para reasignar y mostrar un mensaje al jugador con la información necesaria para realizar la acción.

Por último debemos de tener en cuenta un par de escenarios:

- ¿Qué sucede si el jugador presiona una tecla que no es válida? Para abordar este caso, hemos creado otro archivo .txt con equivalencias similares a las mostradas en el [Código 4.2](#) o [Código 4.3](#), pero solo contiene los *bindings* permitidos para ser reasignados. Al igual que antes, también hemos asignado estas equivalencias a un [diccionario](#) en *Input Mapping*. De esta manera, si al ejecutar la función *PerformInteractiveRebinding* intentamos acceder a un *binding* que no se encuentra en dicho [diccionario](#), el proceso se cancela.
- ¿Qué sucede si el jugador asigna el mismo *binding* a dos *actions* distintas? En este caso, hemos creado una función que itera sobre los *bindings* del *action map* actual y determina si el *binding* que vamos a asignar ya se ha utilizado. Si eso ocurre, se elimina el *binding* más antiguo y prevalece el nuevo.

4.4.5. Vibración

Añadir la vibración a nuestro mando es sencillo gracias a que *Input System* proporciona la función *SetMotorSpeeds(float, float)*, la cual activa la vibración en los motores izquierdo y derecho del mando. Nuestra implementación consiste en un *script* con una función que verifica si estamos utilizando un mando (*Input Detection*) y, en caso afirmativo, activa la vibración durante un tiempo determinado. Este tiempo se puede pasar como parámetro, al igual que la intensidad de vibración deseada para los motores izquierdo y derecho.

Para controlar el tiempo, utilizamos funciones asíncronas ([async](#)) que, como se menciona en el artículo “Asynchronous programming” de Microsoft [55], son ideales para operaciones de E/S ([Figura 4.11](#)). Una vez transcurrido el tiempo especificado, restablecemos los motores del mando a una intensidad de vibración cero, logrando así el efecto comúnmente utilizado en la mayoría de los juegos.

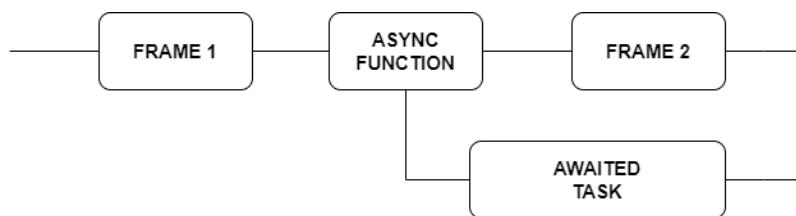


Figura 4.11 – Esquema de funcionamiento de funciones async - Adaptada de [56]

4.5. Sistema de audio

El sistema de audio es responsable de cargar y reproducir los archivos de audio en nuestro juego. Además, debe ser capaz de configurar cada archivo de audio para ajustar el volumen, el tono, el bucle, el sonido 3D y otros parámetros. Nuestro objetivo al implementar este sistema es

proporcionar una forma sencilla y escalable de comunicación entre los *scripts* y el diseñador/programador. De esta manera podemos agregar fácilmente todos los sonidos deseados y ajustar sus parámetros de manera cómoda y conveniente.

Es esencial que todos los *scripts* tengan acceso al sistema de audio, ya que los efectos de sonido se reproducen en diversas situaciones, desde el menú de pausa hasta los pasos del personaje y los sonidos ambientales. Además, si estamos desarrollando un juego en 3D, el audio también debe ser en 3D. Esto significa que los efectos de sonido deben reproducirse en un punto específico del espacio. Si no lo hiciéramos de esta manera, el jugador no podría, por ejemplo, detectar la aproximación o alejamiento de un enemigo.

Para lograr estos objetivos, se ha propuesto crear una estructura utilizando *ScriptableObject*. (con las ventajas que se mencionaron en la [Subsección 4.3.2](#)), basada en tablas hash y un *script* controlador que facilite su acceso desde cualquier parte del código.

En la [Figura 4.12](#) se muestra el diagrama de clases completo del sistema de audio para poder seguir la explicación con una ayuda visual.

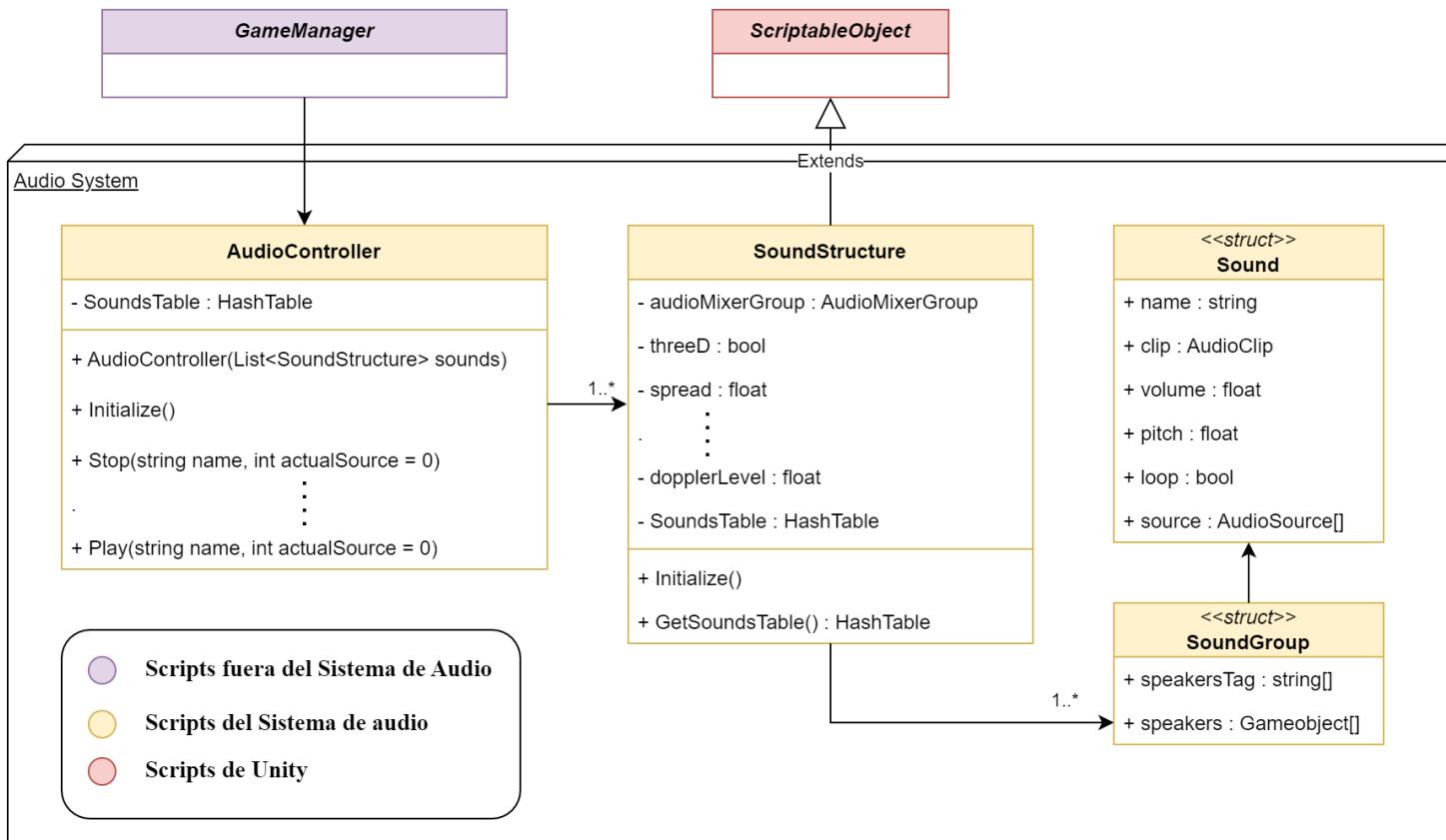


Figura 4.12 – Diagrama de clases del Sistema de Audio

4.5.1. Breve introducción a *AudioSource* de Unity

Para comprender mejor lo que vamos a realizar en esta sección, es conveniente entender cómo Unity maneja las fuentes de audio (*AudioSource*). Básicamente, un *AudioSource* es un elemento en 3D que se puede colocar en un objeto en la *escena* y se utiliza para reproducir, pausar o detener un clip de audio, así como modificar diversos parámetros predefinidos del audio[57]. El *AudioSource* tiene la capacidad de reproducir clips de audio en 2D o en 3D, utilizando la posición relativa en la *escena* para aplicar efectos espaciales como distancia, propagación o efecto Doppler, entre otros. Para que la reproducción de audio tenga sentido en un entorno 3D, se requiere la presencia de un componente llamado *AudioListener*, que establece la posición del oyente en la *escena* y permite aplicar correctamente los efectos espaciales. Por último, si deseamos tener un control conjunto sobre varios *AudioSource*s para modificar el volumen o añadir efectos, podemos utilizar un mezclador de audio (*AudioMixer*), que se configura en la estructura de directorios del proyecto y se asigna a cada *AudioSource* individualmente.

4.5.2. Estructura del sistema

Al abordar un sistema de audio, existen ciertos requisitos indispensables que su estructura debe cumplir:

- La adición, modificación o eliminación de sonidos en la estructura debe ser cómoda y no requerir cambios en numerosos *scripts*, ya que es una tarea frecuente.
- El acceso a estos sonidos en el código debe ser rápido y no ralentizar el ciclo de juego.
- La estructura debe ser escalable, de manera que un mayor número de sonidos no resulte en una pérdida de eficiencia.
- Sería útil contar con un acceso mediante clave o índice para facilitar la reproducción de sonidos en otras partes del código.

Para cumplir con todas estas necesidades es necesario tener un conjunto compacto y accesible de *AudioSource*s fuera de la ejecución del juego, esto es gestionado por las estructuras *Sound* y *SoundGroup*. También necesitamos una estructura de datos escalable y eficiente que nos permita acceder a todos estos *AudioSource*s en el código.

Sin entrar en detalles sobre el funcionamiento específico, el sistema se basa en un *ScriptableObject* llamado *SoundStructure*, que cumple las siguientes funciones:

- Serializar las opciones 3D y un conjunto de *SoundGroups* con los que seremos capaces de definir todos los sonidos fuera de ejecución.
- Define la estructura de datos (tabla *hash*) que establece una relación entre los sonidos y sus ID.
- Inicializa todos los sonidos como *AudioSource*s en un punto específico del espacio de la *escena*.
- Asigna un *AudioMixerGroup* [58] a todos los sonidos para poder cambiar sus parámetros en conjunto desde un *AudioMixer*.

En la [Figura 4.13](#) se representa un esquema de la estructura del sistema que se puede tener presente durante toda la sección.

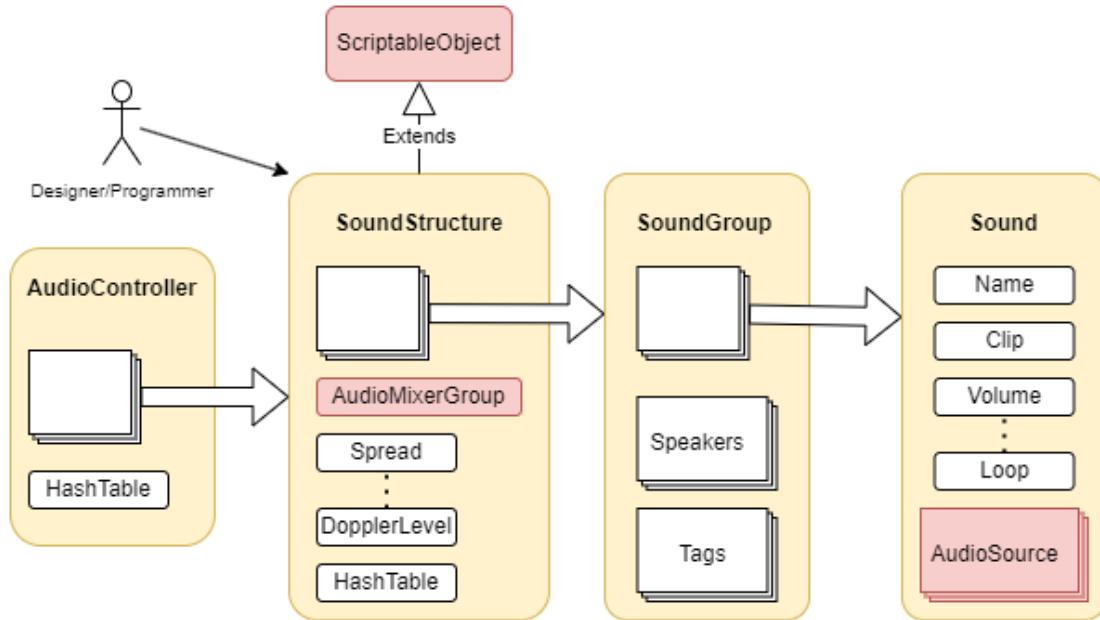


Figura 4.13 – Estructura del sistema de audio

4.5.2.1. *Sound* y *SoundGroup*

Las estructuras presentadas en esta sección serán responsables de definir todos los sonidos que deben inicializarse en la *SoundStructure* al comienzo de la ejecución del juego. Como mencionamos anteriormente, los *AudioSources* deben colocarse en el objeto de la *escena* donde deseamos que se reproduzcan. Por lo tanto, no podemos tenerlos todos juntos en una única estructura, ya que, por ejemplo, algunos estarían en el enemigo, otros en el jugador o en algún objeto ambiental como una radio. La idea, entonces, es crear estructuras que contengan los parámetros de inicialización de un *AudioSource* y una referencia al mismo.

La estructura *Sound* se encarga de definir un sonido específico en el juego y contiene un nombre que lo identificará de manera única. Este nombre será asignado por el programador o el diseñador y nos permitirá acceder a dicho sonido. Sin embargo, surgen un par de problemas en la creación de esta estructura. En primer lugar, necesitamos una forma de crear los *AudioSources* al comienzo de la ejecución y asignarles los parámetros definidos en *Sound*, en el objeto de la *escena* donde deseamos que aparezcan. En segundo lugar, cómo podemos lograr que un mismo sonido, con la misma configuración, aparezca en dos objetos distintos de la *escena*. Esto es especialmente común en objetos ambientales.

La solución a esto es la creación de la estructura *SoundGroup*. Esta estructura está compuesta por un conjunto de sonidos (*Sound*), así como dos listas: *Speakers* y *Tags* (Figura 4.13). La lista *Speakers* contendrá los objetos en los que deben crearse los *AudioSources* correspondientes al conjunto de sonidos, mientras que la lista *Tags* contiene las etiquetas que identifican a dichos objetos. Esto se hace de esta manera debido a que en un *ScriptableObject* no podemos referenciar a objetos de la *escena* fuera de la ejecución. Por lo tanto, la forma de abordar este problema es asignar una etiqueta y, durante la inicialización, utilizar el método de Unity *FindGameObjectsWithTag* [59] para encontrar los objetos en la *escena* y asignarles los *AudioSources*. Recopilando, cada *SoundGroup* tiene una lista de sonidos, y cada sonido de esta lista se asigna a los objetos identificados por sus etiquetas. Ahora cada sonido, en lugar de tener una única referencia a un

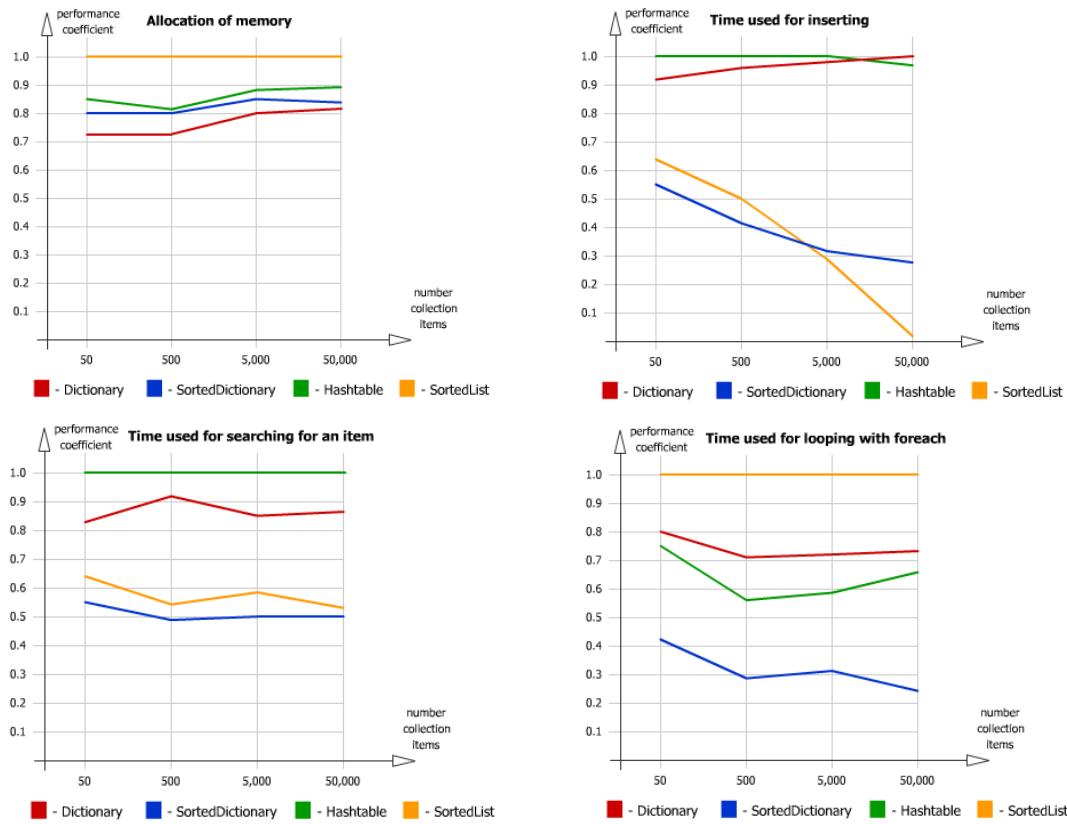


Figura 4.14 – Gráficas de rendimiento (diccionario vs tabla hash)

AudioSource, tiene una lista de ellas (el tamaño de la lista se define en la inicialización cuando sepamos a cuantos objetos se debe asignar el sonido).

4.5.2.2. Estructura de datos (tabla hash)

Es necesario tener una forma de acceder a las referencias de los *AudioSources* en el código, ya que la estructura anterior tenía como objetivo brindar una representación visual. Colecciones como listas, listas ordenadas, pilas, colas o diccionarios ordenados no tienen mucho sentido en este caso, ya que no se requiere un control preciso sobre el ordenamiento de los elementos ni una estructura con un orden específico, en cambio, se necesita un acceso rápido y una identificación mediante ID. A primera vista, un diccionario o una tabla *hash* cumplen con estos requisitos. Para estar seguros, podemos basarnos en un test donde se analiza el rendimiento de estas estructuras [60] (Figura 4.14).

Al analizar la eficiencia en asignación de memoria y en el tiempo utilizado para buscar un elemento, la tabla *hash* supera al diccionario. Por otro lado, el diccionario se destaca en cuanto al tiempo necesario para insertar elementos y para iterar mediante un bucle *foreach*. Si consideramos nuestras necesidades específicas, la inserción de elementos solo se realiza una vez al comienzo de la ejecución del juego, ya que no se añaden nuevos sonidos durante su ejecución. Además, la utilización de bucles *foreach* solo se lleva a cabo en momentos puntuales y específicos en los que, por ejemplo, necesitemos detener todos los sonidos que se estén reproduciendo. Asimismo, la búsqueda de elementos se realiza de manera continua para reproducir o realizar alguna acción en relación a nuestros sonidos. En ese sentido la eficiencia en la búsqueda es una prioridad y la tabla *hash* sería una elección adecuada.

Cada *SoundStructure* tendrá su propia tabla *hash*, que se inicializará junto con la inicialización de los *AudioSources*, estableciendo una conexión entre el nombre asignado a los sonidos y su referencia al *AudioSource*. Para los sonidos que estén asignados a varios *AudioSources*, también se deberá especificar el nombre del objeto al que se ha asignado. Por ejemplo, si el sonido con nombre “punch” se ha asignado a dos personajes distintos, la ID del primero tendrá el nombre “punch_personaje0” y el segundo “punch_personaje1” (esta nomenclatura la debemos tener en cuenta a la hora de indicar que sonido vamos a reproducir en cada *script* del juego).

Se podría haber utilizado el índice del vector de *AudioSources* de cada sonido en lugar del nombre del objeto, pero tomamos la decisión de utilizar el nombre del objeto y asignar índices específicos debido a que la búsqueda basada en *tags* puede variar entre diferentes ejecuciones del proyecto o una vez que se haya realizado la *build*. Esta elección se hizo para prevenir posibles errores y garantizar la estabilidad del sistema.

4.5.3. Controlador del sistema

Después de tener una estructura bien definida, accesible para cualquier diseñador/programador, queda la cuestión de cómo acceder a través de código a todos esos sonidos que se han definido. La respuesta a esta pregunta es el *script* *AudioController*. Como se aprecia en la [Figura 4.13](#), este *script* contiene una lista de todas las *SoundStructures* a las que se requiera acceso. Este *script* es el único que está vinculado con el *GameManager*, como se aprecia en el diagrama de clases ([Figura 4.12](#)), y, por tanto, es el único *script* cuyas funcionalidades se permite manejar desde código.

Lo primero que necesitamos hacer al comenzar la ejecución es inicializar cada uno de los elementos de la lista del controlador, de forma que se definan todos los *AudioSources* en el lugar adecuado y las tablas *hash* que contienen cada uno de los conjuntos de sonidos. Además, debemos inicializar un componente más del controlador de audio: su propia tabla *hash*, que será la unión de todas las tablas *hash* definidas en las *SoundStructures*. Como resultado, obtendremos una única tabla *hash* que contiene todos y cada uno de los sonidos identificados por su ID. Es decir, dado un ID, podemos obtener la referencia a su *AudioSource*. Mediante esta tabla, que como vimos ofrece una velocidad de acceso mínima entre las colecciones disponibles, podemos usar todas las opciones de los *AudioSources*, como reproducir un sonido, pausarlo, detenerlo, ajustar su volumen, tono u otros parámetros.

Para lograr esto, se definen en el controlador de audio funciones específicas que nos permiten realizar cada una de estas acciones. Estas funciones solo requieren como parámetros el nombre del sonido y la funcionalidad específica que se desea ejecutar. Por ejemplo, si queremos cambiar el volumen del sonido “punch_personaje0”, simplemente debemos pasar ese nombre como parámetro junto con el nuevo volumen deseado. La función se encargará de encontrarlo en la tabla *hash* (si es que existe) y reducir o aumentar su volumen.

4.5.4. Conclusiones de la implementación, ventajas e inconvenientes

Aunque parezca algo complejo, esta definición de sonidos nos permite contemplar una gran variedad de situaciones y tener un sistema de audio altamente personalizable. A continuación, se muestran las ventajas e inconvenientes que pueden surgir con esta definición ([Tabla 4.1](#)) y un ejemplo de como se representa visualmente ([Figura 4.15](#)).

Ventajas	Inconvenientes
Toda la configuración de audio del juego se encuentra en archivos fuera de la escena y en el árbol de directorios, lo que facilita su modificación.	Los sonidos no se pueden modificar en tiempo de ejecución desde la <i>SoundStructure</i> (solo se utiliza para la inicialización), pero esto se puede hacer en código, como veremos en el controlador.
Cada <i>SoundStructure</i> tiene su propia configuración de audio 3D y su propio <i>AudioMixerGroup</i> asignable, lo que nos permite tener automáticamente todos sus sonidos en un <i>AudioMixer</i> y ajustar, por ejemplo, el volumen de forma colectiva.	La inicialización se realiza con etiquetas de Unity, lo cual puede no ser el proceso más eficiente según las recomendaciones de rendimiento de Microsoft [61].
Esta estructura nos permite tener los efectos de sonido, la interfaz de usuario, la música y cualquier otra categoría independientes entre sí.	Dado que todo el sistema está centralizado, la inicialización debe realizarse al inicio del juego y en cada cambio de escena , ya que el <i> AudioSource</i> depende de la escena en la que se encuentre.
Al tener parámetros para cada sonido, nos permite ajustar de manera específica las variaciones de volumen o tono.	Todo el sistema está construido para inicializar sonidos a objetos que se encuentran en la escena antes de iniciar el juego, no está pensado para nuevos objetos que requieran sonido y se crean durante la ejecución.
La tabla <i>hash</i> nos brinda un acceso único, eficiente y escalable a los sonidos, independientemente de la estructura visual.	

Cuadro 4.1 – Ventajas e inconvenientes (Estructura del sistema de audio)

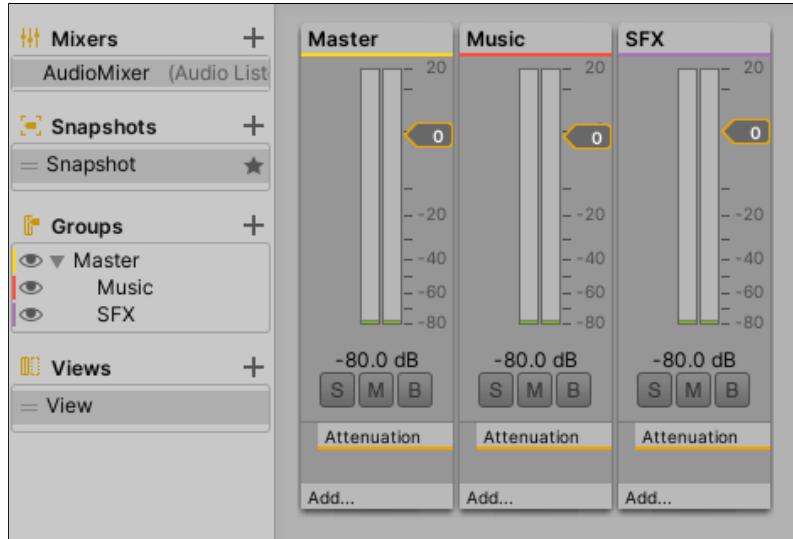
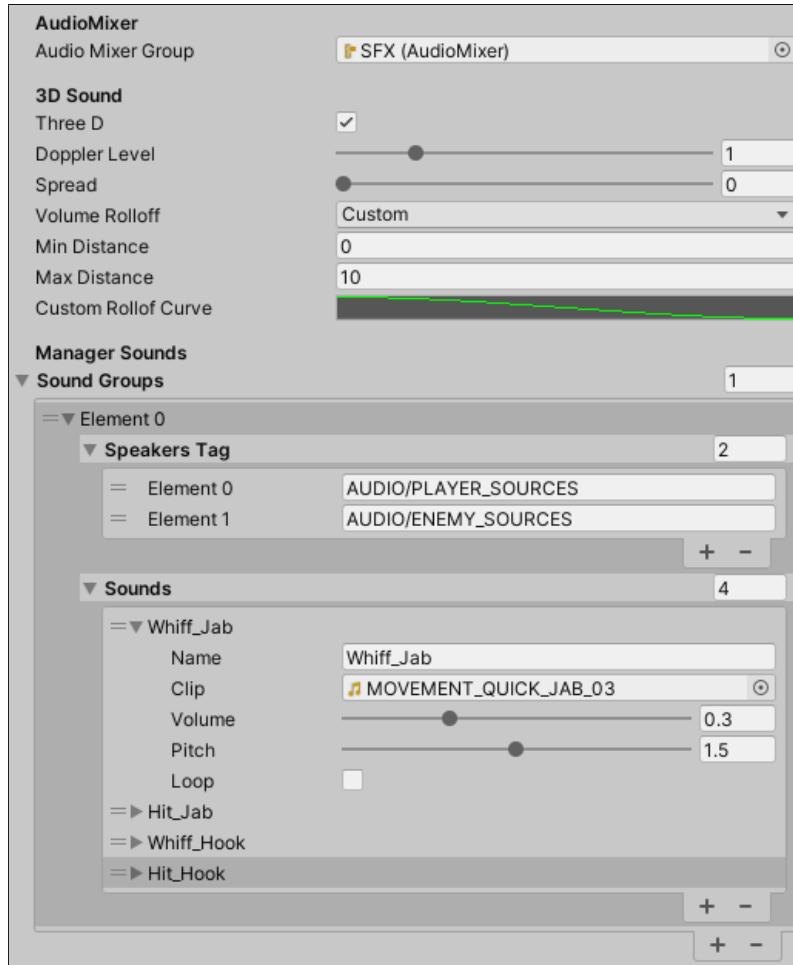
4

Aunque la inicialización pueda ser un poco lenta, como mencionamos en los inconvenientes, creemos que es más importante tener un sistema fácilmente manipulable que optimizar la eficiencia al máximo. Además, esta inicialización solo se realiza al comienzo de la [escena](#) y luego se utilizan referencias, por lo que el ciclo de juego no se ve afectado y se pueden utilizar pantallas de carga para evitar cualquier pérdida, que en cualquier caso no debería ser grave. Por otro lado, el inconveniente de no poder modificar los sonidos en tiempo de ejecución solo se aplica a la *SoundStructure*, ya que siempre se puede navegar hasta los objetos concretos de la [escena](#) donde se hayan inicializado los *AudioSource* y modificarlos.

El último inconveniente mencionado se relaciona más con la implementación actual que con la implementación en sí misma. Queremos destacar que debido a la forma en que todo está definido, es muy sencillo añadir funcionalidades que nos permitan inicializar nuevos sonidos durante la creación de nuevos objetos. Este tema se abordará en detalle en el [Capítulo 6](#).

4.6. Sistema de guardado

El sistema de guardado será el encargado de almacenar la información relevante de nuestro juego entre ejecuciones. Este sistema va a definir la inicialización y el estado de nuestro juego en un momento dado, haciendo indispensable su acceso desde cualquier *script* que inicialice elementos al comienzo de una [escena](#). Para asegurar el correcto funcionamiento de este sistema,

(a) *AudioMixer* (conjunto de *AudioMixerGroups*)(b) *Inspector de SoundStructure***Figura 4.15 – Ejemplo de estructura de sistema de audio**

necesitaremos los siguientes requisitos:

1. Un método de almacenamiento de datos fuera de la ejecución del juego. Esto nos permitirá guardar la información necesaria de manera persistente, para que pueda ser recuperada en futuras sesiones de juego. Dicho método de almacenamiento puede ser un archivo en disco, una base de datos u otra forma de almacenamiento externo.
2. Una forma de especificar los datos de guardado por defecto. Esto es útil en situaciones como querer restablecer los datos guardados o comenzar una nueva partida desde cero. Al establecer valores predeterminados, se facilita el reinicio del juego o la configuración inicial de los datos de guardado.
3. Necesitaremos un acceso global al sistema de guardado a nivel de código. Esto nos permitirá acceder a las funciones y datos del sistema de guardado desde cualquier parte del juego.

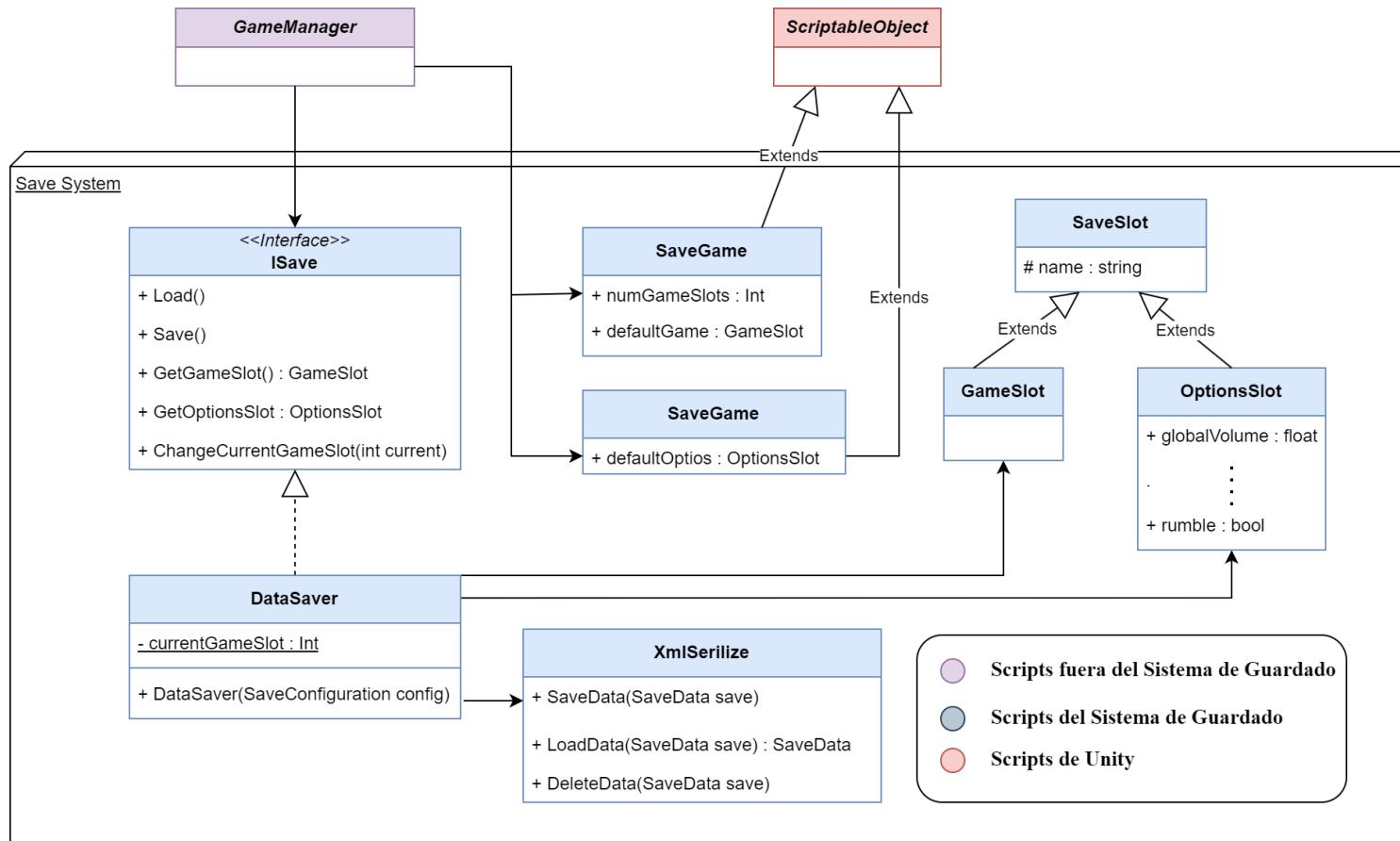


Figura 4.16 – Diagrama de clases del sistema de guardado

En la [Figura 4.16](#) se muestra el diagrama de clases completo del sistema de guardado para poder seguir la explicación con una ayuda visual.

4.6.1. Estructura y funcionamiento del sistema

La clave de la implementación que vamos a abordar es dividir en todo momento, los datos a los que accedemos durante la ejecución del juego (estructuras locales) y los datos persistentes que permanecen entre ejecuciones.

Los datos persistentes se almacenan en archivos y solo necesitan ser cargados al inicio de la ejecución. Posteriormente cuando el jugador decida guardar o mediante autoguardado debemos asegurarnos de que estos archivos siempre tienen una versión actualizada. Por otro lado durante la ejecución del juego se accede y almacenan los datos en alguna estructura que defina todas las variables que necesitamos almacenar. El funcionamiento del sistema viene resumido y esquematizado en el diagrama de flujo de la [Figura 4.17](#).

4.6.2. Estructura de guardado local

Para los datos a los que accedemos durante la ejecución del juego, hemos creado un *script* llamado *SaveSlot* que se encarga de almacenarlos. Tanto la clase *GameSlot* como la clase *OptionsSlot* heredan de la clase *SaveSlot* y se encargan de almacenar los datos específicos del juego y sus opciones. Para la creación de los datos de guardado por defecto es de utilidad asegurarnos de que podemos realizar una copia profunda de estas clases. La implementación de la interfaz *ICloneable* [62] nos permite definir nuestra propia función *Clone* para realizar copias profundas de los datos que estamos tratando. Al implementar la interfaz en *SaveSlot*, cualquier clase que herede de ella también debe implementarla, asegurando así que podemos clonar correctamente los datos.

En este punto, es posible que surja la pregunta de por qué es necesario una clase padre como *SaveSlot*. La respuesta es que esta clase es abstracta, lo que significa que no se puede instanciar por sí sola, pero resulta útil cuando queremos acceder a un *slot* sin importar el tipo específico que sea, por ejemplo, cuando queremos guardar los datos en un archivo lo podemos hacer con una única función a la que se pase como parámetro un *SaveSlot*.

En la clase *DataSaver*, se ha creado una instancia estática de *OptionsSlot*, que almacenará las opciones del juego para que el jugador no tenga que cambiarlas cada vez que lo inicie. También se ha definido una lista estática de *GameSlot*, ya que puede haber más de un archivo de guardado del juego. Además, un entero estático se utiliza para indicar el *GameSlot* que se está utilizando actualmente. Con esta estructura, tenemos un *script* al que podemos acceder y sabemos que solo contendrá la instancia correcta de los datos de guardado locales, facilitando la gestión y acceso a los datos guardados del juego. *DataSaver* también tiene definidas funciones para ordenar al serializador que almacene los datos locales en un archivo.

Ahora necesitamos alguna forma de comunicación entre *GameManager* y *DataSaver* para que cualquier *script* pueda acceder a los datos y las funciones definidas por *DataSaver*. Pensamos que debíamos tener alguna manera de desacoplar el acceso global del módulo de guardado y así poder definir múltiples instancias de “*DataSaver*”, como por ejemplo, con guardado en la nube u otro tipo de serialización. La solución viene dada por el principio de inversión de dependencia de SOLID, que en pocas palabras nos dice que las capas que queremos desacoplar deben depender

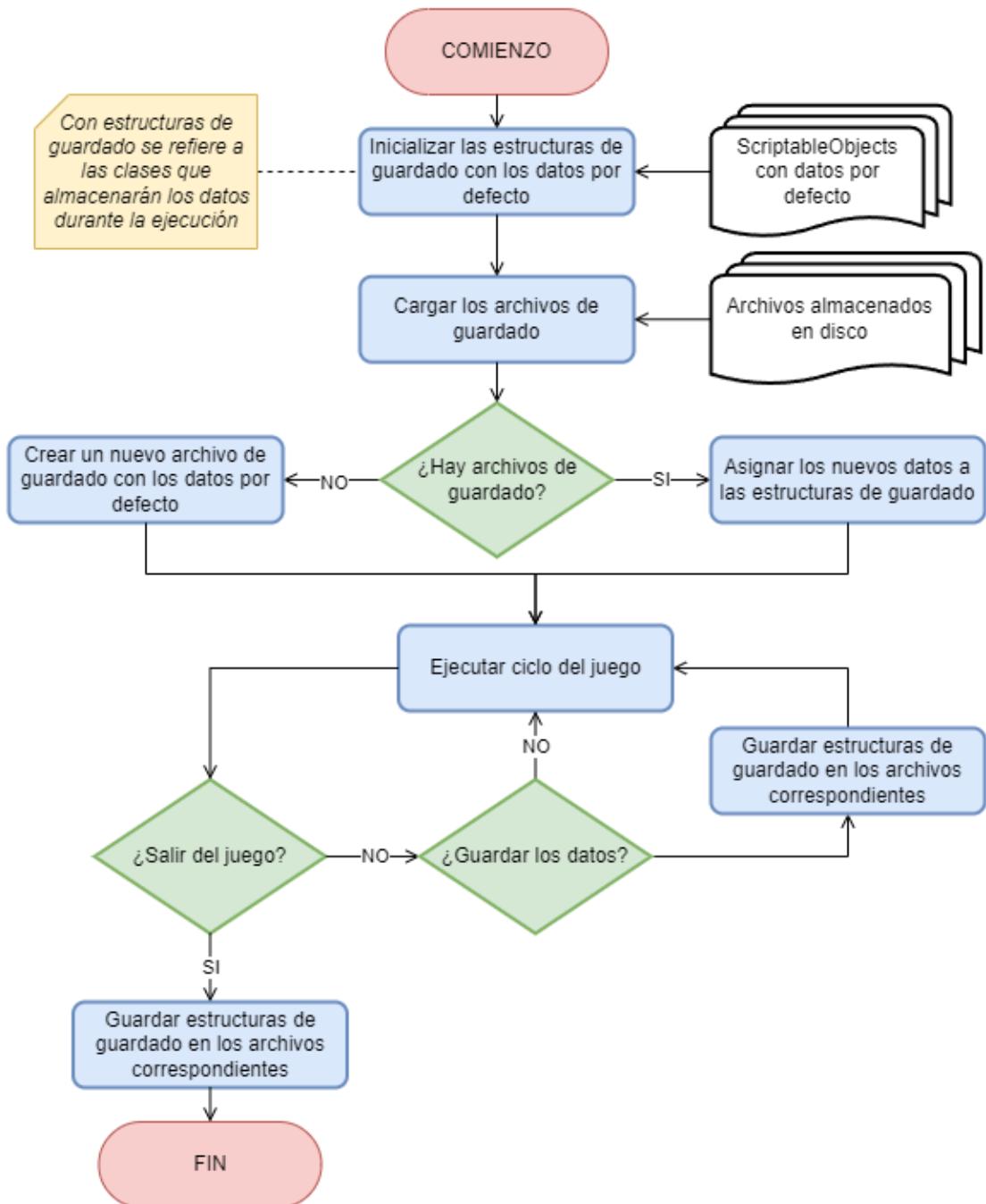


Figura 4.17 – Diagrama de flujo del sistema de guardado

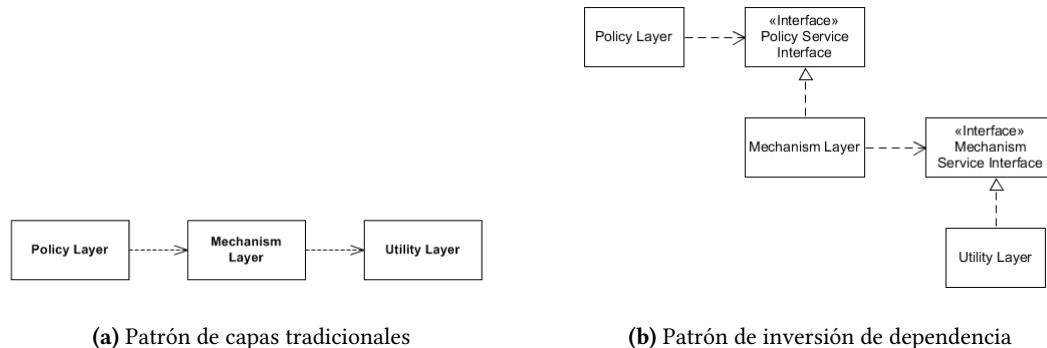


Figura 4.18 – Ejemplo de Principio de inversión de la dependencia

de abstracciones que describan el comportamiento que necesitan las capas de nivel superior [63] ([Figura 4.18](#)). En nuestro caso, esto se aplica creando una interfaz implementada por *DataSaver* e instanciada en *GameManager*. Esta interfaz define solo las funciones que todo *DataSaver* debe tener y a las que necesitamos acceder desde *GameManager*. De esta manera, las clases solo son accesibles a través de la interfaz, y gracias a las conversiones implícitas del compilador[64], podemos instanciar en *GameManager* cualquier tipo de dato que implemente esa interfaz.

Con este enfoque, logramos un sistema flexible y desacoplado que nos permite cambiar o añadir diferentes tipos de *DataSaver* sin afectar otras partes del código. Además, garantizamos una comunicación clara y definida entre *GameManager* y *DataSaver*, lo que facilita el acceso a los datos de guardado y las funciones relacionadas en todo el juego.

4.6.3. Guardado por defecto

En la introducción de la sección, se mencionó la importancia de tener datos de guardado por defecto antes de realizar cualquier cambio en el juego. Estos datos de guardado son fundamentales para inicializar las estructuras de almacenamiento locales. De esta manera, si no existe ningún archivo de guardado o si han sido eliminados, siempre contaremos con una configuración inicial. El diagrama de flujo ([Figura 4.17](#)) representa claramente este proceso, mostrando cómo los datos de guardado por defecto se utilizan para establecer el punto de partida antes de cualquier interacción con los archivos de guardado.

Para lograr esto, se han definido instancias de *GameSlot* y *OptionsSlot* en dos *ScriptableObject*s separados, los cuales proporcionan al diseñador o programador la interfaz de *inspector* necesaria para definir las variables por defecto. En el caso de los *GameSlot*, también se ha añadido una variable adicional para especificar el número máximo de *GameSlots* permitidos. Ambos *ScriptableObject*s deben ser pasados al constructor de *DataSaver* en *GameManager*, asegurándonos así de que las estructuras locales se inicialicen de manera correcta. La interfaz *ICloneable*, implementada previamente en las estructuras locales, permite una inicialización sencilla en *DataSaver* mediante la clonación de las estructuras por defecto pasadas al constructor y su asignación a las estructuras locales.

4.6.4. Serialización XML (guardado permanente)

Hemos decidido que el guardado permanente de nuestro sistema se realice en archivos de nuestro disco duro, independientes del juego, permitiendo así crear copias de esos guardados o

acceder a ellos sin conexión a internet. Además, siempre podemos optar por un guardado en la nube de los archivos que creemos.

Para lograr un guardado permanente, necesitamos alguna forma de serializar nuestras estructuras locales, las cuales cambian durante la ejecución, y almacenarlas en algún lenguaje que pueda ser guardado en archivos de nuestro disco duro. Existen diversos lenguajes que nos permiten esta serialización, como YAML (YAML Ain't Markup Language), binario, XML (Extensible Markup Language), JSON (JavaScript Object Notation), Binary XML (BXML), e incluso bases de datos como SQLite. No obstante, hemos decidido enfocarnos en los dos formatos más populares: XML o JSON. Ambos lenguajes tienen sus ventajas e inconvenientes, por lo que es importante analizarlos para determinar cuál se adapta mejor a nuestro caso [65] ([Tabla 4.2](#)).

XML	JSON
Estructura jerárquica: XML utiliza etiquetas para organizar los datos en una estructura jerárquica, lo que facilita la representación de datos complejos y su comprensión humana.	Ligero: JSON es más ligero que XML, ya que utiliza una estructura más simple basada en pares clave-valor, lo que resulta en archivos más pequeños y tiempos de carga más rápidos.
Extensibilidad: Como sugiere su nombre, XML es altamente extensible, lo que significa que es posible agregar nuevos elementos o atributos sin romper la estructura existente.	Procesamiento rápido: Debido a su simplicidad, JSON es más fácil y rápido de procesar, lo que puede mejorar el rendimiento en aplicaciones de tiempo real como los videojuegos.
Validación: XML se puede validar fácilmente mediante esquemas XML, lo que permite asegurarse de que los datos estén bien formados y cumplan con ciertas reglas definidas.	Legibilidad: JSON es fácilmente legible por humanos debido a su sintaxis más simple y concisa.
Mayor tamaño de archivo: XML tiende a ser más verbose (detallado) en comparación con JSON, lo que puede resultar en archivos más grandes y, por lo tanto, un mayor tiempo de carga.	Menor extensibilidad: JSON puede ser más rígido en comparación con XML, ya que agregar nuevos campos o elementos podría requerir modificaciones en la estructura existente.
Procesamiento más lento: Debido a su naturaleza más compleja, el procesamiento de archivos XML puede ser más lento en comparación con JSON.	Menos soporte para comentarios: JSON no admite comentarios, mientras que XML sí. Los comentarios pueden ser útiles para documentar el código y hacer notas adicionales.

Cuadro 4.2 – XML vs JSON

Tras analizar las características de cada formato, hemos decidido utilizar XML debido a su mayor extensibilidad y facilidad para representar datos complejos, lo que nos permite abarcar numerosos géneros de videojuegos independientemente de los datos que necesiten. Además, contamos con un conocimiento previo del lenguaje que facilita su implementación. No obstante, esto no significa que debamos descartar los otros lenguajes de forma absoluta. La elección dependerá de varios factores, como la complejidad de los datos, los requisitos de rendimiento, la facilidad de lectura y escritura, y la compatibilidad con las herramientas y plataformas existentes. Cada formato tiene sus propias fortalezas y debilidades, por lo que es importante evaluar las necesidades específicas del proyecto antes de tomar una decisión.

Una vez elegido el formato de serialización de clases, creamos un *script* encargado de serializar y deserializar un *SaveSlot* (*XMLSerialize*). La implementación de los métodos se puede seguir con facilidad en la documentación de Microsoft y creemos que no requiere más explicación [66]

[67]. Por último, es importante mencionar que la ruta donde almacenaremos los archivos de guardado viene dado por la variable estática “Application.persistentDataPath”, que, dependiendo del sistema operativo, los almacenará en el directorio habitual donde se encuentran los datos persistentes de las aplicaciones[68].

4.7. Gestión de escenas

Uno de los aspectos más importantes al desarrollar un juego en Unity es la gestión adecuada de las [escenas](#). Las escenas son como lienzos en los que se desarrolla cada nivel, pantalla o área del juego, y es esencial poder transicionar entre ellas de manera cómoda y eficiente, manteniendo la integridad de los aspectos que deben inicializarse entre cada cambio y controlando el tipo de transición que se realiza. Al cargar una nueva escena, es fundamental tener en cuenta que los objetos y recursos de la escena actual pueden ser destruidos o reemplazados, dependiendo de cómo se configure la carga. Por lo tanto, es crucial realizar una correcta administración de recursos y datos compartidos entre [escenas](#) para evitar pérdidas de información y asegurar una transición suave entre ellas.

En esta sección, abordaremos los aspectos fundamentales que debe contener un controlador de escena, y cómo adaptar dicho controlador a la arquitectura que estamos desarrollando mediante el uso de [ScriptableObjects](#). y el [GameManager](#). Además, definiremos un funcionamiento que nos permita implementar transiciones animadas entre [escenas](#), buscando proporcionar una experiencia más inmersiva al cambiar entre distintas áreas del juego.

En la [Figura 4.19](#) se muestra el diagrama de clases completo del controlador de escenas para poder seguir la explicación con una ayuda visual.

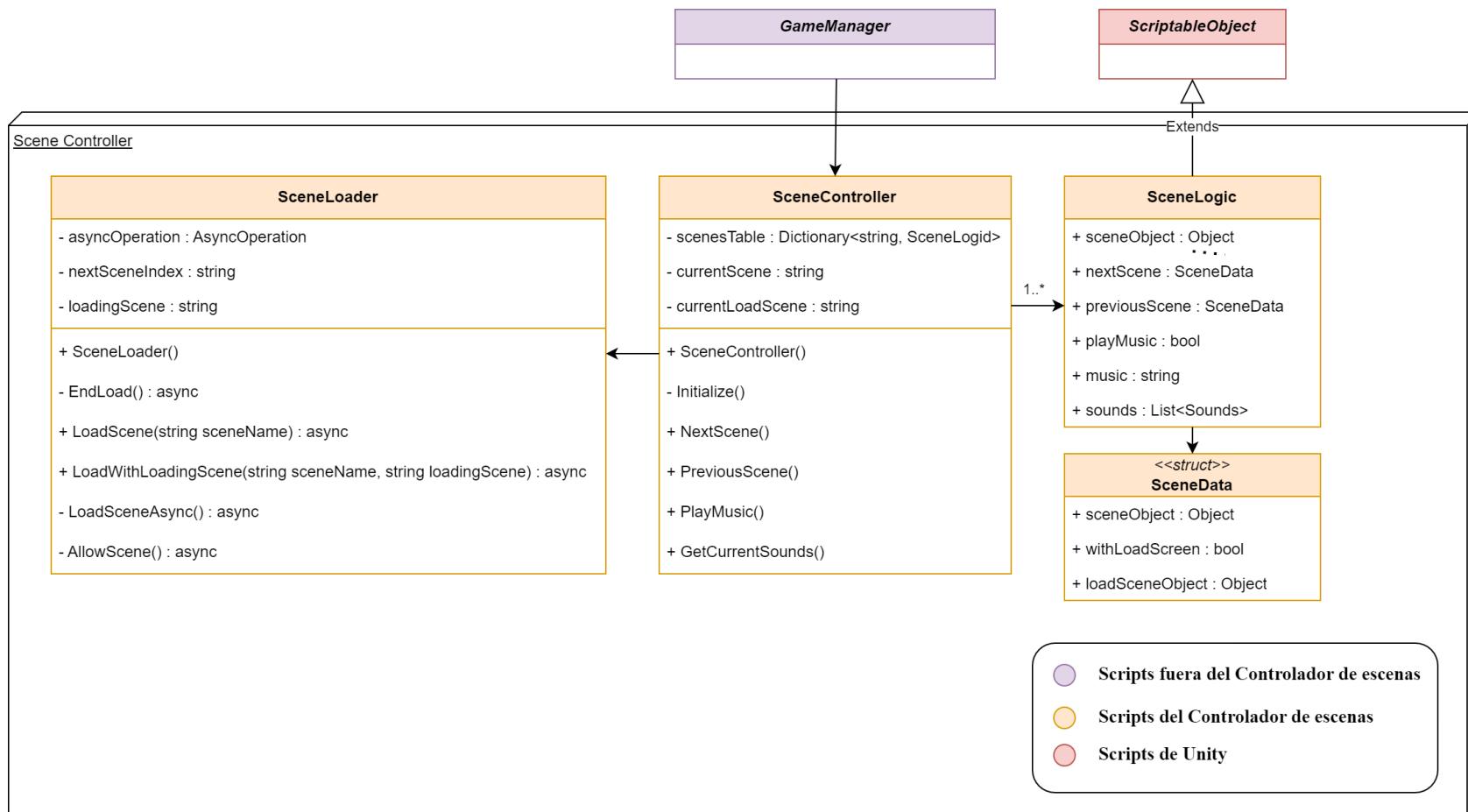


Figura 4.19 – Diagrama de clases del controlador de escenas

4.7.1. Breve introducción a SceneManager de Unity

Durante esta sección, estaremos utilizando la clase *SceneManager* [69] en Unity, la cual nos proporciona las herramientas necesarias para gestionar e identificar escenas en nuestro proyecto. Pondremos el foco de atención en tres funciones clave:

- ***GetActiveScene()***. Esta función nos permite obtener la referencia a la escena que está actualmente cargada y en la que se está reproduciendo el juego. Es útil para obtener información sobre la escena en la que nos encontramos, como su nombre, índice o ruta de archivo, y así realizar acciones basadas en esa información.
- ***LoadScene(String name)***. La función *LoadScene()* nos brinda la capacidad de cargar una escena específica por su nombre. Para utilizar esta función de manera efectiva, es importante asegurarse de haber agregado las escenas que deseamos cargar a la lista de escenas en el “Build Settings” de Unity. Al llamar a esta función, Unity iniciará la carga de la escena y realizará una transición inmediata a ella.
- ***LoadSceneAsync(String name)***. Esta función es especialmente útil cuando queremos cargar una escena en segundo plano, sin que la transición ocurra de manera inmediata. La carga asíncrona nos permite mantener el juego en ejecución mientras la escena se está cargando. Esto es beneficioso para evitar tiempos de espera prolongados durante las transiciones entre escenas y mantener una experiencia de juego fluida y sin interrupciones. Al igual que con *LoadScene()*, es necesario haber agregado la escena objetivo a la lista de escenas en “Build Settings”.

Además, vamos a suscribir funciones con frecuencia al evento *sceneLoaded* de *SceneManager*, que se invoca cuando una escena ha sido cargada.

4.7.2. Configuración de escenas

Vamos a crear una clase llamada *SceneLogic*, que nos permita tener toda la configuración adicional que no nos proporciona la propia clase *Scene* de Unity, pero que necesitamos al inicializar o cargar una escena. Deseamos que esta clase sea configurable desde fuera de la ejecución de forma sencilla y cómoda, por lo tanto, al igual que en otras ocasiones, hereda de *ScriptableObject*. En la clase *SceneLogic* (ejemplo en [Figura 4.20](#)), incluiremos información relevante como qué escena va antes y después de la misma, si hay que reproducir música en la escena, qué *SoundStructures* debemos cargar al iniciar la escena o si la transición se realiza de forma asíncrona o no. Esto nos permitirá tener un mayor control sobre la configuración y el comportamiento de cada escena en nuestro proyecto de Unity, manteniendo la flexibilidad para ajustar estos parámetros según las necesidades del juego.

4.7.3. Cargador de escenas y transiciones

Todo el funcionamiento relacionado con cargar las escenas y transiciones se lo vamos a delegar a la clase *SceneLoader*. En esta clase, crearemos las funciones que nos permitan cargar una escena por su nombre o cargar una escena con una pantalla de carga ([Figura 4.21](#)).

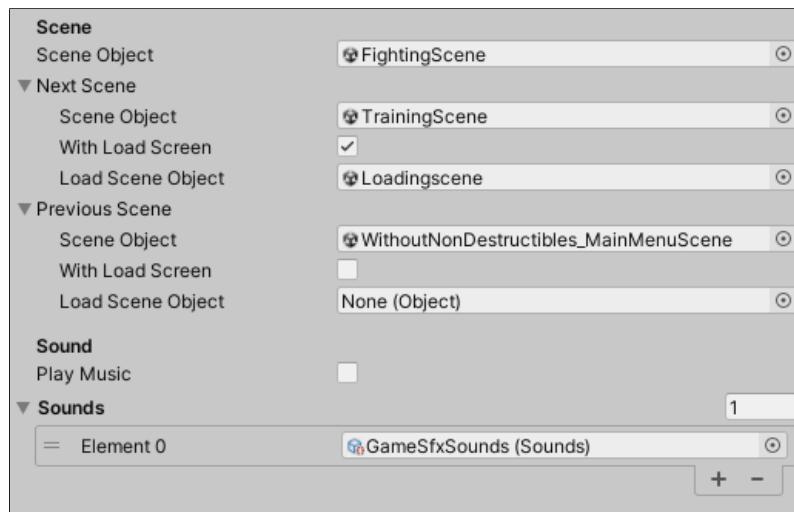


Figura 4.20 – Ejemplo de inspector de SceneLogic

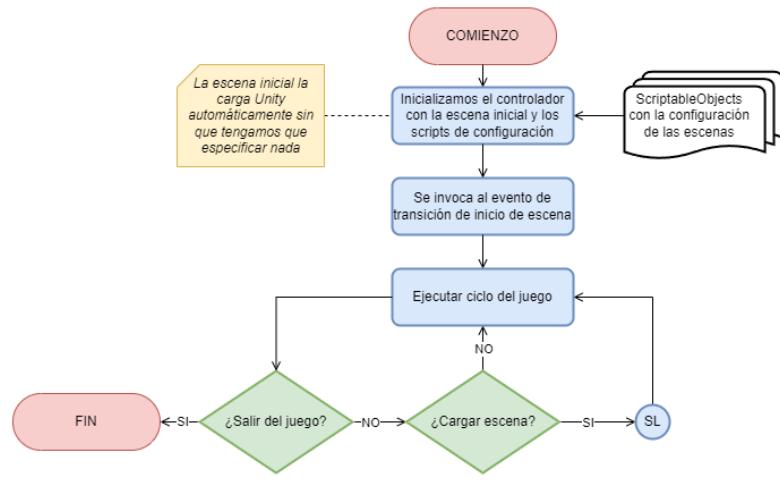
Para la carga de escenas no asíncronas, simplemente utilizaremos la función *LoadScene* de *SceneManager*. En el caso de la carga de escenas de manera asíncrona, utilizaremos una escena de pantalla de carga como intermediario para poder seguir el progreso de la carga en segundo plano. Podemos monitorear el progreso de la carga utilizando una variable *asyncOperation* que devuelve la función *LoadSceneAsync*. Es importante deshabilitar la activación automática de la escena (*allowSceneActivation*) en la operación asíncrona para tener un control total sobre cuándo queremos pasar de la pantalla de carga a la escena principal.

Nos podemos valer del evento *sceneLoaded* que proporciona *SceneManager* para tener un punto centralizado donde gestionar las acciones que deben llevarse a cabo cuando una escena ha sido completamente cargada. En la clase *SceneLoader*, suscribimos una función para aplicar la transición de inicio de escena cada vez que cargamos una escena, sin tener que especificarlo para cada una en particular. Además, implementamos otra función para comprobar si nos encontramos en una pantalla de carga y, en caso afirmativo, comenzar a cargar la siguiente escena de forma asíncrona.

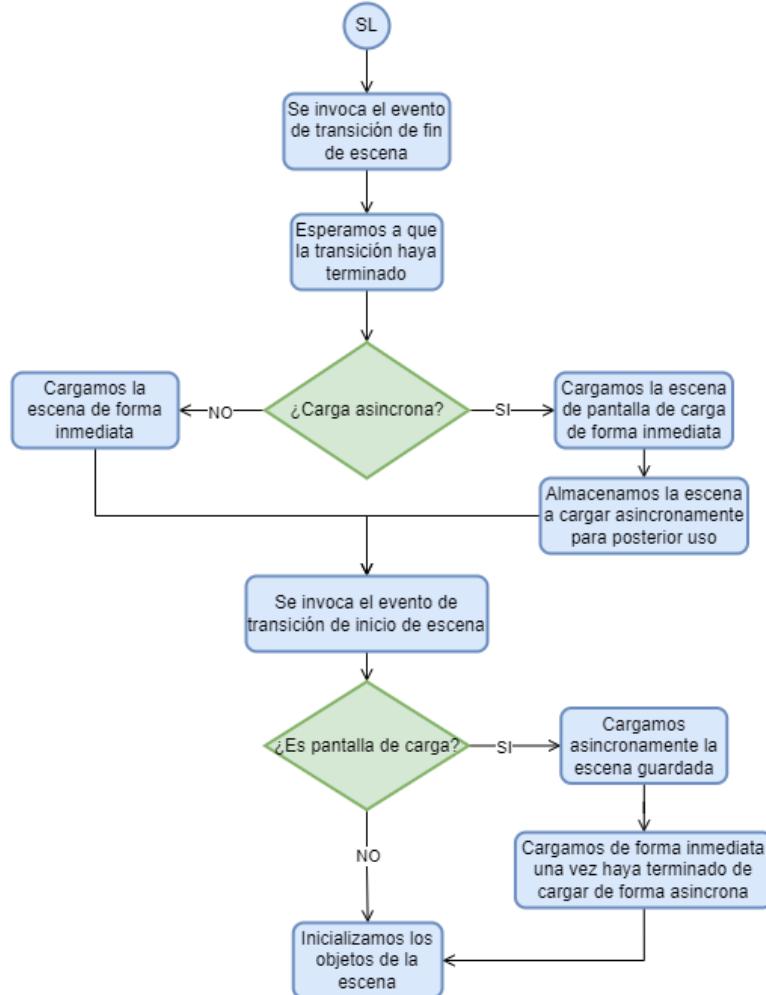
4.7.3.1. Gestión y Control de Transiciones Animadas

Para la gestión y control de las transiciones animadas, se han empleado eventos de C#, previamente discutidos en el inicio de este capítulo, y funciones *async*, que también fueron abordadas en la sección dedicada al sistema de *inputs*.

- Se han implementado dos eventos que se activan al inicio y fin de cada transición de escena. De este modo, se permite la suscripción y desuscripción de todas las funciones necesarias a dichos eventos, lo que posibilita la creación de transiciones específicas para cada escena, siguiendo el Patrón de diseño Observador [46]. Estos eventos pueden ser considerados estáticos debido a que solo existe una instancia de la clase *SceneLoader* en todo momento, encargada de cargar todas las escenas del juego.
- Se establece un único requisito para las funciones que se suscriben a los eventos, el cual consiste en que deben devolver una tarea (*task*) de C#. De esta manera, cualquier función *async* puede realizar una operación de *await* y esperar a que la transición haya concluido. Con esta estructura, es posible esperar a que las transiciones finalicen antes de llevar a



(a) Diagrama de flujo global

(b) Diagrama de flujo de *SceneLoader***Figura 4.21** – Diagramas de flujo del controlador de escenas

cabo la carga de la escena o cualquier otra acción relevante.

En el [Código 4.4](#) y [Código 4.5](#), se presentan dos fragmentos de código que ilustran la carga de escenas, tanto con como sin pantalla de carga. Estos ejemplos muestran cómo se aplican las funciones `async` para lograr una gestión eficiente de las transiciones entre escenas.

Código 4.4 – Carga de escenas inmediata

```
1 public async void LoadScene(string sceneName)
2 {
3     await StartTransition.Invoke();
4     SceneManager.LoadScene(sceneName);
5 }
```

Código 4.5 – Carga de escenas con pantalla de carga

```
1 public async void LoadWithLoadingScreen(string nextScene, string loadingScene)
2 {
3     await StartTransition.Invoke();
4     nextSceneIndex = nextScene;
5     this.loadingScene = loadingScene;
6     SceneManager.LoadScene(loadingScene);
7 }
```

4.7.4. Controlador del sistema

Para lograr la integración del controlador de escenas (*SceneController*) con la gestión de escenas mediante *SceneLogic* y *SceneLoader*, se utilizará un [diccionario](#) que relacione el nombre de las escenas con su respectiva lógica (*SceneLogic*). Esto evitará la necesidad de trabajar con índices para identificar y cambiar entre escenas.

El controlador de escenas ofrece dos funciones principales: *NextScene* y *PreviousScene*, las cuales se encargan de gestionar los cambios de escena y serán las únicas funciones accesibles por *GameManager*. Cuando se invoca la función *NextScene* o *PreviousScene*, el controlador obtiene el nombre de la escena actual mediante *SceneManager.GetActiveScene()*. Luego, utilizando el diccionario, se accede a la lógica correspondiente a esa escena, donde se define qué escena debe seguir o preceder a la actual y si la transición debe ser asíncrona o no. Una vez determinada la escena siguiente o previa, el controlador emplea el *SceneLoader* para cargar la nueva escena. Cualquier componente que se deba inicializar al comienzo de la nueva escena viene definido también en *SceneLogic*.

Gracias a esta estructura, el controlador de escenas puede manejar de manera más eficiente y organizada las transiciones entre escenas. Además, proporciona flexibilidad para agregar nuevas escenas y lógicas al proyecto sin afectar la funcionalidad existente.

4.8. Controlador de cámara

El controlador de cámaras desempeñará la función de proporcionar la flexibilidad necesaria en nuestro juego para facilitar cambios en la cámara, su configuración y movimientos de manera sencilla. Además, es fundamental que este controlador sea altamente escalable y versátil,

dado que las cámaras pueden variar considerablemente entre distintos videojuegos. Por lo tanto, debemos proporcionar al programador las herramientas necesarias para que pueda desarrollar comportamientos específicos para su juego.

En nuestra implementación, hemos utilizado el complemento de Unity llamado *Cinemachine* [70], que se ha convertido en una herramienta imprescindible para los desarrolladores de este motor. Dicho complemento proporciona funcionalidades básicas de gestión y configuración de cámaras, las cuales no están incluidas por defecto en Unity, y cuyo desarrollo resultaría laborioso. Nuestro objetivo, por tanto, consiste en crear un controlador que nos permita gestionar y añadir variabilidad a las cámaras de *Cinemachine*, de modo que podamos modificar sus parámetros durante la ejecución del juego, y crear efectos de cámara que contribuyan a una mayor inmersión en la experiencia del videojuego.

En la [Figura 4.22](#) se muestra el diagrama de clases completo del controlador de cámara para poder seguir la explicación con una ayuda visual.

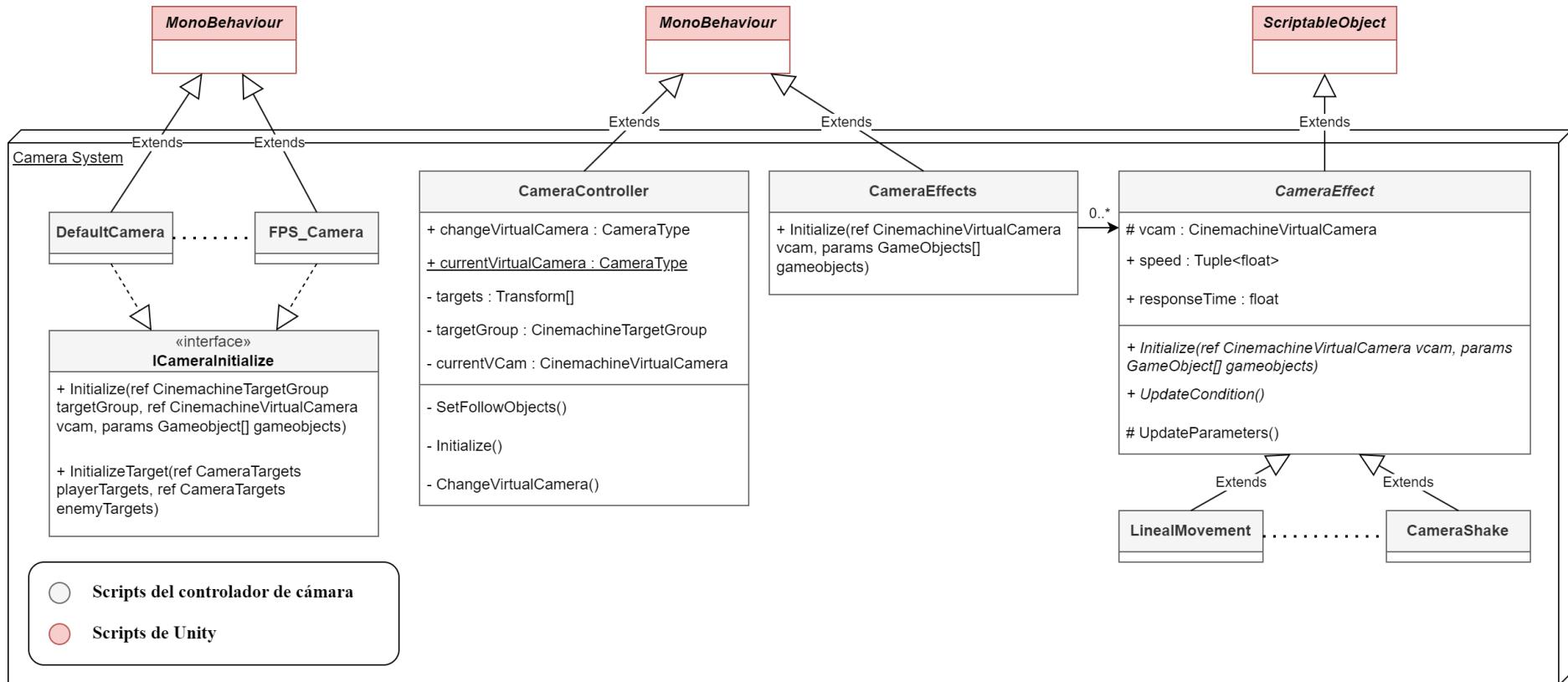


Figura 4.22 – Diagrama de clases del controlador de cámaras

4.8.1. Uso de MonoBehaviour y Prefabs

La estructura que hemos seguido hasta ahora para desarrollar los sistemas sufrirá un cambio ligero con la incorporación del controlador de cámaras. Como mencionamos al inicio de este capítulo, los sistemas de cámara e interfaz de usuario no pueden centralizarse en una única instancia estática (*GameManager*). Estos sistemas tienen una estructura presente en cada escena y, por lo tanto, es necesario que se instancien en cada una de ellas, dado que pueden variar significativamente entre diferentes escenas. Sería contraproducente tratar de contemplar toda esta variabilidad en una única instancia, en lugar de crear distintas instancias en cada escena.

Cabe destacar que cualquier *script* que se desee instanciar en algún objeto de la escena debe heredar de la clase *MonoBehaviour*, para que pueda ser serializado en el inspector del objeto y pueda estar presente en el ciclo del juego. Debido a que *Cinemachine* opera exclusivamente con *scripts MonoBehaviour*, resulta más conveniente adaptarnos y seguir su funcionamiento. De esta manera, utilizamos la jerarquía de objetos y de la escena para organizar la estructura de datos, y el inspector de objetos para cambiar la mayoría de parámetros.

Además del cambio mencionado en la estructura de desarrollo, me gustaría señalar una herramienta que resultará de gran utilidad para facilitar la creación de jerarquías de objetos que se instancian varias veces en nuestra escena o se utilizan en múltiples escenas: los *Prefabs* de Unity [6]. Estos *Prefabs* nos permiten almacenar un objeto *GameObject* completo con sus componentes y propiedades, así como sus descendientes. Si realizamos modificaciones en algún elemento de este objeto, se reflejarán automáticamente en todas las instancias que hayamos creado, debido a que en tiempo de compilación es el mismo objeto, pero en tiempo de ejecución, se generan instancias del mismo en cada lugar donde lo coloquemos.

Esta característica es especialmente valiosa en el contexto de nuestro controlador de cámaras, ya que nos permitirá crear un *Prefab* que contenga todas las configuraciones y parámetros deseados para una determinada cámara o conjunto de cámaras. Luego, podremos instanciar este *Prefab* en diferentes escenas o replicarlo cuantas veces sea necesario, manteniendo la flexibilidad y coherencia en la gestión de las cámaras a lo largo del juego.

4.8.2. Breve introducción a Cinemachine de Unity

Cinemachine proporciona numerosas utilidades para el manejo de cámaras que nos van a facilitar la gestión y configuración de las mismas de manera dinámica. Vamos a ver cuatro de los *scripts* que se deberán instanciar en la escena y serán indispensables prácticamente para todo videojuego.

- ***CinemachineVirtualCamera***. El componente *CinemachineVirtualCamera* es el corazón de *Cinemachine* y es esencial para definir cómo se comportará una cámara en particular en el juego. Esta utilidad permite crear cámaras virtuales que pueden seguir y enfocar automáticamente a objetos específicos en la escena. Al asignar un objetivo (*target*) a la cámara virtual, esta podrá seguir su posición y movimiento, lo que resulta especialmente útil para seguir a personajes, objetos o puntos de interés en el juego.

Además, *CinemachineVirtualCamera* ofrece una amplia gama de ajustes para controlar la composición, el campo de visión, el enfoque, el seguimiento suave y otros efectos cinematográficos.

- **CinemachineTargetGroup:** El componente *CinemachineTargetGroup* nos permite agrupar varios objetivos (*targets*) para que una cámara virtual pueda enfocarlos de manera conjunta. Esto es especialmente útil cuando deseamos mantener en el encuadre a múltiples personajes o elementos relevantes de la escena.
- **CinemachineBrain:** El componente *CinemachineBrain* actúa como el administrador central de las cámaras virtuales en la escena. Permite controlar cuál cámara virtual está activa en un momento dado y facilita la transición suave entre cámaras virtuales.
- **CinemachineFreeLook:** El componente *CinemachineFreeLook* permite al jugador cambiar la vista libremente, mediante el control de joysticks, ratón o teclado.

4

4.8.3. Componentes y funcionamiento

El controlador de cámaras que estamos implementando consta de tres componentes distintos, cada uno con un propósito específico para la gestión y variación de las cámaras virtuales definidas en la escena. Estos componentes están diseñados para ofrecer una estructura sólida y flexible que permita a los programadores crear y personalizar sus propios comportamientos de cámara de acuerdo con las necesidades de sus juegos.

En primer lugar, tenemos los componentes de definición de cámaras. Estos son *scripts* asociados a cada cámara virtual en la escena, y se encargan de establecer su comportamiento general. Aquí se define cómo la cámara sigue y enfoca a los objetivos, y si es necesario, se pueden agregar comportamientos específicos que van más allá de las capacidades proporcionadas por *Cinemachine*. El objetivo principal de estos *scripts* es mantener un comportamiento constante para la cámara, asegurando una experiencia coherente y bien definida para el jugador.

En segundo lugar, contamos con componentes para la variación de parámetros en tiempo de ejecución. Estos *scripts* permiten ajustar los parámetros de las cámaras virtuales durante la ejecución del juego, basándose en condiciones o eventos específicos que ocurran en la partida. Por ejemplo, pueden utilizarse para activar movimientos de cámara concretos cuando el jugador interactúa con ciertos elementos del entorno o enfrenta situaciones particulares.

Finalmente, tenemos el componente de gestión y cambio de cámaras. Este componente actúa como un administrador central que tiene conocimiento de todas las cámaras virtuales instanciadas en la escena. Su principal función es permitir cambios fluidos entre cámaras y establecer cuál cámara virtual está activa en cada momento del juego. De esta manera, se facilita la transición entre diferentes perspectivas sin interrupciones abruptas.

4.8.4. Definición de cámaras

Las cámaras definen su comportamiento individualmente, cada una en su propio *script*. Sin embargo, todas ellas deben implementar la interfaz *ICameraInitialize*. De esta forma, nos aseguramos de que el controlador de todas las cámaras virtuales (*CameraController*), que también será el encargado de inicializarlas, tenga acceso a los métodos necesarios para configurar adecuadamente las cámaras. En el diagrama de clases, se muestra que la interfaz cuenta con dos métodos de inicialización. El primero que inicializa la *CinemachineVirtualCamera* y el *targetGroup*. Dependiendo del juego que estemos desarrollando, también puede ser necesario proporcionar un

jugador o enemigo. El segundo método que se encarga de configurar los *targets* de la cámara. Este proceso es crucial, ya que aquí se definen los puntos a seguir o mirar por parte de la cámara.

Una vez tenemos en el *script* una referencia a la cámara virtual, al targetGroup o a cualquier otro componente de *Cinemachine*, podemos definir un comportamiento específico para cada cámara.

4.8.5. Efectos de cámara

Para la creación de efectos de cámara, hemos desarrollado la clase abstracta *CameraEffect*, que contiene los métodos necesarios para el funcionamiento de cualquier efecto visual en la cámara. Esta clase hereda de *ScriptableObject*, lo cual nos permite crear varias instancias de los efectos en la estructura de directorios con distintos parámetros. Esta flexibilidad nos resulta útil para realizar pruebas con diferentes configuraciones o para utilizar los efectos en varias cámaras del juego. Toda clase de efecto de cámara debe heredar de *CameraEffect*. Cada efecto se implementa de manera independiente, lo que significa que funcionará por sí solo, sin depender de la cámara virtual que estemos utilizando. La lógica y comportamiento de cada efecto están definidos en su propio *script*, y *CameraEffect* sirve como una plantilla que facilita la integración con *Cinemachine* y otros objetos del juego.

El funcionamiento de cualquier efecto de cámara está determinado por su proceso de inicialización, el cual puede requerir un número variable de objetos, además de la cámara virtual. Para lograr esto, hemos utilizado la palabra clave “params” de C# [71], lo que nos permite tener un número indefinido de parámetros del mismo tipo en la inicialización del efecto. Al llamar a la inicialización del efecto, proporcionamos los objetos relevantes que este necesite para llevar a cabo su función. Por ejemplo, un efecto de cámara puede depender de un enemigo específico, mientras que otro efecto puede requerir un objeto ambiental que lo desencadene. Al pasar los *GameObjects* como parámetros, siempre podemos obtener cualquier componente o utilidad asociada a ese objeto mediante la función *GetComponent*[72].

Cada efecto tiene parámetros comunes, como velocidad y tiempo de reacción, que suelen ser los más usuales. Además, puede definir otros parámetros que considere convenientes para su funcionamiento específico. Una vez que tengamos la inicialización de los objetos y parámetros necesarios, debemos definir la condición o desencadenante del efecto. Para ello, utilizamos la función *UpdateCondition*, la cual es abstracta y, por lo tanto, es de obligatoria definición en cualquier efecto. Hay dos formas en las que podemos utilizar esta función:

1. Comprobación constante de la condición - accedemos a los objetos que hemos inicializado y verificamos si se cumple alguna condición para activar o desactivar el efecto. Con este enfoque, debemos realizar la comprobación constantemente en el ciclo del juego. Sin embargo, a cambio, obtenemos efectos cancelables o modificables por alguna otra condición. Es decir, podemos detener o ajustar el efecto en función de ciertas situaciones.
2. Efectos reactivos mediante eventos - accedemos a los objetos inicializados y suscribimos funciones del efecto a eventos específicos que hemos definido. De esta manera, los efectos funcionarán de manera reactiva, activándose automáticamente ante la invocación de los eventos correspondientes. Sin embargo, en este caso, debemos tener algún mecanismo, como un temporizador o contador, para determinar el tiempo que durará el efecto. Esto es necesario ya que un evento solo dispara la función asociada, pero no la detiene automá-

ticamente. Si queremos que el efecto tenga una duración limitada, deberemos tener otro evento o mecanismo que lo detenga.

Ambas opciones son válidas y la elección depende de cómo estén funcionando las demás clases en nuestro juego. Además, es importante tener en cuenta que, con el segundo método de efectos reactivos mediante eventos, solo necesitamos llamar a la función *UpdateCondition* una única vez para las suscripciones a eventos. En cambio, con la primera opción de comprobación constante, debemos llamarla en cada frame del juego, lo que puede afectar al rendimiento si hay muchos efectos activos al mismo tiempo.

Por último, necesitamos una clase que se encargue de la inicialización de todos los efectos que hemos definido, asociándolos con las cámaras virtuales y los objetos que los desencadenan. Esta clase se llamará *CameraEffects*, y será una clase que herede de *MonoBehaviour*. La instancia de esta clase se debe colocar en el mismo objeto de la escena que nuestra cámara virtual y nuestro script de cámara. A través de esta clase, pasaremos todos los efectos que deseamos que funcionen, y nuestro controlador se encargará de inicializarlos cuando cambiemos de cámara.

4.8.6. Controlador

El controlador de cámaras (*CameraController*) será el núcleo encargado de inicializar todos los *scripts* de cámaras y efectos, así como de facilitar el cambio entre cámaras virtuales. Para lograr esto, debe ser capaz de acceder a todos los componentes de cámara presentes en la escena, incluyendo las *CinemachineVirtualCameras* junto con sus *scripts* de cámara y efectos, así como al *targetGroup* o *targetGroups* definidos y a los *targets* de los objetos externos a la cámara, como el jugador o los enemigos.

Debido a la estructura jerárquica con la que funcionan los objetos en la escena, podemos simplemente definir el controlador como el padre del resto de cámaras y elementos relacionados. Luego, utilizando la función *GetComponentsInChildren* [73], podemos obtener un vector de todos los elementos que son hijos del controlador y que contienen un componente concreto. Para acceder a los *targets* a los que la cámara debe apuntar, utilizamos, como en otras ocasiones, el sistema de etiquetas de Unity. Es decir, al inicio de la carga de la escena, buscamos los objetos que necesitamos identificados por una etiqueta y almacenamos sus referencias en variables. Esto lo logramos con la función *FindGameObjectsWithTag* [49].

A continuación, para que todos los comportamientos definidos en la cámara y sus efectos puedan funcionar correctamente, debemos inicializarlos de manera adecuada. Para lograr esto, aplicamos el principio de Inyección de Dependencias [74], que nos indica que en lugar de ser la propia clase la que crea los objetos, estos se le suministran para que pueda funcionar adecuadamente. De esta forma, las clases “inferiores” tienen referencias a los objetos de la cámara, proporcionadas a través de un único *script* centralizado, evitando así la necesidad de realizar más búsquedas de datos y en general, operaciones más costosas de obtención de componentes [61].

Para finalizar, la realización del cambio de cámara es muy sencilla debido al funcionamiento interno de *Cinemachine*. Para nuestro propósito, necesitamos tener definido un *CinemachineBrain*, que automáticamente gestiona todas las cámaras definidas en la escena, independientemente del lugar que ocupen en su jerarquía. La cámara virtual activa es la única cuyo objeto está habilitado. En resumen, para cambiar de cámara, simplemente deshabilitamos las otras cámaras virtuales y dejamos habilitada la que deseamos utilizar.

Dado que desde el controlador tenemos acceso a todas las cámaras virtuales, hemos decidido crear un enumerador llamado *CameraType*, que define el nombre de cada cámara creada. Esto nos permite cambiar de cámara de forma más visual y no solo mediante un índice del vector de cámaras.

La función de cambio de cámaras solo necesita deshabilitar todas las cámaras virtuales, habilitar la cámara definida en el actual *CameraType* e inicializar los *scripts* asociados a dicha cámara con los objetos que ya hemos preparado desde el inicio de la ejecución en nuestro controlador.

4.9. Sistema de interfaz de usuario

4

La interfaz de usuario es un componente imprescindible para que el jugador pueda navegar entre menús, cambiar la configuración del juego, guardar o pausar la partida, e incluso realizar acciones propias del juego, como cambiar de arma o seleccionar una vestimenta diferente, entre otras funciones.

Al igual que ocurre con el sistema de cámaras, el diseño de la interfaz de usuario puede variar significativamente dependiendo del tipo de juego que se esté desarrollando. Por ejemplo, un videojuego arcade prácticamente no requerirá una interfaz de usuario compleja, ya que está diseñado para partidas rápidas que no requieren configuraciones ni guardado. En cambio, en un videojuego de rol, gran parte del tiempo del jugador podría dedicarse a interactuar con la interfaz de usuario para gestionar inventarios, realizar acciones específicas o tomar decisiones importantes. Por ello, es fundamental que nuestro sistema de interfaz de usuario sea altamente flexible y permita al diseñador realizar su trabajo con el menor número de obstáculos posible. Con este objetivo en mente, hemos desarrollado un *framework* para la navegabilidad de menús, que facilita su implementación, y hemos creado algunos *scripts* que, en conjunto con el Sistema de *UI* integrado de Unity, nos permiten crear menús de manera rápida y escalable.

En la [Figura 4.23](#) se muestra el diagrama de clases completo del sistema de interfaz de usuario para poder seguir la explicación con una ayuda visual.

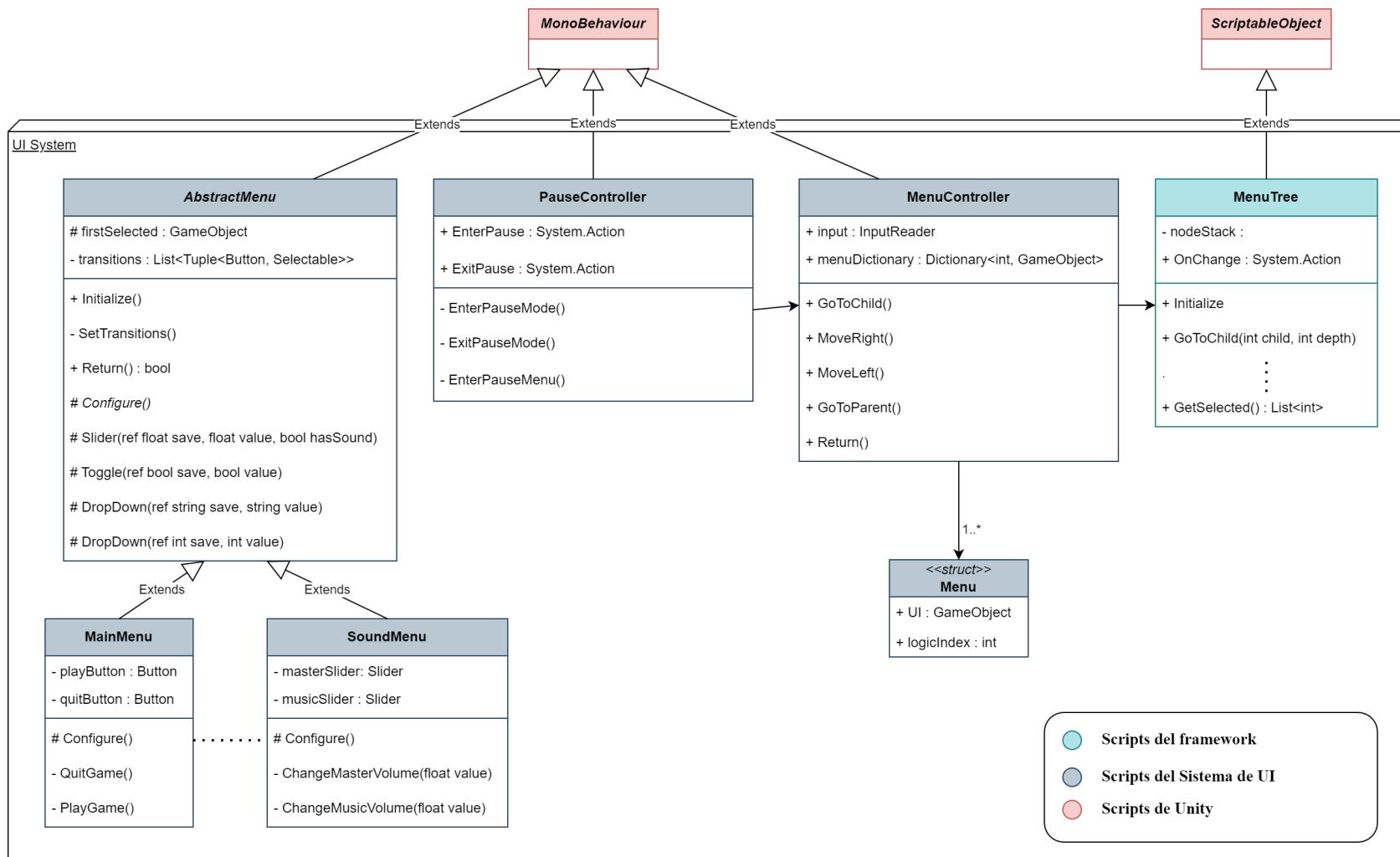


Figura 4.23 – Diagrama de clases del sistema de interfaz de usuario

4.9.1. Uso de MonoBehaviour y Prefabs

Siguiendo la misma lógica que en la gestión de cámaras, en el sistema de interfaz de usuario todos los componentes y objetos se instancian a nivel de escena. Por esta razón, resulta conveniente que los controladores y sus parámetros estén disponibles en el inspector de los objetos de la escena, y para lograr esto, nuestros *scripts* heredan de *MonoBehaviour*.

Una vez más, el uso de *Prefabs* de Unity [6] juega un papel clave en nuestro desarrollo. Nos permite almacenar menús completos y autocontenidos en forma de *Prefabs*, que podemos instanciar en todas las escenas que necesitemos. Esto nos brinda la ventaja de mantener una estructura organizada de directorios para todos los menús de nuestro juego, facilitando así su reutilización y mantenimiento.

4.9.2. Acercamiento a Unity UI

La comprensión del alcance de las funcionalidades que ofrece Unity UI [75] en cualquier proyecto resulta compleja, ya que se equipara a la profundidad de programas o bibliotecas dedicados a la creación de interfaces de usuario como Android Studio, tkinter de Python o QT. De manera similar a QT o Android Studio, podemos diseñar nuestras interfaces gráficas (UI) en Unity UI mediante un sistema visual basado en la interacción de arrastrar y hacer clic, eliminando así la necesidad de involucrarse en la codificación subyacente. Esta herramienta comprende una amplia gama de elementos característicos en cualquier software de desarrollo de UI, tales como *canvas*, *layouts*, botones, *sliders*, cuadros de texto, desplegables, entre otros. Además, cada uno de estos objetos ofrece una configuración accesible a través del inspector del objeto en la escena. Los objetos que son interactivos proporcionan eventos a nivel de código a los cuales uno puede suscribirse, lo que permite responder automáticamente cuando una acción es desencadenada.

En lo que respecta a la navegación y la interacción con los elementos de los diversos menús, Unity ofrece opciones en cada elemento seleccionable para definir su capacidad de navegación con otros elementos [76]. Además, podemos acceder al estado en el que se encuentra un objeto seleccionable, que puede ser normal, *highlighted*, *pressed* o *disabled*. Mediante la utilización de la clase *EventSystem* [77], podemos obtener en el código información sobre el objeto que se encuentra actualmente seleccionado o incluso seleccionar un objeto mediante programación. Esta funcionalidad es muy valiosa para resaltar un objeto al inicializar un menú o para conservar la información sobre el último objeto seleccionado.

Unity UI proporciona todas las herramientas esenciales requeridas para desarrollar una interfaz de usuario. Estas herramientas pueden ser manejadas por el propio diseñador si está familiarizado con ellas. Por lo tanto, en esta sección no nos centramos en la creación de los propios menús o interfaces, sino en cómo los objetos interactivos se relacionan con el resto del código, así como la interconexión de los diferentes menús para ofrecer una navegabilidad fluida al usuario.

4.9.3. Estructura y funcionamiento

El sistema de interfaz de usuario se fundamenta en dos componentes que siguen el patrón Modelo Vista Controlador (MVC) [78] de manera autónoma.

1. MVC para la interconexión de objetos interactivos: En este contexto, buscamos aprovechar

los eventos proporcionados por Unity UI en los objetos interactivos para vincularlos con comportamientos en el código externo o dentro del propio sistema.

Ejemplo: Cuando ajustamos el volumen mediante un slider, nuestro objetivo es que este ajuste se refleje efectivamente en el volumen del juego. Para lograrlo, accedemos al *AudioMixer* (consultar la sección Sistema de Audio) para disminuir el volumen y al *DataSaver* (ver la sección Sistema de Guardado) para conservar los datos, todo ello gestionado a través del *GameManager*.

2. MVC para la navegabilidad del sistema: En esta instancia, empleamos un controlador para enlazar los menús diseñados en la vista con la lógica subyacente en el *framework* de navegabilidad.

Esta separación de funciones se adoptó con el fin de ejercer un mayor control sobre ambas áreas. Por un lado, los menús son tratados como entidades independientes capaces de navegar entre sí. Por otro lado, los objetos individuales presentes en cada menú y sus conexiones con el resto del código son manejados de manera específica. Esta división resulta beneficiosa, ya que asignar funcionalidades a los eventos de objetos interactivos requiere que los *scripts* estén ubicados en la escena misma. En contraste, definir la navegabilidad no precisa esta ubicación, como veremos posteriormente. Por tanto, ambas funciones se trabajan por separado debido a las diferencias en sus requisitos de implementación y alcance.

No obstante, los controladores de ambos componentes tienen la capacidad de interactuar entre sí para proporcionar retroalimentación, sin que esto afecte a cada vista o modelo individual. Por ejemplo, al presionar un botón para efectuar un cambio de menú, el controlador que administra la funcionalidad del botón deberá comunicarse con el controlador de navegabilidad para llevar a cabo la transición de menú. En un escenario inverso, podría ser necesario que, al realizar un cambio de menú, el controlador de navegabilidad se conecte con el controlador específico del menú con el fin de inicializar sus componentes.

Además de estos componentes, es necesario establecer la conexión entre el sistema de inputs que hemos desarrollado y el sistema de interfaz de usuario. Esta vinculación permite que el jugador pueda navegar libremente por los menús, utilizando tanto un controlador como teclado y ratón.

4.9.4. MVC para la interconexión de objetos interactivos

En este componente, la aplicación del patrón Modelo Vista Controlador resulta fundamental para lograr la independencia de todos los objetos interactivos en cuanto a su funcionamiento. En caso de seguir la aproximación propuesta por Unity, estaríamos obligados a navegar por el menú hasta ubicar dichos objetos y suscribir funciones públicas a ellos directamente en el inspector. Este enfoque generaría dependencias directas entre los objetos y los *scripts* que contienen tales funciones. En contraste, con nuestra metodología, las funcionalidades se suscriben en el controlador, lo que simplifica el proceso de gestión y alteración de estas funcionalidades ([Figura 4.24](#)).

Con el objetivo de vincular todos los objetos interactivos con distintas partes de nuestro código y, al mismo tiempo, brindar máxima flexibilidad, hemos optado por asignar a cada menú su propio controlador, instanciado en la escena. Siguiendo una lógica similar a la empleada para la cámara, hemos creado una clase abstracta que sirve como plantilla para la creación de

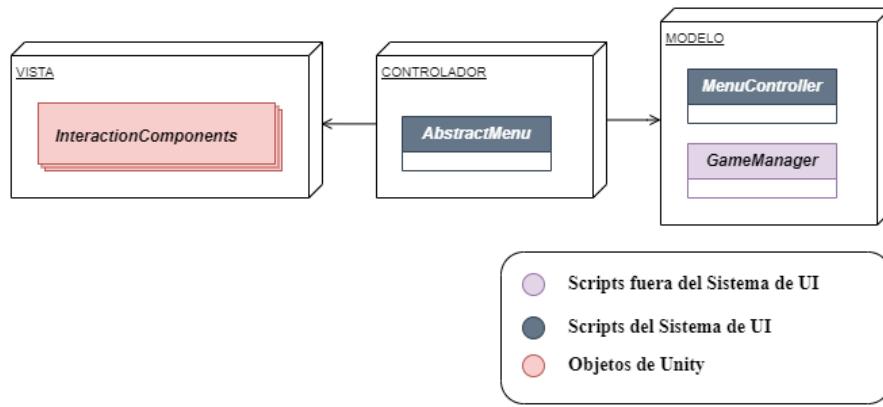


Figura 4.24 – Esquema de MVC para la interconexión de objetos interactivos

controladores de objetos personalizados, que enlacen su funcionamiento de manera acorde a las necesidades. Esta clase se denomina *AbstractMenu* y está compuesta principalmente por:

- Una referencia a un *GameObject* que actúe como el elemento inicialmente seleccionado en el menú.
- Método abstracto (*Configure*) que cada menú debe implementar, utilizado para la inicialización de todos los elementos del menú con datos de guardado. Además, este método debe suscribir funciones a los eventos de cada elemento, estableciendo así la conexión entre la vista y el controlador.
- Lista de transiciones representadas mediante tuplas de botones y elementos seleccionables.

Transición: Se refiere a cuando una función activada por un botón provoca la selección de otro elemento, o viceversa, cuando desde un elemento seleccionado se retorna a un botón.

Ejemplo: Un botón denominado "volumen" que al ser pulsado selecciona automáticamente un *slider*, permitiendo así ajustar el volumen.

- Método para configurar las transiciones (*SetTransitions*), es decir, establecer como función del evento de un botón, la selección de otro elemento.
- Varios métodos destinados a simplificar el almacenamiento de valores de elementos específicos (*Toggle*, *DropDown*, *Slider*, etc.) en el Sistema de Guardado.

En cada clase que herede de *AbstractMenu*, será necesario definir referencias a los objetos que deseamos vincular con determinadas funcionalidades, así como métodos para implementar las propias funcionalidades, en caso de que los *scripts* que representan el modelo no posean métodos adecuados. Mediante este componente, hemos logrado la capacidad de establecer funcionalidades altamente específicas al definir un *script* para cada menú individual. Además, para acceder a funciones de inicialización generales, podemos recurrir a una clase abstracta, independientemente del menú.

4.9.5. Acercamiento al framework de navegación de menús

Consideraremos que es relevante entender cómo se utiliza el *framework* que hemos desarrollado para adquirir una comprensión sólida de su aporte al patrón MVC. Primeramente, es funda-

mental tener en cuenta que este *framework* nos capacita para construir una estructura en forma de árbol que establece la navegación entre menús. Dicha estructura es creada mediante una interfaz intuitiva que nos permite crear y conectar nodos, de manera similar a la interfaz utilizada por el sistema de animaciones de Unity [79]. A cada nodo se le asigna un ID siguiendo el orden de una búsqueda en profundidad y se pueden consultar estos IDs desde un inspector diseñado para cada nodo.

Una vez que hemos establecido la estructura, esta se define en un *ScriptableObject* que alberga las funciones necesarias para navegar a través de los nodos y transiciones. Se han definido comportamientos específicos en algunos nodos para simplificar la navegación, y todo el enfoque está diseñado para evitar acciones de navegación inviables. Por ejemplo, no se permitirá realizar una acción de navegación que intente ir al nodo padre cuando ya se encuentra en el nodo raíz, o intentar acceder a un nodo hijo desde un nodo hoja.

En la [Figura 4.25](#) se muestra un ejemplo de árbol de navegación de menús. Para saber más acerca del funcionamiento de este *framework* se puede consultar el Capítulo 5.

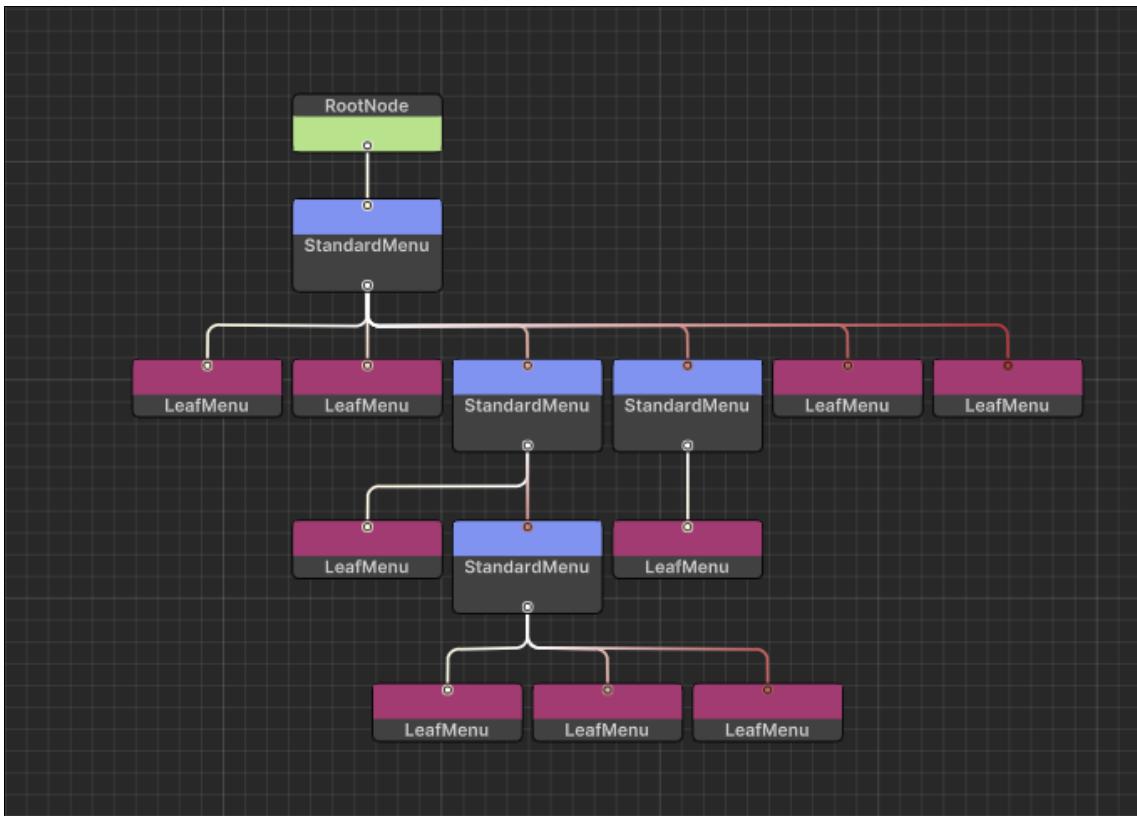


Figura 4.25 – Ejemplo de árbol de navegación de menús

4.9.6. MVC para la navegabilidad del sistema

Por lo general, la navegación entre diferentes menús puede ser un proceso complicado en Unity, ya que suele requerir que especifiquemos en cada botón o acción el menú específico al que deseamos dirigirnos. Esto se debe a que no existe una jerarquía predefinida ni un orden establecido para acceder a los menús. Aunque podríamos utilizar la jerarquía de objetos en la escena, esto nos limitaría a seguir ciertas pautas de diseño y a ajustar la disposición visual para satisfacer necesidades lógicas de navegación. Mediante el uso del *framework* que hemos desarrollado,

esta problemática se resuelve al proporcionar una estructura de árbol que establece el orden secuencial de los menús. Con funciones tan sencillas como *GoToChildren* o *GoToParent*, podemos navegar entre diversos.

En nuestro patrón MVC, el modelo abarca la lógica definida en el *framework*, la cual es completamente independiente de los elementos visuales presentes en la escena. Por otro lado, el controlador se encarna en el *script MenuController*, el cual establecerá la conexión entre la lógica y la presentación, y asumirá la administración y gestión del menú (Figura 4.26).

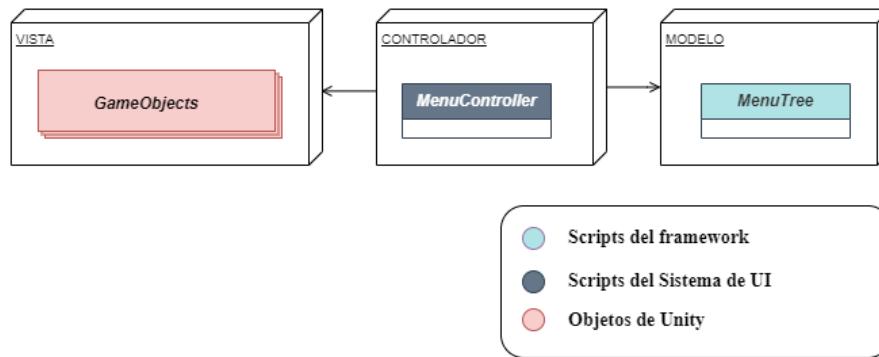


Figura 4.26 – Esquema de MVC para la navegabilidad del sistema

El controlador de menú se compone de una instancia del modelo, lo que nos permite navegar a través de la estructura de árbol y determinar en qué menú nos encontramos en la actualidad. También incluye una lista de elementos *Menu*, que sirve para vincular la representación visual con el modelo. En el inspector del controlador de menú, podemos modificar esta lista, asociando a cada *GameObject* que representa un menú su correspondiente índice lógico definido en el árbol de navegación. Dado que el índice se genera automáticamente en el árbol, solo necesitamos observar el inspector de cada nodo y asignarlo al menú que visualmente representa dicho nodo. La estructura *Menu* actúa como una herramienta para permitir estas asignaciones en el inspector. A nivel de código, cada asociación se agrega a un *diccionario* que proporciona una correspondencia directa entre un índice lógico y el *GameObject* que lo representa.

El proceso de asociación se lleva a cabo al inicio de la ejecución. De esta manera, podemos solicitar cualquier funcionalidad al *framework*, y este nos proporcionará un índice que siempre estará vinculado a un menú. Para visualizar este menú en la escena, simplemente debemos ocultar los demás menús y mostrar el menú actual.

4.9.7. Vínculo con el Sistema de Inputs

La integración entre la interfaz de usuario y el sistema de inputs es esencial, ya que permite al usuario controlar la interfaz con el dispositivo que esté utilizando para jugar. Unity proporciona un *script* llamado *UI Input Module*, que al ser instanciado en la escena, genera un *InputActionAsset* (Sección 4.4) con una configuración predeterminada para manejar tanto controles de mando como teclado/ratón en Unity UI. Esto abarca la navegación entre objetos interactivos, como mencionamos anteriormente, así como la interacción con el ratón, que se traduce en resaltar el objeto sobre el cual se encuentra el cursor. Además, se definen acciones de aceptación para el clic izquierdo y algún botón del mando.

Sin embargo, surgen desafíos cuando tratamos aspectos no contemplados por Unity, como la navegación entre menús o la incorporación de botones especiales específicos para nuestros

menús. Un ejemplo común es tener una tecla que permita al usuario regresar al menú anterior sin necesidad de utilizar un objeto interactivo. En estos casos, aprovechando la estructura previamente definida en nuestro *InputReader* (manejador de eventos), simplemente necesitamos establecer un nuevo evento para esta funcionalidad específica. Luego, asignamos este evento en nuestro *PlayerInput* al *action* que hayamos definido en el *action map* y nos suscribimos a este evento con la función encargada de realizar el cambio de menú ([Figura 4.8](#)). En nuestro caso, esta función sería *GoToParent*, ubicada en el *MenuController*.

Un aspecto adicional de relevancia es la representación de los inputs en el menú, lo cual es una necesidad común. En la primera sección de este capítulo, ya hemos discutido cómo se estaban empleando dos fuentes de letra con sus respectivos mapeados para lograr una representación uniforme de los inputs, independientemente del tipo de controlador que esté siendo utilizado. No obstante, el desafío que enfrentamos ahora es cómo lograr un cambio coordinado en todos los textos que involucran esta representación en caso de que cambiemos de dispositivo. Para abordar esta cuestión, se presentan dos opciones:

1. Mantener referencias a cada uno de los textos que requieren esta adaptación y modificarlos cuando nuestro *script InputDetection* detecte un cambio en el dispositivo.
2. Permitir que todos los textos que necesitan esta adaptación accedan a una variable global que defina constantemente la fuente que debe ser utilizada.

En nuestra situación, hemos decidido optar por la segunda opción. Para implementar esto, creamos un *script TextAutoSet* que debe instanciarse en cada objeto que desee cambiar automáticamente el texto a la fuente correspondiente. Esta tarea se logra de manera sencilla mediante el patrón Observador [46]. En nuestra clase *InputDetection*, hemos incorporado un evento que se activa cada vez que se detecta un cambio en el dispositivo. En esta misma clase, también mantenemos las variables que nos indican qué fuente debe ser utilizada. Al suscribir una función de cambio de fuente a este evento, podemos lograr el resultado deseado. Es importante destacar que múltiples instancias pueden suscribirse a este evento y todas serán notificadas y ejecutarán la función correspondiente.

4.10. Conclusiones

Concluyendo este capítulo, es importante reconocer que nuestra implementación no pretende rivalizar con la sofisticación de las empresas líderes en la industria del desarrollo de videojuegos. Sin embargo, nuestra intención es que este trabajo sea una herramienta útil y accesible para aquellos que se aventuran en el mundo de la creación de videojuegos. Cada uno de los componentes abordados, desde el sistema de audio hasta la interfaz de usuario o la gestión de escenas, ha sido diseñado y ejecutado con la intención de fomentar un flujo de trabajo más eficiente y una base sólida para el desarrollo de proyectos en Unity. Estamos comprometidos a seguir mejorando y refinando estos sistemas, con la esperanza de que puedan ser aprovechados y adaptados de acuerdo a las necesidades y metas individuales de cada desarrollador.

Por último mencionar que el rendimiento, la aplicabilidad práctica y una comprensión más profunda de las áreas que pueden ser mejoradas o expandidas se exploran en detalle en el [Capítulo 6](#).

Framework de creación de menús

5.1. Introducción

En este capítulo, profundizamos en la implementación del marco de navegación de menús que empleamos en la sección de interfaz de usuario descrita en el capítulo anterior. Nos enfocamos en detallar la estructura y el funcionamiento que definen el núcleo del entorno de desarrollo, dejando un enfoque más limitado en el [frontend](#), dado que esta última requiere un amplio conocimiento de la API de desarrollo propio de Unity. Nuestra intención consiste en proporcionar a los lectores perspicacia sobre las utilidades que se pueden desarrollar en Unity. En caso de que deseen tener un conocimiento más exhaustivo del apartado visual, siempre pueden recurrir a la documentación disponible [39].

5.2. Motivación

¿Cuántas veces os habéis encontrado en la encrucijada de implementar un controlador en primera persona, un gestor de menús, una interfaz de audio y otros retos similares? Son preguntas que lamentablemente cualquier desarrollador de videojuegos respondería con un número más alto de lo deseado. La cuestión aquí radica en la realidad que hemos ido descubriendo a lo largo de este proyecto: la asombrosa cantidad de recursos y sistemas que un videojuego típico demanda por defecto. Nuestra propuesta tiene como fundamento crear un código reutilizable e integrable en cualquier proyecto que se fragüe en Unity.

Lo que se presenta en el [Capítulo 4](#), por decirlo así, son entidades que funcionan como marcos de trabajo a una escala más reducida, en sintonía con la interfaz que Unity propone. A pesar de que Unity nos allana en gran medida el camino cuando su interfaz se ajusta, hemos llegado a la conclusión de que ciertos aspectos del desarrollo podrían fluir más ágil y cómodamente mediante nuestra propia interfaz o funcionalidad. Y aquí es donde entra en escena la creación del [framework](#) de navegación de menús.

Si realizamos una pequeña búsqueda en internet, nos damos cuenta de que no abundan los complementos que sean verdaderamente útiles en Unity; este campo es algo aún poco explorado y los que existen a menudo requieren un desembolso. Con este capítulo, nuestra intención es dar a conocer una funcionalidad en Unity que solo es utilizada por "los más avezados", pero que en realidad no dista mucho de cualquier otra aplicación con gestión de interfaces propia. Nuestro principal objetivo aquí es mostrar cómo podemos crear estas utilidades, cómo se despliega un abanico nuevo de posibilidades para los creadores y, al mismo tiempo, incentivar a las personas a confeccionar sus propios marcos de trabajo, lo que agilizaría el desarrollo y lo tornaría más fluido.

5.3. Idea y fundamentos del marco de trabajo

Antes de sumergirnos en la implementación, es esencial abordar la idea subyacente, la chispa que encendió el primer engranaje y nos impulsó a crear este entorno de trabajo.

5.3.1. Problema y Desafíos en la Navegabilidad de Menús

Imaginemos por un instante que estamos inmersos en Baldur's Gate 3 y deseamos revisar el atuendo de nuestro personaje. Sin embargo, al hacer clic en el botón de “inventario y equipamiento”, se despliega ante nosotros un mosaico de posibilidades (ver Figura 5.1).

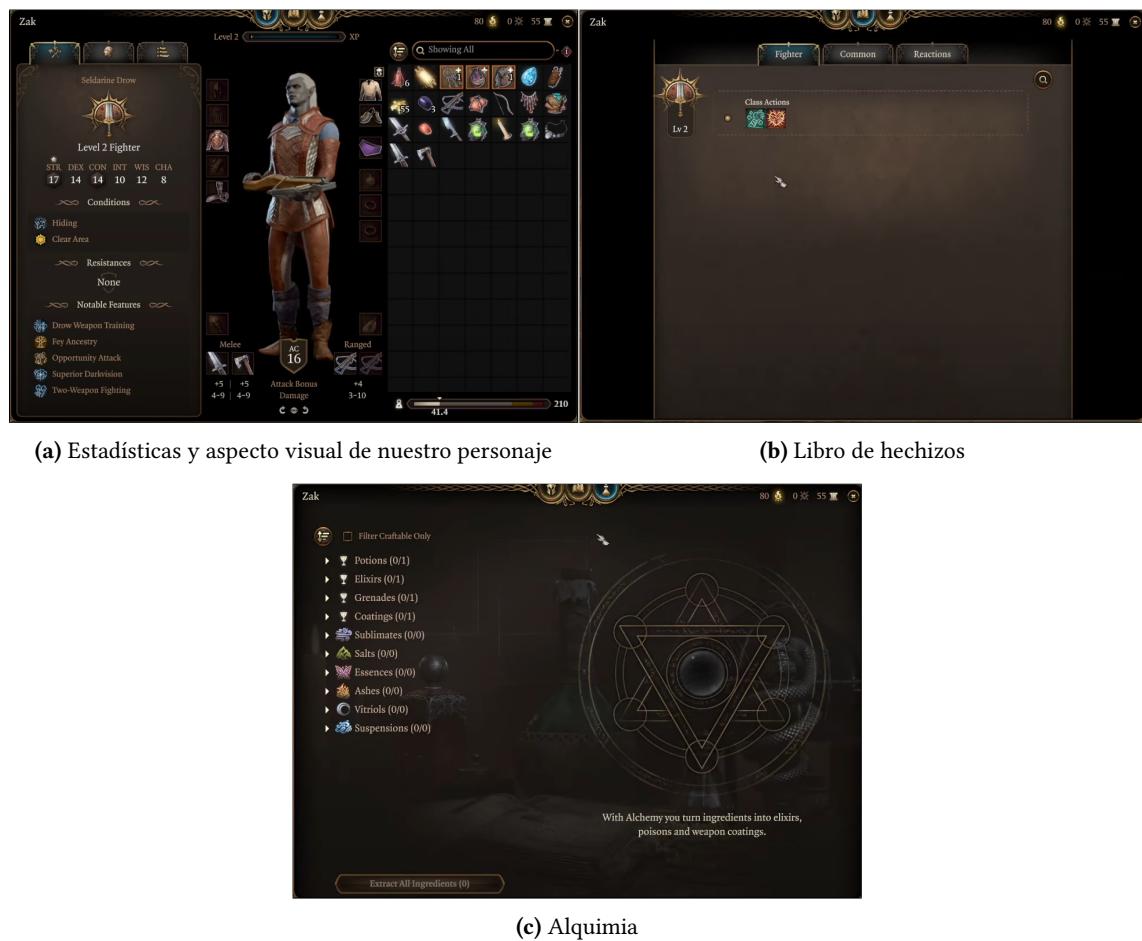


Figura 5.1 – Menú de inventario y equipamiento de Baldur's Gate 3 - Obtenido de [80]

Dado que quedamos prendados por esta disposición de menú en un juego de rol, decidimos emular un menú similar para nuestro videojuego en Unity. Tras un análisis minucioso y un desglose en sus elementos esenciales, notamos que está compuesto por un menú seleccionable en la parte superior central. Este menú nos permite alternar entre la sección de equipamiento básico, el compendio de hechizos y la esfera alquímica. En el espacio de equipamiento básico, encontramos un menú permanente, junto con otro menú seleccionable en el lateral izquierdo que facilita la alternancia entre diferentes visualizaciones de atributos. En la sección de hechizos, también encontramos un menú seleccionable que nos permite cambiar entre hechizos de combate, conjuros comunes o reacciones.

Además, concebimos la idea de independizar cada menú de manera individual, con la intención de brindar comodidad al diseñador y evitar la creación de una abrumadora estructura de objetos en la *escena*. Esto podría incluso traducirse en diversas *prefabs* para cada menú. Una vez que creamos todas estas estructuras para cada menú, surge la necesidad de establecer conexiones entre ellos. Para lograrlo, identificamos los botones en cada menú que permiten la navegación entre ellos. Les asignamos eventos *OnClick* con funcionalidades específicas: desactivar el menú actual e activar el nuevo menú que deseamos mostrar. Al completar esta tarea con todos los botones que activan nuevos menús, nos sorprende darnos cuenta de que, sin haberlo planeado, hemos creado una estructura lógica similar a la representada en [Figura 5.2](#).

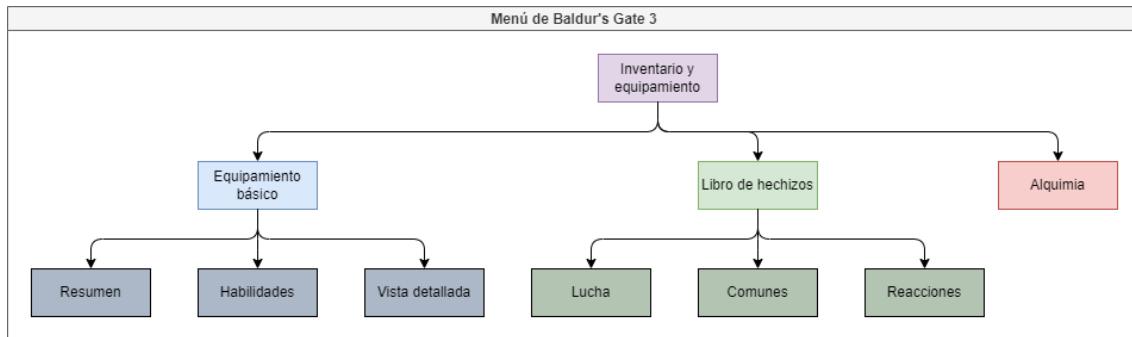


Figura 5.2 – Árbol lógico de navegabilidad para menú de inventario y equipamiento de Baldur's Gate 3

Es una estructura en apariencia sencilla y nos sorprende que, para llevarla a cabo, hayamos tenido que definir en cada menú elementos como el retroceso al menú anterior, la navegación entre menús hermanos y el mantenimiento de un menú desplegable mientras sus elementos cambian en cada ocasión. Surge la pregunta obvia: ¿por qué no centralizar toda esta lógica en un único editor de árboles lógicos?

5.3.2. Propuesta: Creación de un Editor de Árboles Lógicos

Proponemos la creación de un editor de árboles lógicos que integre las funcionalidades comunes de navegación de menús. Si bien los menús en los videojuegos pueden variar notablemente en términos visuales, la mayoría sigue una estructura de árbol subyacente. Este árbol define cómo el usuario navega entre las opciones. Visualmente, es como si el usuario estuviera siguiendo un sendero en este árbol interactivo. Al interactuar con el ratón o controladores, el usuario elige el camino que desea explorar. Tomando como ejemplo el menú de Baldur's Gate 3, nos enfrentamos a un dilema. Existen comportamientos recurrentes en los menús que nuestro árbol debe captar.

- Si un menú es seleccionable, normalmente debería permanecer visible junto con sus descendientes, permitiendo al usuario seleccionar cualquier opción del menú.
- Si el usuario emplea un mando, no necesitamos seleccionar un menú específico para navegar. En su lugar, recorremos de izquierda a derecha entre los submenús.
- En todo momento necesitamos conocer todo el camino del árbol para poder regresar al menú padre.

En consecuencia, para implementar este editor de árboles lógicos con funciones de menú, es imperativo mantener una comprensión constante de la ruta actual en el árbol y qué elementos

deben visualizarse. Para ilustrar, consideremos la ruta inicial y los elementos que deben ser visibles cuando se abre el menú de Baldur's Gate 3. En la [Figura 5.3](#), resaltamos en rojo el camino recorrido y en un trazo más grueso marcamos los menús que deben ser visibles en esa trayectoria.

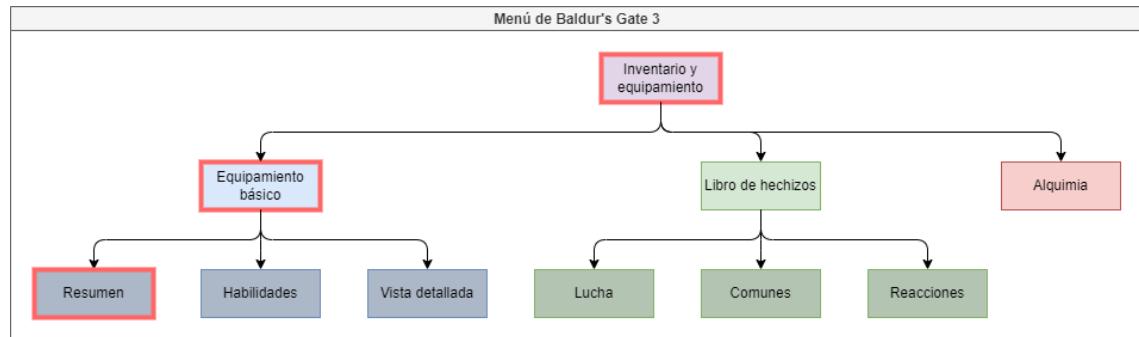


Figura 5.3 – Camino completamente visualizable del menú de Baldur's Gate 3

En este caso, se debe mantener visible todo el camino del árbol, ya que, excepto por las hojas, todos los menús son seleccionables y necesarios para el cambio entre ellos.

Ahora, imaginemos que añadimos un botón en el menú de alquimia que conduce a otro menú para la creación de pociones. En este escenario, el menú de alquimia ya no necesita permanecer visible mientras exploramos el menú de creación de pociones. La visualización y ruta cambiarían, como se observa en la [Figura 5.4](#).

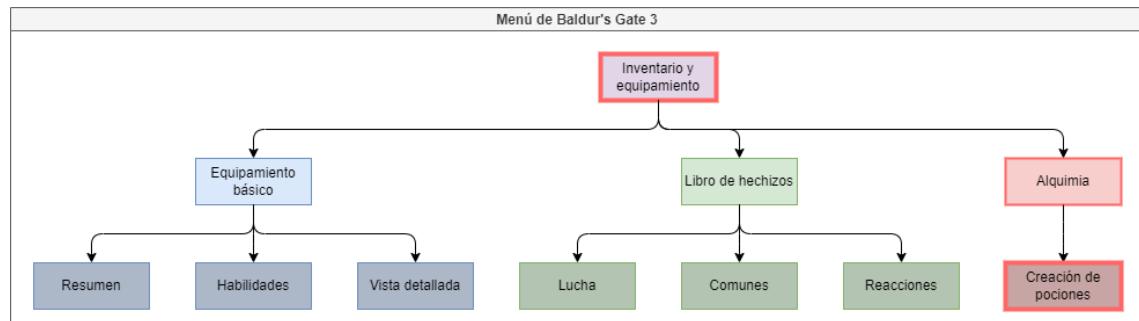


Figura 5.4 – Camino no completamente visualizable del menú de Baldur's Gate 3

En este caso, el menú “Inventario y Equipamiento” debe permanecer visible debido a su accesibilidad constante. Sin embargo, el menú de alquimia se ha desplazado hacia el menú de creación de pociones. Para lograr una visualización efectiva en todo momento, debemos considerar otros factores, como la profundidad del nivel y el tipo de menú (nodo) que estemos tratando.

5.3.2.1. Tipos de nodos

Para abarcar una amplia gama de menús en nuestra propuesta, es esencial establecer diferentes tipos de nodos, cada uno con sus propias características funcionales. Estos nodos seguirán siendo elementos del árbol al uso, pero se ajustarán mediante modificaciones en sus parámetros básicos o simplemente actuarán como identificadores para distinguirlos durante la navegación por el árbol.

1. **Raíz.** Este tipo de nodo no tiene permitido tener un parent, pero sí puede tener un único hijo. No está vinculado a ningún menú en particular, simplemente sirve como el nodo

inicial para construir la estructura. Cuando se inicia el árbol, el nodo hijo del nodo raíz siempre es visible.

2. **Compuesto**. Este tipo de nodo identifica aquellos que pueden tener más de un hijo y también pueden tener un parente. Además, este nodo puede ser estático, lo que significa que seguirá siendo visible cuando el camino avance hacia sus hijos.

- **Estándar**. Este nodo no tiene características especiales, es simplemente un nodo compuesto. Sin embargo, “Nodo Compuesto” es una abstracción y no se puede instanciar directamente.
- **Selectable**. Este nodo permite la navegación entre sus hijos de izquierda a derecha.

3. **Hoja**. Este tipo de nodo no puede tener hijos, pero sí puede tener un parente.

5.3.2.2. Funcionamiento

La premisa detrás de seguir esta estructura y crear el editor de árboles lógicos es que, en esencia, existe un árbol lógico subyacente que opera cuando realizamos cualquier acción en el menú que requiere consultar la navegabilidad. La propuesta consiste en que en nuestro controlador de menú solo necesitemos una instancia del árbol de menú. A partir de esta instancia, podremos ejecutar funciones básicas de navegación, tales como.

- **Navegar al hijo**. El camino se extiende hacia el hijo del nodo en el que nos encontramos actualmente. Si el nodo tiene varios hijos, se especificará al cuál queremos navegar. De lo contrario, se dirigirá al hijo por defecto, que es el primer hijo añadido al nodo.
- **Navegar al parente**. Dado que, por el momento, solo se permite tener un único parente, esta acción nos llevará al nodo “superior”, reduciendo el camino.
- **Navegar a la derecha**. Si el nodo parente actual es seleccionable, podemos desplazarnos hacia el hermano a la derecha del nodo actual.
- **Navegar a la izquierda**. Similar al caso anterior, si el nodo parente actual es seleccionable, podemos movernos hacia el hermano a la izquierda del nodo actual.

En la [Figura 5.5](#) se ilustran las acciones disponibles según el tipo de nodo que estemos manejando en el menú de ejemplo que hemos utilizado a lo largo de esta sección.

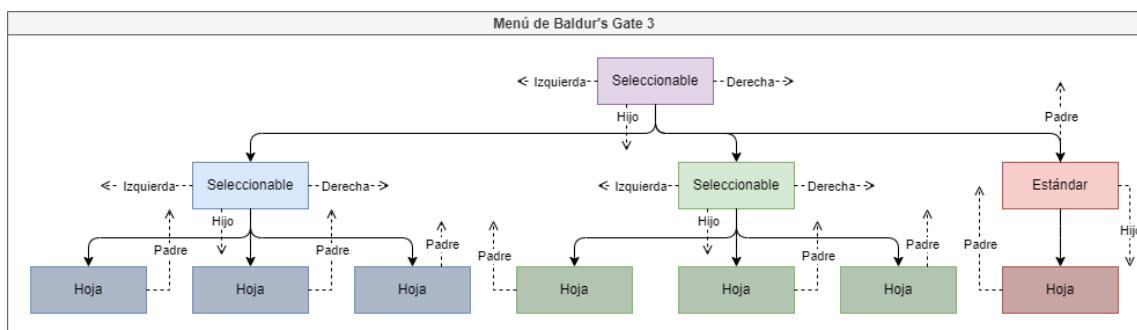


Figura 5.5 – Posibles acciones disponibles del menú de Baldur's Gate 3

5.3.2.3. Lógica subyacente

Para lograr un funcionamiento adecuado, las funciones de navegación deben tener en cuenta ciertos requisitos esenciales que aseguran la visualización correcta de menús.

- **Nodos estáticos.** Los nodos compuestos pueden ser marcados como estáticos, indicando que su existencia por sí sola carece de sentido. Por ejemplo, un menú seleccionable sin elementos seleccionados no tiene mucha utilidad. Los menús estándar estáticos también pueden utilizarse para transmitir información, ya que están divididos lógicamente, pero siempre acompañan a otros menús.
- **Navegar al hijo.** Si deseamos navegar a un nodo hijo, primero debemos verificar si el nodo en el que nos encontramos tiene permitido tener hijos (es compuesto). En segundo lugar, se debe comprobar si el nodo hijo es estático, ya que en caso de serlo, no puede existir por sí solo. En este caso, tendríamos que descender un nivel más hasta llegar al siguiente hijo y repetir el proceso.
- **Navegar al padre.** Si queremos navegar a un nodo padre, primero debemos comprobar que no sea el nodo raíz, ya que este no representa ningún menú y no tiene sentido navegar hasta él. En segundo lugar, debemos verificar si el nodo padre es estático, ya que si lo es, no puede existir por sí solo. En este caso, tendríamos que ascender un nivel más hasta llegar al siguiente padre y repetir el proceso.
- **Navegar a la derecha:** Esta acción se compone de los siguientes pasos:
 1. Comprobar si el padre es seleccionable.
 2. Navegar al padre.
 3. Identificar el hijo a la derecha del nodo actualmente seleccionado.
 4. Navegar al hijo a la derecha.
- **Navegar a la izquierda:** Similar al caso anterior.

Existen algunas consideraciones importantes para las acciones de navegación a la izquierda y a la derecha. Al navegar al padre para determinar el siguiente hijo a visualizar, debemos obviar si el padre es estático, lo que significa que ascenderemos al padre sin importar si es estático o no, lo que realmente importa es si es seleccionable. Otro aspecto crucial es especificar el nivel hasta el cual deseamos llevar a cabo la navegación, ya que podría haber múltiples menús seleccionables anidados. Por lo tanto, debemos indicar la profundidad deseada de padres a la que queremos llegar para efectuar el cambio de menú. Por ejemplo, en el caso de Baldur's Gate 3 ([Figura 5.2](#)), si nos encontramos en el menú “Resumen”, podemos realizar una navegación a la derecha en profundidad 1 en el menú “Equipamiento básico”, o una navegación a la derecha en profundidad 2 en el menú “Inventario y Equipamiento”.

Naturalmente, cuando no se cumplen alguno de los requisitos mencionados, la acción simplemente no se llevará a cabo porque no es el momento ni el lugar adecuado para ejecutarla. Por ejemplo, si nos encontramos en el menú inicial y presionamos la tecla en el controlador que nos permite retroceder al menú anterior, como el anterior es el nodo raíz, la acción no tendrá efecto alguno.

5.4. Estructura software

La estructura de software que hemos venido utilizando hasta este punto debe ser modificada para abordar esta tarea. Ahora no solo requerimos una comunicación entre el sistema en cuestión, el resto del juego y el diseñador, sino que también debe considerarse una nueva capa: el editor de Unity. Al referirnos al “editor”, hablamos de la aplicación propia de Unity, es decir, cómo, por ejemplo, se establece la comunicación entre el editor de animaciones (*animator*) [81] y las demás funcionalidades de nuestro proyecto.

Ya hemos observado que el controlador del menú de nuestro juego accede a la estructura del árbol de navegabilidad, utilizando dicha estructura como un modelo que proporciona la información necesaria para presentar la vista de manera adecuada. También es posible suscribir las funciones de los eventos de la vista a utilidades del controlador que acceden al modelo. De esta manera, logramos obtener una respuesta inmediata cuando se hace clic en algún botón de la interfaz de usuario del juego.

En cuanto a la comunicación entre el editor y la estructura del árbol, hemos optado por no aplicar un patrón típico de interfaces de usuario como patrón Modelo-vista-controlador (MVC), Modelo–Vista–Presentador (MVP) o modelo–vista–modelo de vista (MVVM) . Esto se debe a que, en este caso, la estructura no actúa tanto como un modelo del editor, sino más bien como una extensión. La estructura simplemente funciona como una interpretación de la información generada por el usuario en el editor del árbol. Además, dado el enfoque de creación de editores en Unity, esta propuesta resulta ser más efectiva.

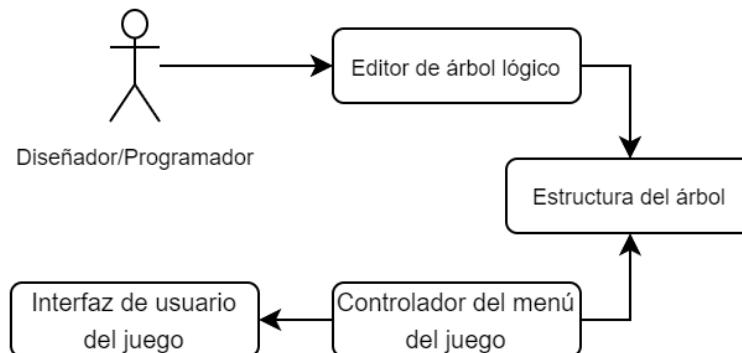


Figura 5.6 – Estructura y dependencias en el framework

5.5. Estructura del árbol

Vamos a comenzar por definir cómo hemos diseñado la estructura interna del árbol, que permite su navegabilidad, la creación de caminos y la gestión de los nodos.

5.5.1. Estructura abstracta

En primer lugar, exploraremos la estructura base del árbol, que nos proporciona una representación por defecto sin características particulares. Hemos diseñado esta estructura utilizando clases abstractas con el propósito de permitir a cualquiera implementar sus propias clases derivadas de estas, creando así nuevos nodos o árboles con funcionalidades específicas (Figura 5.7).

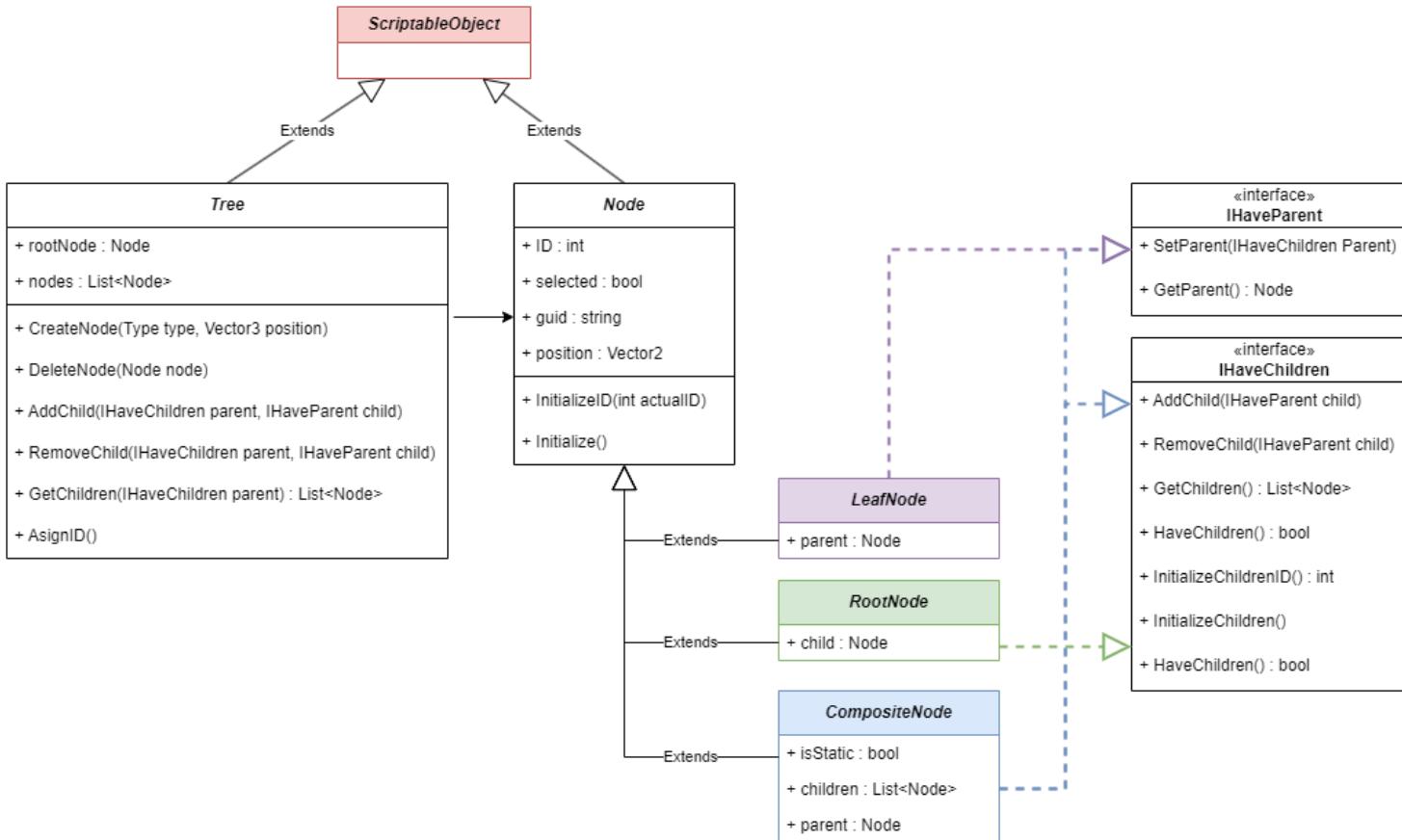


Figura 5.7 – Diagrama de clases del la estructura abstracta del árbol

Existe una clase abstracta *Tree* que, por defecto, puede crear y eliminar nodos del árbol, así como establecer enlaces entre los nodos (relaciones padre-hijo). Podemos heredar de esta clase para aplicar funcionalidades generales del árbol, como una navegabilidad especial o restricciones en la creación o eliminación de nodos según su tipo.

También contiene una clase por defecto llamada *Node*, de la cual podemos derivar, y que nos proporciona la información esencial que debe contener un nodo (ID, si está seleccionado, GUID o posición en el editor), además de funciones de inicialización para estos parámetros.

Sin embargo, lo más interesante aquí radica en la creación de nodos concretos, que heredan de *Node* e implementan interfaces. Incluimos dos interfaces de manera predeterminada: una para funciones comunes de nodos que contienen hijos, y otra para nodos que contienen padres. Si creamos nodos que no heredan de estas interfaces ni implementan sus funciones, no podrán tener padres ni hijos, lo cual se reflejará en el editor al no permitir enlaces entre ellos. El uso de interfaces en este contexto resulta especialmente útil, ya que nos permite determinar si un nodo, sin importar su tipo, puede o no tener hijos. Esto nos permite crear una gran variedad de nodos que pueden ser almacenados en variables como *IHaveParent* o *IHaveChildren* [64].

Además, es posible heredar de los nodos ya creados, como nodo hoja, compuesto o raíz, para crear tipos especiales de estos nodos. Cabe mencionar que el nodo raíz es un caso especial debido a su creación en el editor, aunque es posible modificarlo si es necesario.

Por último hay un par de funciones adicionales que pueden resultar interesantes con esta estructura:

- La estructura está diseñada para soportar el patrón Composite, lo que nos permite crear conjuntos de nodos simplemente creando una clase como *GroupOfNodes* y heredando de *Node*. De esta manera, si mantenemos una relación todo-uno de multiplicidad, podemos tener listas de grupos de nodos gracias a la recursividad que provoca el patrón Composite [82] (Figura 5.8).
- Tal y como está diseñada, esta estructura nos capacita para crear nodos que cumplan con los requisitos de los árboles de comportamiento. Estos árboles pueden ser empleados para definir, por ejemplo, la inteligencia artificial de los enemigos [83].

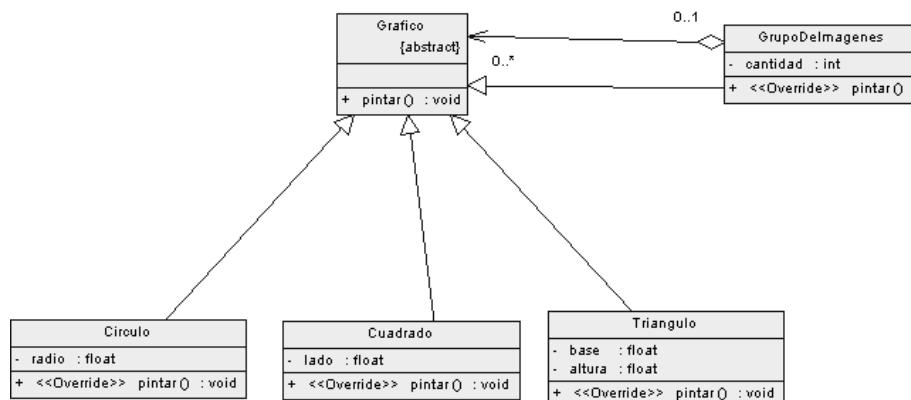


Figura 5.8 – Ejemplo de aplicación del patrón Composite - Adaptado de [82]

5.5.1.1. Uso de *ScriptableObject* y creación de asociaciones

La estructura del árbol se basa en un nodo raíz y una lista con el resto de nodos. Sin embargo, ¿cómo se generan las asociaciones padre-hijo?

La respuesta radica en la estructura interna de los nodos individuales. Cada nodo siempre tiene una referencia a sus relaciones directas. Si un nodo tiene hijos, posee una lista de referencias a esos hijos. Si tiene un parente, tiene una referencia a ese parente. Esta simplicidad nos garantiza una estructura de árbol en la que siempre podemos rastrear tanto los descendientes como los ancestros de cualquier nodo. Pero, ¿por qué la clase *Tree* también mantiene una lista con todos los nodos?

La clase *Tree* es la única que va a guardar las instancias de los nodos, dentro de los mismos solo tendremos referencias a nodos de esta lista, creando así asociaciones y asegurando un bajo coste de memoria.

Queda una última pregunta por responder sobre esta estructura abstracta: ¿por qué heredamos de *ScriptableObject* en las clases de árbol y nodos? Como se ha mencionado en otras ocasiones, esta clase es utilizable incluso cuando el juego no se está ejecutando. Esto es esencial, ya que esta estructura se va a editar fuera de la ejecución del juego, utilizando el editor que hemos diseñado. Además, la ventaja más significativa es que también podemos utilizar esta estructura mientras el juego está en ejecución, a diferencia de las utilidades del editor. En consecuencia, con una misma estructura, obtenemos un recurso donde crear y almacenar la configuración definida en nuestro editor, al mismo tiempo que obtenemos una funcionalidad que permite navegar por el árbol durante la ejecución del juego.

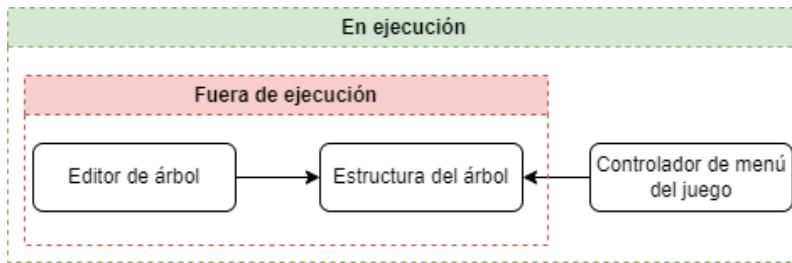


Figura 5.9 – Multifunción de *ScriptableObject* como estructura del árbol

5.5.1.2. Asignación de ID

La estructura por defecto también incluye una funcionalidad denominada *AsignID*, la cual recorre el árbol en una búsqueda en profundidad (Figura 5.10), comenzando por el nodo raíz, y asigna un ID a cada nodo en ese orden. Esta asignación de ID debe ejecutarse cuando el usuario desee actualizar la información del árbol, utilizando un botón que le permita realizar esta acción.

Este enfoque nos ofrece una asignación automática de identificadores que el usuario puede utilizar sin preocuparse por asignarlos manualmente. De esta manera, un nodo en este árbol puede representar cualquier entidad que el usuario desee en su juego. Solo necesita mapear (crear una asociación) entre ese número identificativo y cualquier otro objeto dentro de su juego. En nuestro caso, establecemos esta asociación con los objetos que definen los menús, pero las posibilidades son infinitas.

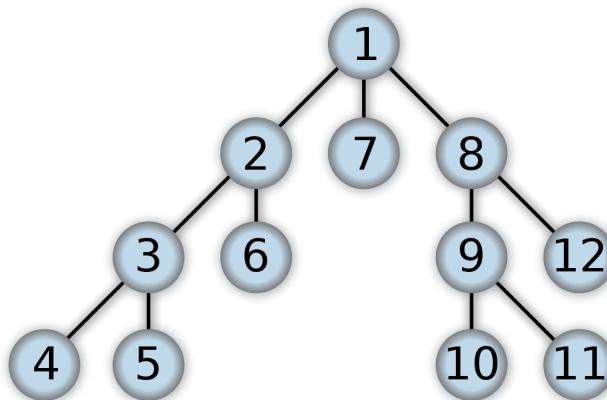


Figura 5.10 – Ejemplo de búsqueda en profundidad - Obtenido de [84]

5.5.2. Árbol de menús

Utilizando las clases abstractas mencionadas, podemos definir nuestras propias variantes como se muestra en la Figura 5.11. Para nuestro árbol de navegabilidad de menús hemos creado una clase nodo hoja, una clase estándar de nodo compuesto y otra herencia de nodo compuesto que representa el menú seleccionable. Esta última tiene la funcionalidad de proporcionar el hijo siguiente o anterior al que está actualmente seleccionado. Para lograrlo, hemos creado otra interfaz llamada *ISelectable*, que contiene las funciones que definen esta capacidad.

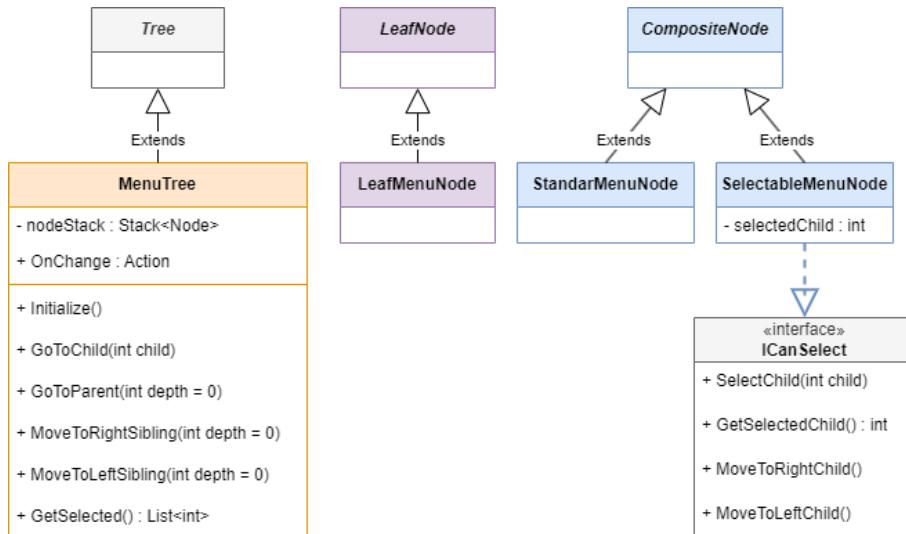


Figura 5.11 – Estructura del árbol de navegabilidad de menús

En lo que respecta a la comunicación con el editor, solo requerimos los nodos especializados para que el usuario pueda instanciarlos y crear la jerarquía del árbol. Sin embargo, para que el árbol adquiera funcionalidad durante la ejecución, debemos dotarlo de la capacidad de crear un camino dinámico y de determinar qué elementos deben ser visibles. Además, es esencial aplicar todas las restricciones que discutimos en la Subsección 5.3.2.3.

5.5.2.1. Creación de caminos

La responsable de esta tarea es la clase *MenuTree*, que hereda de *Tree*. Si recordamos las restricciones necesarias para que la navegabilidad funcione, debemos mantener un registro cons-

tante del camino que hemos seguido a lo largo del árbol y de los nodos que deben estar visibles en ese momento. Para lograr esto, la opción más simple pero efectiva es crear una pila que mantenga un registro de los nodos (menús) por los que estamos navegando. Además, cada nodo por defecto contiene una variable booleana llamada *selected*, que determina qué nodos deben estar visibles en todo momento. A partir de ahora, nos referiremos a los nodos que deben estar visibles como “seleccionados” debido al uso de esta variable.

Para llevar a cabo este proceso, es importante considerar lo siguiente:

1. El último nodo en el camino siempre será el tope de la pila y debe estar marcado como seleccionado.
2. Todas las funciones de navegabilidad operan siempre con el nodo que se encuentra en el tope de la pila.
3. Cumplir todas las restricciones mencionadas en la sección [Subsubsección 5.3.2.3](#) en cada una de las funciones de navegabilidad (Navegar al padre, Navegar al hijo, etc.), realizando las comprobaciones necesarias utilizando las interfaces proporcionadas, en lugar del tipo de nodo concreto.
4. Cambiar el valor del parámetro *selected* del nodo cuando sea necesario, teniendo en cuenta las restricciones mencionadas anteriormente.

Ejemplo: Si estamos en un nodo compuesto y deseamos navegar a uno de sus hijos, debemos verificar si el hijo es estático; en tal caso, debe permanecer seleccionado. De lo contrario, se deselecciona y solo permanece seleccionado su hijo.

5. La posibilidad de que los nodos compuestos sean estáticos presenta un desafío que superamos utilizando funciones recursivas.

Ejemplo: Si el hijo del ejemplo anterior fuera estático, no puede permanecer seleccionado por sí solo. Por lo tanto, debemos agregarlo a la pila y llamar nuevamente a la función de Navegar al hijo, creando así una recursión. El mismo enfoque se aplica al navegar al padre.

6. Si realizamos alguna navegación a la izquierda o a la derecha, primero debemos verificar si hay algún menú seleccionable en la pila, ya que no podríamos deshacer el camino si no existiera ninguno.

Para comprender mejor estos aspectos, se presentan los ejemplos ilustrados de la [Figura 5.12](#) y [Figura 5.13](#)

5.6. Editor de árbol de navegación de menús

Hasta este punto, hemos logrado establecer una estructura de árbol convencional que podría ser utilizada mediante el [inspector](#) por defecto de Unity. En esta sección, vamos a dar una descripción general de la ventana de edición que hemos desarrollado y de las funcionalidades que ofrece. También vamos a discutir cómo se integra esta ventana con los demás componentes de nuestra estructura de árbol.

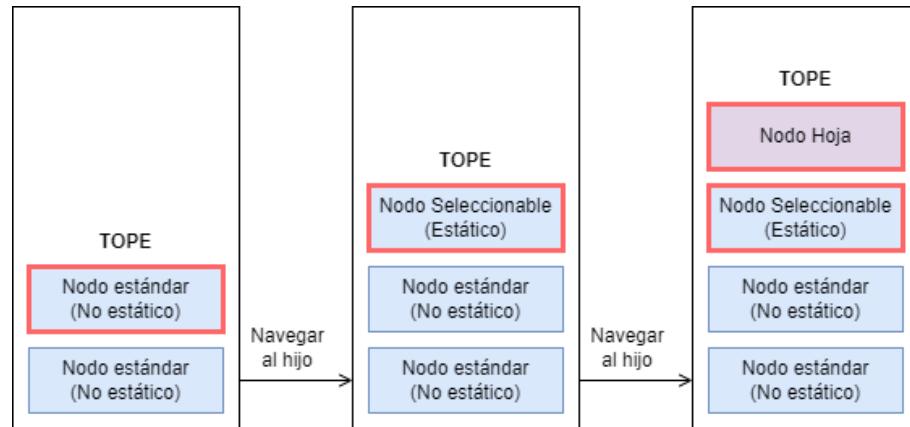


Figura 5.12 – Contenido de la pila en una navegabilidad al hijo con nodos estáticos

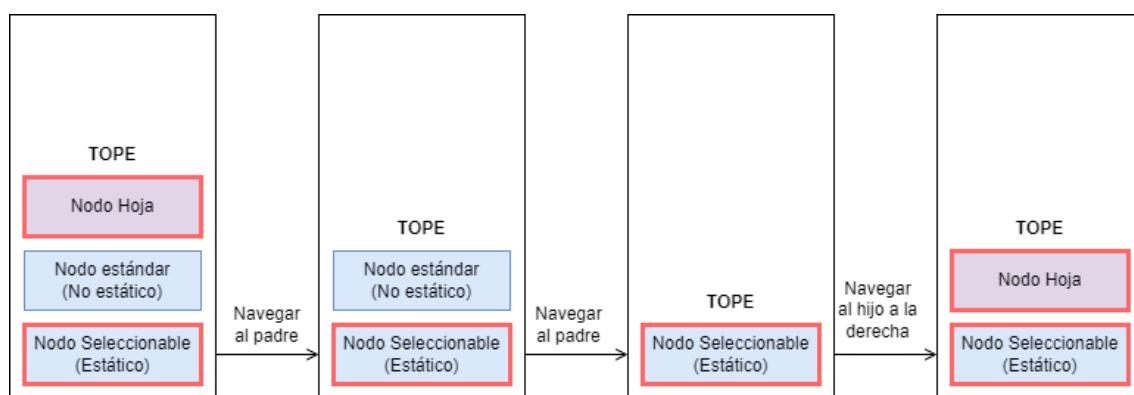


Figura 5.13 – Contenido de la pila en una navegabilidad a la derecha

Esta puede ser considerada como la parte más compleja de la creación de nuestro **framework**, ya que aprender a crear tus propios editores en Unity, especialmente uno basado en nodos como el que estamos desarrollando, demanda un buen nivel de conocimiento previo. No obstante, haremos nuestro mejor esfuerzo desglosando los aspectos importantes y proporcionando referencias a los manuales que utilicemos en el camino.

5.6.1. Breve introducción a la creación de editores en Unity

La construcción de nuestros propios editores en Unity se centra principalmente en dos características principales. La primera se refiere a la herramienta de Unity llamada *UI Builder* [85], la cual nos permite crear interfaces de usuario tanto para nuestro juego como para nuevas ventanas en el motor gráfico. Esta utilidad emplea archivos UXML [86] y USS [87], ambos versiones adaptadas para Unity de los formatos XML y CSS, respectivamente. Estos archivos nos posibilitan la creación de interfaces de manera similar a la que se realiza en entornos web u otros contextos similares.

La segunda característica son los *scripts* que proporciona Unity para la creación de editores, similares a los que vienen por defecto en su motor gráfico. Podemos heredar de muchos de estos *scripts* para crear nuestras propias versiones de ventanas de editor, gráficos, nodos y otras utilidades que el sistema brinda para el desarrollo de extensiones. En nuestro caso, vamos a utilizar las siguientes clases:

- **EditorWindow** [88]. Nos permitirá crear nuestra propia ventana que pueda abrirse desde el menú de “Ventanas” de Unity. En ella se deben incluir todos los componentes de nuestro editor.
- **Editor** [89]. Nos possibilita crear nuestro propio *inspector* de objetos, por ejemplo para visualizar o modificar los parámetros de los nodos.
- **TwoPaneSplitView** [90]. Contiene dos paneles de tamaño variable donde el borde entre los paneles se puede arrastrar para cambiar el tamaño de ambos paneles. Esto nos permitirá tener el gráfico de nodos y el *inspector* en una misma ventana con tamaños ajustables.
- **VisualElement** [91]. Es necesario para controlar elementos típicos de interfaz, como el *layout* o el estilo de ciertos elementos. En nuestro caso, lo utilizaremos para ajustar el editor de parámetros de nodos.
- **GraphView** [92]. Nos permite crear una interfaz para manipulación de nodos y conexiones. Esta clase tiene poca información en la documentación ya que todavía está en fase de prueba.
 - *Edge* [93]. Representa el enlace que une dos nodos. Se puede heredar de esta clase para crear enlaces con estilos personalizados.
 - *Node* [94]. Representa un nodo en nuestro gráfico. También podemos heredar de esta clase para definir nuestro propio estilo.

Es importante recordar que todo lo que definamos en UXML, USS o a través de los scripts debe ser código funcional que se carga en el editor, incluso si el juego no está en ejecución en ese momento. Para lograrlo, Unity proporciona nombres especiales de carpetas con propósitos específicos. En este caso, todos los *scripts* relacionados con el editor deben almacenarse en la carpeta llamada “Editor” [95].

5.6.2. Detalles de la implementación

Antes de pasar a toda la implementación de este editor queremos comentar un par de cuestiones a tener en cuenta. El uso de *UI Builder* con UXML y *StyleSheet* en USS no se va a tratar ya que esto supone principalmente el aspecto visual y como se distribuyen los elementos en el editor, creemos que no tiene caso en este trabajo. Si que vamos a tratar en profundidad la carga de estos elementos visuales, la creación de los elementos en la interfaz y su comunicación con nuestra estructura de árbol.

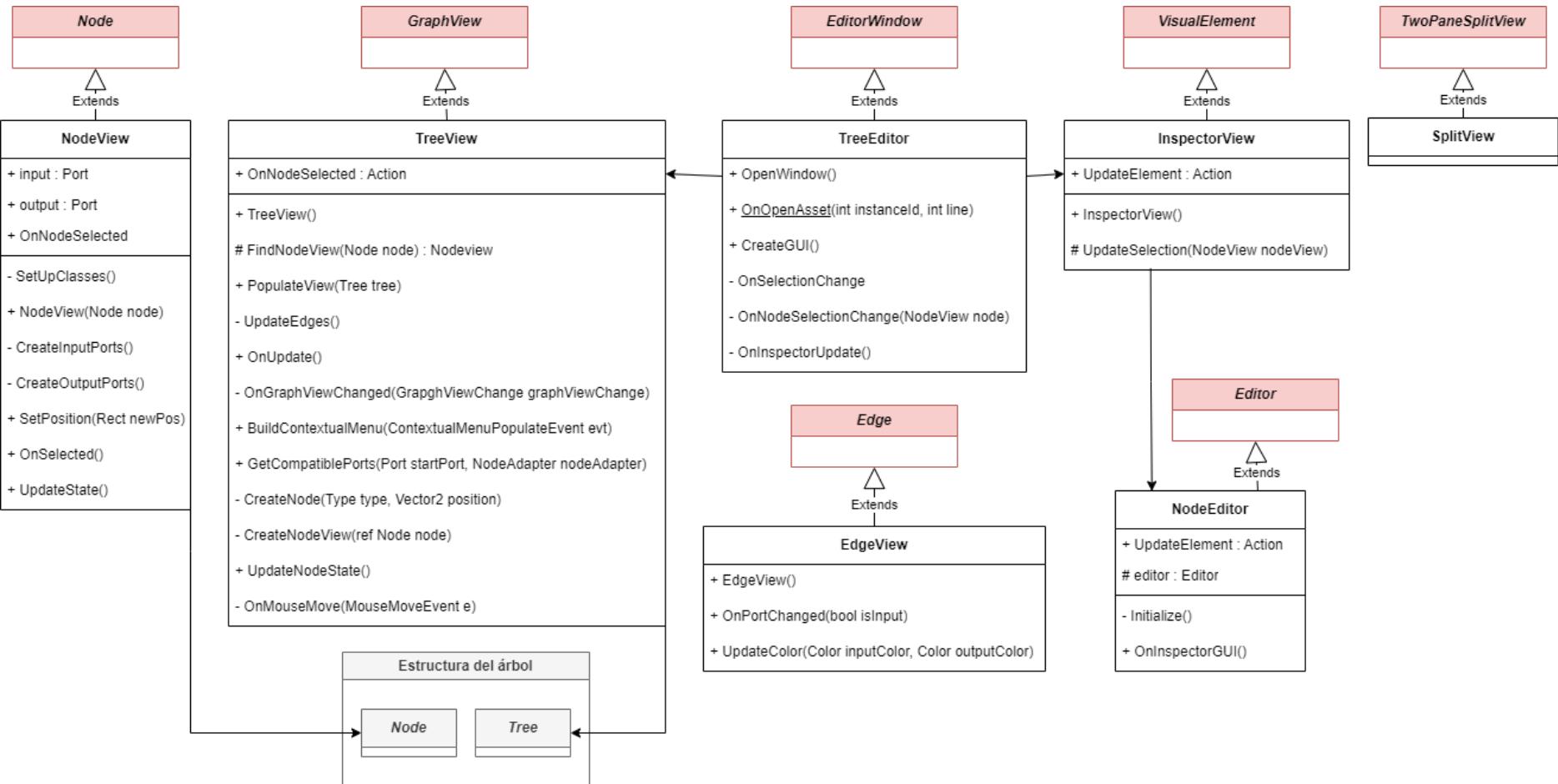
Al igual que en otras implementaciones, vamos a describir la organización general y la forma en que estas partes se comunican entre sí. Luego nos centraremos en las implementaciones específicas de cada clase. Se presentarán los fragmentos de código que se consideren más relevantes con el fin de brindar una mayor claridad en las explicaciones.

5.6.3. Diagrama de clases y comunicación

Mediante el diagrama de la [Figura 5.14](#), buscamos proporcionar una comprensión más clara de la estructura global del editor. En este diagrama, no hay muchas asociaciones directas debido a la forma en que está estructurado *UI Builder* y al funcionamiento subyacente del código en Unity. En su mayoría, las interacciones entre objetos o acciones se realizan a través de funciones sobrescritas de las clases heredadas o mediante eventos. Esta dinámica se analizará con mayor detalle en las secciones individuales de cada clase.

Las únicas clases que interactúan directamente con nuestra estructura de árbol son *TreeView* y *NodeView*. Esta comunicación es bidireccional y es esencial para varios aspectos:

- *TreeView* → *Tree*. La vista informa al árbol sobre la creación o eliminación de un nodo, y el árbol agrega o elimina ese nodo de su lista correspondiente. Además, la vista debe notificar al árbol sobre cualquier asociación (padre-hijo) que se haya establecido o deshecho mediante un enlace, para que el nodo padre pueda administrar su lista de hijos en consecuencia.
- *Tree* → *TreeView*. La vista no retiene la estructura cuando se cierra la ventana del editor. Por lo tanto, al inicio, el árbol proporciona información a la vista sobre cómo debe representar los nodos y enlaces.
- *NodeView* → *Node*. Cualquier modificación en los parámetros de un nodo en la vista debe reflejarse en la estructura del árbol. En nuestro caso, el único parámetro que puede alterarse es si un nodo compuesto es estático o no.
- *Node* → *NodeView*. Los parámetros de los nodos no se almacenan en la vista. Por lo tanto, deben cargarse desde la estructura del árbol y reflejarse en la vista correspondiente.

Figura 5.14 – Diagrama de clases de los *scripts* del editor

5.6.4. Creación de la ventana del editor y acceso directo

En este apartado queremos crear con la clase *EditorWindow* nuestra propia ventana de editor. En nuestro caso hemos heredado de esta clase para crear *TreeEditor*, nuestro editor de árboles. Al heredar de esta clase, nuestro editor ya adquiere muchas de las funcionalidades necesarias para su funcionamiento. Comenzamos con un elemento visual inicial llamado *rootVisualElement*, el cual sirve como base para el resto de elementos en la ventana. Hay varias tareas que deseamos realizar en nuestra ventana de editor para que sea funcional:

1. Creamos instancias (variables) de todos los elementos que están almacenados en otras clases y que queremos que aparezcan en nuestra ventana. En nuestro caso, instanciamos *TreeView* e *InspectorView*, los cuales describiremos más adelante.
2. Deseamos que la ventana muestre algo tan pronto como se abra, por lo que debemos agregar contenido a la función *CreateGUI* de *EditorWindow*. Esta función se ejecuta automáticamente cuando *rootVisualElement* está listo para ser llenado con contenido adicional.

Código 5.1 – Código de la función *CreateGUI*

```

1  public void CreateGUI()
2  {
3      VisualElement root = rootVisualElement;
4
5      //Importamos UXML
6      var visualTree =
7      → AssetDatabase.LoadAssetAtPath<VisualTreeAsset>("Path/To/TreeEditor.uxml");
8      visualTree.CloneTree(root);
9
10     // Añadimos stylesheet al elemento visual raíz.
11     // El estilo se aplica a todos los hijos de ese elemento también.
12     var styleSheet =
13     → AssetDatabase.LoadAssetAtPath<StyleSheet>("Path/To/TreeEditor.uss");
14     root.styleSheets.Add(styleSheet);
15
16     //Cargamos los dos elementos principales de nuestro editor
17     treeView = root.Q<TreeView>();
18     inspectorView = root.Q<InspectorView>();
19
20     //Se suscriben funciones a los eventos necesarios
21     treeView.OnNodeSelected = OnNodeSelectionChange;
22     inspectorView.UpdateEvent += treeView.OnUpdate;
23 }
```

3. También queremos asegurarnos de que la vista en la ventana sea apropiada cada vez que cambiamos el objeto seleccionado. Para lograr esto, identificamos el objeto activo y actualizamos la vista del árbol cada vez que cambiamos de objeto.

Código 5.2 – Código de la función *OnSelectionChange* de *TreeEditor*

```

1  private void OnSelectionChange()
2  {
3      Tree tree = Selection.activeObject as Tree;
4      if (tree && AssetDatabase.CanOpenAssetInEditor(tree.GetInstanceID()))
5      {
6          treeView.PopulateView(tree);
7      }
8 }
```

4. Deseamos que nuestro editor sea accesible desde el menú de Unity, al igual que otros editores. Para lograr esto, agregamos el atributo `MenuItem`, como se ilustra en el ejemplo del manual [88].
5. Por último, deseamos que nuestro editor se abra al hacer doble clic en uno de los elementos que definen nuestro árbol. Para ello, utilizamos el atributo `OnOpenAsset`. Verificamos que el objeto activo sea un árbol (*Tree*) y abrimos la ventana como en el paso anterior.

Código 5.3 – Código de la función `OnOpenAsset` de *TreeEditor*

```

1  [OnOpenAsset]
2  public static bool OnOpenAsset(int instanceId, int line)
3  {
4      if(Selection.activeObject is Tree)
5      {
6          OpenWindow();
7          return true;
8      }
9      return false;
10 }
```

Con esto tenemos un editor completamente funcional y accesible pero aún vacío, hay que implementar las clases que lo complementan, *TreeView* e *InspectorView*.

5.6.5. Uso de *UxmlFactory* y *UxmlTraits*

Usamos la clase *UxmlFactory* [96] y *UxmlTraits* [97] para generar el esquema UXML por defecto de la clase *TwoPaneSplitView*, *TreeView* e *InspectorView*. De esta forma nos permite tener incorporado el UXML en la clase y así poder utilizarla en nuestro *UI Builder* para definir su integración en la interfaz. La jerarquía de *TreeEditor.uxml* quedaría como en la **Figura 5.15**.



Figura 5.15 – Jerarquía del editor en *UI Builder*

Para el caso de *TwoPaneSplitView* no añadimos ninguna funcionalidad extra a la clase, solo necesitamos la base y el código UXML por defecto, tal y como aparece en el **Código 5.4**

Código 5.4 – Código de la clase *SplitView*

```

1  public class SplitView : TwoPaneSplitView
2  {
3      public new class UxmlFactory : UxmlFactory<SplitView, UxmlTraits> { }
4  }
```

5.6.6. Creación de la vista del inspector

Nuestro *InspectorView*, que hereda de *VisualElement*, se compone de un botón que permite actualizar la información de los nuevos nodos del árbol y de un cuadro que muestra la información del nodo que esté actualmente seleccionado, es decir, sobre el cual se haya hecho clic.

Para integrar estos componentes en nuestro *VisualElement*, debemos añadirlos a su contenedor de contenido [98]. Agregar contenido a un *VisualElement* es sencillo gracias al modo de GUI inmediato (*IMGUI*) de Unity [99]. Solo necesitamos crear un *IMGUIContainer* con los elementos que deseamos “dibujar” y agregarlo al contenedor de nuestro *InspectorView*.

Código 5.5 – Código de la función *UpdateSelection* de *InspectorView*

```

1  internal void UpdateSelection(NodeView nodeView)
2  {
3      Clear();
4
5      Object.DestroyImmediate(editor);
6      editor = Editor.CreateEditor(nodeView.node, typeof(NodeEditor));
7
8      IMGUIContainer container = new((() =>
9      {
10         if (GUILayout.Button("UPDATE"))
11         {
12             UpdateEvent?.Invoke();
13         }
14
15         if (editor.target)
16         {
17             EditorGUILayout.BeginVertical(GUILayout.Width(200));
18             editor.OnInspectorGUI();
19             EditorGUILayout.EndVertical();
20         }
21     });
22
23     Add(container);
24 }
```

Hay tres aspectos que comentar del Código 5.5.

1. El editor que se agrega a nuestro *IMGUIContainer* es el correspondiente a la clase *NodeEditor* (que hereda de *Editor*), un *inspector* especial que hemos creado para los nodos. Este lo empaquetamos en un *layout* vertical y ejecutamos su función *OnInspectorGUI* que muestra todos los parámetros del nodo.

Para hacer esto correctamente, primero debemos inicializar el editor con el nodo que se pasa como parámetro a *UpdateSelection*. De esta manera, nuestra clase *NodeEditor* puede leer los parámetros correctamente al tener asociado un nodo de nuestra estructura del árbol.

2. Cuando presionamos el botón “UPDATE”, se llama a un evento. A este evento se deberá suscribir una función que actualice el árbol. Ver Figura 5.16.
3. *UpdateSelection* debe ser llamada cada vez que seleccionemos un nodo del árbol, para que el inspector pueda actualizar su vista. Ver Figura 5.17.

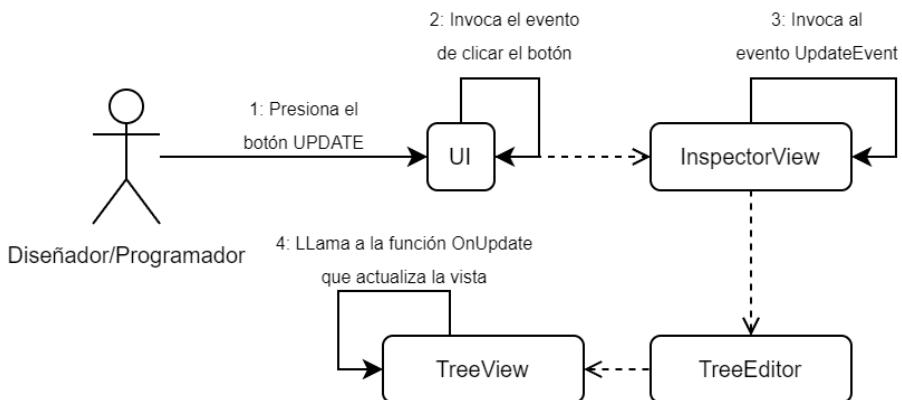


Figura 5.16 – Diagrama de comunicación del botón “UPDATE” de *InspectorView*.

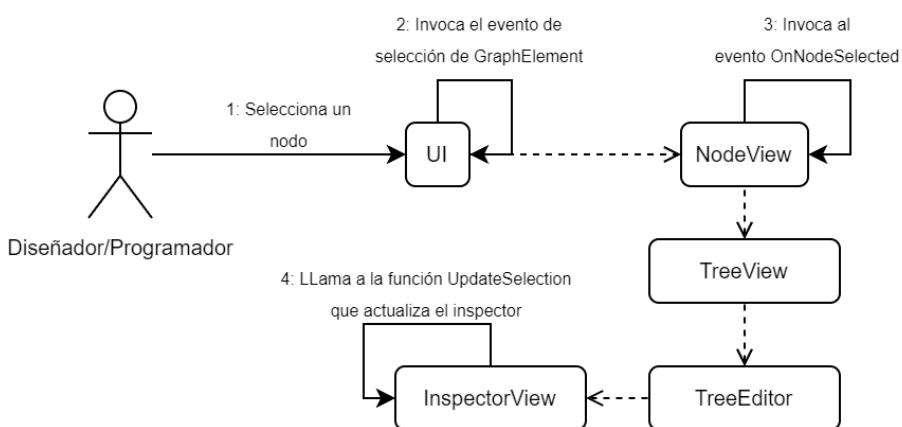


Figura 5.17 – Diagrama de comunicación tras seleccionar un nodo en el editor.

5.6.7. Creación de la vista del árbol

La vista del árbol, representada por la clase *TreeView*, es la parte más compleja, ya que abarca toda la funcionalidad del árbol al heredar de *GraphView*. Para inicializar esta clase, debemos llamar a su constructor, el cual realiza las funciones del [Código 5.6](#), que no es más que añadir los manipuladores necesarios para poder navegar con el ratón por la vista, agregar un fondo de cuadrícula, cargar el *StyleSheet* correspondiente y registrar la posición del ratón en la vista.

Código 5.6 – Código del constructor de la vista del árbol

```

1  public TreeView()
2  {
3      Insert(0, new GridBackground());
4
5      this.AddManipulator(new ContentZoomer());
6      this.AddManipulator(new ContentDragger());
7      this.AddManipulator(new SelectionDragger());
8      this.AddManipulator(new RectangleSelector());
9
10     var styleSheet = AssetDatabase.LoadAssetAtPath<StyleSheet
11     ("Path/To/TreeEditor.uss");
12     styleSheets.Add(styleSheet);
13
14     RegisterCallback<MouseMoveEvent>(OnMouseMove);
    }
```

Con todo lo que hemos realizado hasta ahora, deberíamos tener una ventana de editor parecida a la de la [Figura 5.18](#). Ahora tenemos que permitir a la vista la creación, manipulación y eliminación de nodos y enlaces.

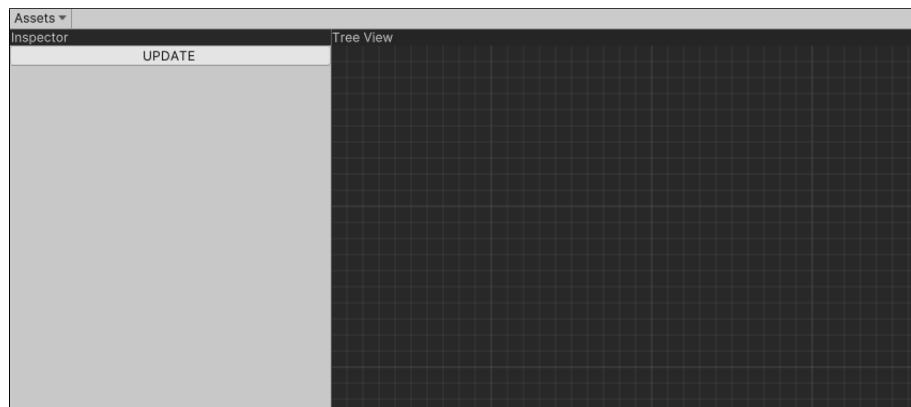


Figura 5.18 – Ventana del editor con la vista de árbol por defecto.

5.6.7.1. Creación de la vista del nodo

Los nodos que se pueden crear por defecto en nuestro *GraphView* no son relevantes para nosotros, por lo que vamos a crear nuestra propia vista de nodos o *NodeView*, la cual hereda de *Node*. Cada elemento de esta vista debe contener una instancia de un nodo de nuestra estructura del árbol, de manera que podamos acceder a sus parámetros y configurar su aspecto visual y funcionalidad en función de estos. Por lo tanto, crear un *NodeView* implica pasar una referencia del nodo correspondiente en la estructura del árbol como parámetro, lo que permite inicializarlo adecuadamente en el constructor.

Código 5.7 – Código del constructor de la vista del nodo

```

1  public NodeView(Node node) : base("Path/To/NodeView.uxml")
2  {
3      this.node = node;
4      title = node.name;
5      viewDataKey = node.guid;
6
7      style.left = node.position.x;
8      style.top = node.position.y;
9
10     CreateInputPorts();
11     CreateOutputPorts();
12     SetUpClasses();
13 }
```

5

En el [Código 5.7](#), además de la inicialización de parámetros, se llaman a tres funciones importantes para el aspecto visual y la funcionalidad de los nodos en la vista.

1. **CreateInputPorts**. Esta función crea los puertos de entrada que permiten conectar enlaces. Dado que los puertos de entrada se conectan con el padre, es necesario verificar si el nodo actual implementa la interfaz *IHaveParent*, y si es así, se le añade un puerto de entrada.
2. **CreateOutputPorts**. Esta función es similar a la anterior, pero verifica la interfaz *IHaveChildren*, y asigna múltiples puertos de salida para poder tener uno o varios hijos. Se debe tener en cuenta el caso especial de *RootNode*, que solo puede tener un puerto de salida.
3. **SetUpClasses**. En nuestro archivo *NodeView.uxml*, hemos creado varios *StyleSheet* que definen los diferentes tipos de nodos instanciables en nuestro árbol: compuestos, hojas y raíces. Estos estilos se pueden definir en el archivo UXML como tipos a los que asignar una vista concreta. En otras palabras, podemos asociar un *NodeView* a un tipo de estilo específico en UXML, lo que dará a cada tipo de nodo un aspecto visual diferente. La función *SetUpClasses* se utiliza para asignar los *NodeView* a los tipos de estilos.

5.6.7.2. Integración de *NodeView* en la vista del árbol

La integración de la vista del nodo en nuestro gráfico implica proporcionar al usuario una forma de crear nodos. Para lograr esto, sobreescriviremos la función *BuildContextualMenu* de *GraphView*, añadiendo como opciones posibles todos los nodos que hereden de un nodo compuesto o un nodo hoja. La acción de crear un nodo requiere dos pasos: por un lado, crear el correspondiente *ScriptableObject* en la estructura del árbol y añadirlo a la lista, y por otro lado, crear la vista del nodo y agregarla al *GraphView*. Consulta el [Código 5.8](#) para más detalles.

Código 5.8 – Código de creación de nodos en *TreeView*

```

1  public override void BuildContextualMenu(ContextualMenuPopulateEvent evt)
2  {
3      foreach (var type in TypeCache.GetTypesDerivedFrom<LeafNode>())
4          evt.menu.AppendAction($"[{type.BaseType.Name}] {type.Name}", (a) =>
5              CreateNode(type, localMousePosition));
6
7      foreach (var type in TypeCache.GetTypesDerivedFrom<CompositeNode>())
8          evt.menu.AppendAction($"[{type.BaseType.Name}] {type.Name}", (a) =>
9              CreateNode(type, localMousePosition));
```

```

8    }
9
10   void CreateNode(System.Type type, Vector2 position)
11  {
12      Node node = tree.CreateNode(type, position);
13      CreateNodeView(ref node);
14  }
15
16  void CreateNodeView(ref Node node)
17  {
18      NodeView nodeView = new(node);
19      nodeView.OnNodeSelected = OnNodeSelected;
20      AddElement(nodeView);
21  }

```

OnGraphViewChanged es una función que se activa automáticamente por eventos cuando el gráfico experimenta algún cambio. Si deseamos que la eliminación de nodos en nuestra estructura se aplique correctamente, podemos agregar el fragmento de código que se encuentra en el [Código 5.9](#).

Código 5.9 – Código de la función *OnGraphViewChange* de *TreeView*

```

1  private GraphViewChange OnGraphViewChanged(GraphViewChange graphViewChange)
2  {
3      graphViewChange.elementsToRemove?.ForEach(elem =>
4      {
5          if (elem is NodeView nodeView)
6          {
7              tree.DeleteNode(nodeView.node);
8          }
9      });
10
11     return graphViewChange;
12 }

```

También es necesario agregar la funcionalidad de creación de nodos a la función *PopulateView*, que se encargará de actualizar la vista cuando abramos el editor por primera vez o realicemos cambios. Para lograr esto, incorporamos el fragmento de código que se encuentra en el [Código 5.10](#). Este fragmento crea el nodo raíz en caso de que no exista y genera las vistas de todos los nodos presentes en nuestra estructura del árbol. Antes de hacer esto, es importante eliminar cualquier nodo existente y limpiar la vista para prevenir posibles errores.

Código 5.10 – Código de la función *PopulateView* de *TreeView*

```

1  internal void PopulateView(Tree tree)
2  {
3      this.tree = tree;
4
5      //Desabilitar el evento que gestiona los cambios en el gráfico.
6      graphViewChanged -= OnGraphViewChanged;
7
8      //Eliminar todos los elementos del gráfico
9      DeleteElements(graphElements);
10
11     //Volver a activar el evento
12     graphViewChanged += OnGraphViewChanged;
13 }

```

```

14     if(tree.rootNode == null)
15         tree.rootNode = tree.CreateNode(typeof(RootNode), new Vector2(0,0)) as
16     RootNode;
17
18     tree.nodes.ForEach(n => CreateNodeView(ref n));
19 }
```

Como resultado de esta integración, podemos llevar a cabo la creación de nodos en nuestro gráfico a través de un menú desplegable como el de la [Figura 5.19](#). Ahora podemos crear todos los nodos que necesitemos, y estos serán almacenados en la lista de *ScriptableObjects*. Del mismo modo, si deseamos borrar nodos, se eliminarán de la lista correspondiente. Sin embargo, todavía necesitamos establecer la estructura del árbol, permitiéndonos agregar enlaces padre-hijo.

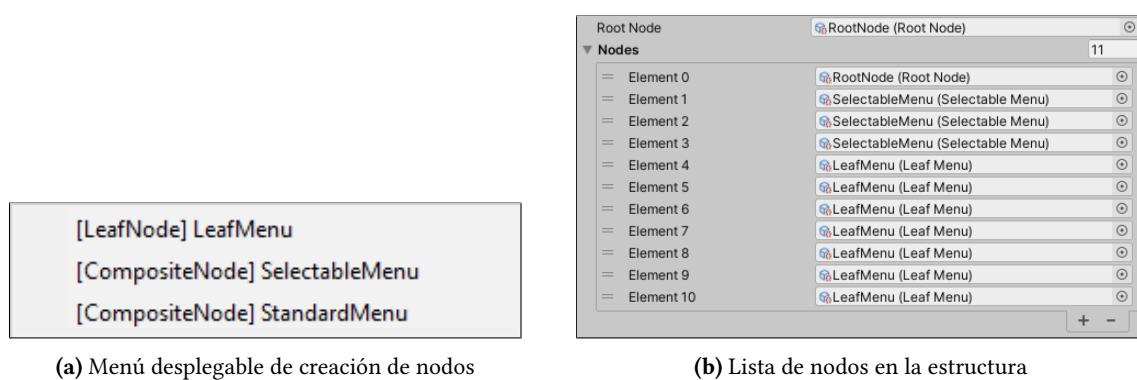


Figura 5.19 – Creación de nodos dentro y fuera del editor

5.6.7.3. Creación de la vista del enlace

La clase *EdgeView* hereda de *Edge* y no presenta ninguna particularidad especial, a excepción de que hemos agregado una nueva funcionalidad: la capacidad de cambiar el color de entrada y salida del enlace. Hemos implementado esta función para poder distinguir el orden en que se asignan los enlaces de salida. Nos referimos a que cuando asignamos un enlace de padre a hijo, el puerto de salida es el mismo para todos los enlaces, lo que impide determinar qué hijo se ha asignado primero y, por lo tanto, el orden de los hijos. Con esta nueva función, ahora podemos asignar un gradiente de color que permita diferenciarlos. Ver [Figura 5.20](#).

5.6.7.4. Integración de *EdgeView* en la vista del árbol

Para integrar los enlaces que hemos definido en nuestra vista del árbol solo debemos agregar algunas líneas más a nuestras funciones *PopulateView* y *OnGraphViewChanged*. En la primera para obtener la información de la estructura del árbol y poder actualizar nuestro gráfico.

Código 5.11 – Integrar la vista del enlace en *PopulateView*

```

1 internal void PopulateView(Tree tree)
2 {
3     ...
4
5     //Actualizamos la vista con la estructura del árbol
```

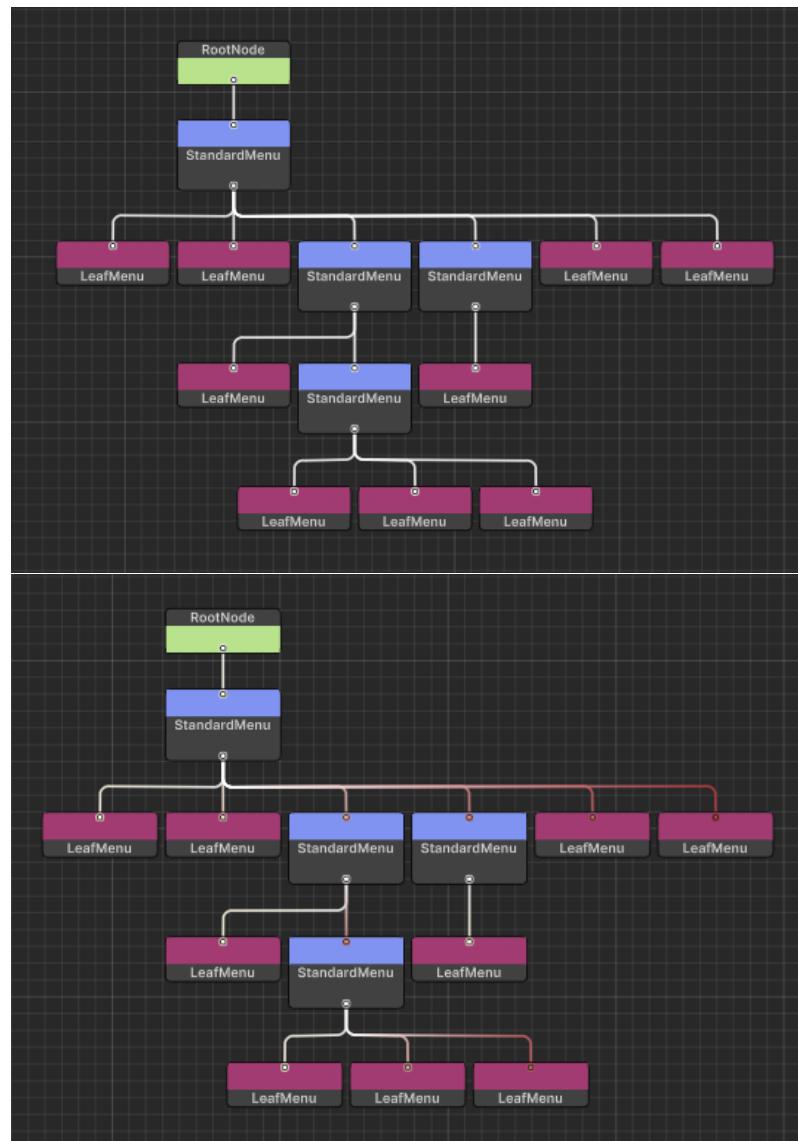


Figura 5.20 – Comparativa entre enlaces con y sin diferencia de color en asignación

```

6     UpdateEdges();
7
8     //Asignamos el ID a los nodos en la estructura
9     tree.AsignID();
10

```

En la segunda para poder crear y deshacer asociaciones padre-hijo en nuestra estructura cuando el gráfico detecte que hemos creado o destruido un nuevo enlace.

Código 5.12 – Integrar la vista del enlace en *OnGraphViewChanged*

```

1  private GraphViewChange OnGraphViewChanged(GraphViewChange graphViewChange)
2  {
3      ...
4
5      graphViewChange.elementsToRemove?.ForEach(elem =>
6      {
7          if (elem is Edge edge)
8          {
9              NodeView parentView = edge.output.node as NodeView;
10             NodeView childView = edge.input.node as NodeView;
11             tree.RemoveChild((IHaveChildren)parentView.node,
12             (IHaveParent)childView.node);
13         }
14     });
15
16     graphViewChange.edgesToCreate?.ForEach(edge =>
17     {
18         NodeView parentView = edge.output.node as NodeView;
19         NodeView childView = edge.input.node as NodeView;
20         tree.AddChild((IHaveChildren) parentView.node, (IHaveParent)
21         childView.node);
22     });
23
24     ...
25 }

```

5.6.7.5. Comunicación durante la ejecución: Nodo seleccionado

Consideramos que es necesario establecer una comunicación adicional entre la estructura del árbol y el editor para representar, durante la ejecución del juego, qué nodos están actualmente visibles. Esto proporciona al usuario una retroalimentación sobre si está navegando por los menús correctamente según la lógica interna. Para lograrlo, debemos seguir los siguientes pasos:

1. Debemos definir diferentes “estilos” de nodo en nuestro UXML, para los nodos seleccionados y no seleccionados. Luego, en la clase *NodeView*, necesitamos crear una función que asigne la vista a uno de esos estilos, según el valor del parámetro *Selected* del nodo que se le ha asignado.
2. Creamos una función en la clase *TreeView* que itere a través de todos los nodos y llame a la función anterior para establecer sus estilos.
3. En la clase *TreeEditor*, agregamos a la función *OnInspectorUpdate*, que se actualiza a 10 fotogramas por segundo, la llamada a la función de la vista del gráfico mencionada en el paso anterior.

El resultado final se representa en la [Figura 5.21](#)

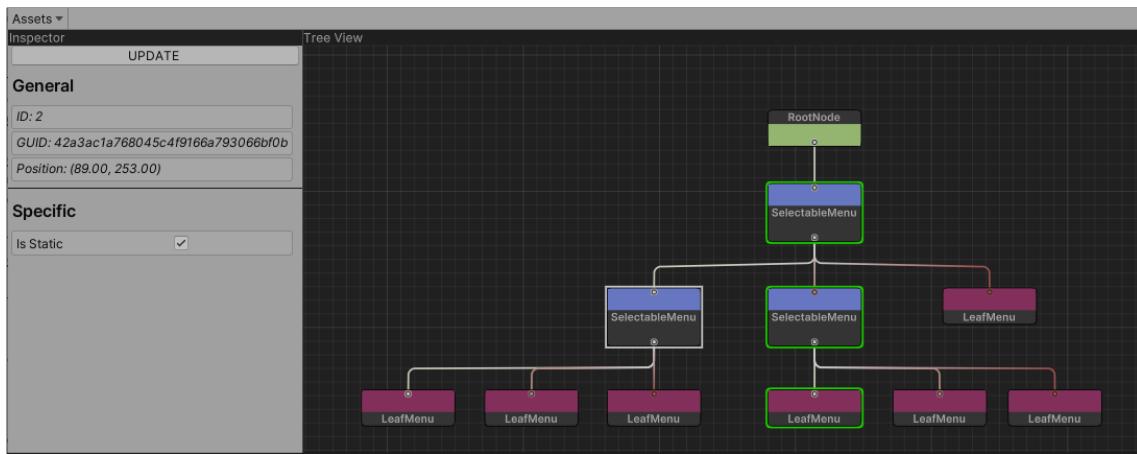


Figura 5.21 – Representación de nodos seleccionados en ejecución

5.7. Conclusiones

Finalmente y tras una tediosa implementación, tenemos un editor de árboles en Unity, totalmente personalizable y ajustable, que podemos utilizar para cualquier función que requiera de este tipo de estructuras. Todavía queda mucho trabajo para poder tener una versión que se pueda mostrar cara al público, pero tenemos una base sólida sobre la que construir y que de momento funciona a la perfección para un menú como el de nuestro juego. Su uso y las posibles mejoras se hablan en el [Capítulo 6](#).

Con este capítulo esperamos haber sido de inspiración para otros desarrolladores y haber dado la información suficiente no solo para comprender nuestro entorno de trabajo, sino para poder idear muchos otros de estas características y así mejorar el desarrollo de videojuegos en esta plataforma.

CAPÍTULO 6

Caso de uso: Carnem Levare

En este capítulo, daremos un paso adelante al integrar la implementación realizada en los capítulos anteriores en el contexto de nuestro videojuego en desarrollo. A lo largo de estas páginas, analizaremos minuciosamente los resultados que emergen de estas implementaciones. Nos adentraremos en su comportamiento en el contexto del videojuego, evaluando si se ajustan a nuestras expectativas y si cumplen con los objetivos que nos propusimos. Este análisis detallado nos permitirá identificar posibles áreas de mejora o ajustes necesarios para garantizar su integración en nuestro proyecto.

Consideramos este capítulo esencial, ya que contribuirá a una mejor comprensión de las implementaciones, y complementa los conceptos presentados en el [Capítulo 4](#), brindándonos un ejemplo de la aplicación de cada uno de los sistemas y consolidando así nuestro conocimiento.

Es importante mencionar que en este capítulo se manejan con fluidez los *scripts* y las funciones que se han detallado exhaustivamente en el [Capítulo 4](#). Por lo tanto, si en algún momento alguna explicación resulta confusa, es posible recurrir a la sección correspondiente de dicho capítulo para una comprensión más completa.

6.1. Acerca de Carnem Levare

Antes de sumergirnos en el funcionamiento de cada sistema, permíteme guiarte por un viaje a través de las entrañas de nuestro propio videojuego. Si bien mi compañero ha ahondado profundamente en todos los aspectos de diseño y elementos específicos, considero fundamental que conozcas la esencia de Carnem Levare para comprender las decisiones que se despliegan en este capítulo.

6.1.1. Un Vistazo a la Temática

Carnem Levare no es un simple videojuego de acción en 3D; es una inmersión en la esencia misma del combate. Nos sumerge en un mundo que trasciende la frontera entre la vida y la muerte, un lugar donde los condenados hallan su destino final: un enigmático carnaval. Aquí, el combate se convierte en el núcleo mismo de la existencia. Dos contendientes se enfrentan en una lucha a vida o muerte, un concurso que atrae a un público de todo tipo.

Adentrarse en Carnem Levare es ingresar a un reino que desafía las normas de la realidad. Los luchadores no solo pelean por su supervivencia, sino que también compiten por ser los protagonistas de este siniestro espectáculo. Cada golpe, cada movimiento, es un paso hacia la victoria y una declaración de estilo en medio de esta danza macabra.

6.1.2. Explorando el Género

Pasemos ahora a un análisis más profundo del género que encapsula Carnem Levare. No nos conformamos con encasillar nuestro juego en una única categoría; aspiramos a algo más di-

námico y festivo, como un carnaval que abraza múltiples dimensiones. Aunque el corazón del juego late al ritmo de la lucha, nos esforzamos por infundir aspectos narrativos y jugables que lo singularicen y distingan de la multitud.

Mientras examinamos los rasgos esenciales que definen a Carnem Levare como un juego de lucha, emergen dos pilares que sostienen nuestra visión: la profundidad y la versatilidad. Concebimos el combate como un escenario donde los jugadores pueden exprimir al máximo sus habilidades y tácticas. En esta arena, se entrelazan mecánicas emblemáticas del género de lucha, como la perfecta sincronización de un *Parry* [100], los combos, el arte de bloquear con precisión y la agilidad en las evasiones.

Sin embargo, no nos contentamos con trillar caminos ya conocidos. En Carnem Levare, incorporamos elementos que lo distinguen y lo hacen una experiencia única. La esencia del boxeo se entrelaza con el arte del esquive, dotando al jugador con maniobras evasivas. Uno de nuestros puntos de inspiración más significativos fue un video donde Muhammad Ali logra esquivar veintiún puñetazos en tan solo diez segundos [101]. Pero es en la asignación de movimientos cuando damos un salto donde, en nuestra opinión, podemos destacar. Damos a los jugadores el poder de personalizar su estilo de lucha, permitiéndoles seleccionar los movimientos con los que se sienten más conectados. Esta innovación no solo añade profundidad estratégica, sino que también infunde el juego con un toque personal y una huella única. Además las recompensas de luchar contra ciertos enemigos será ganar su experiencia, obteniendo para tu arsenal movimientos únicos a los que te has enfrentado.

A pesar de que la idea de la asignación de movimientos no es del todo novedosa, ya que títulos como Absolver (2017) [102], por ejemplo, exploraron algo similar, en Carnem Levare esta idea se convierte en un distintivo que va más allá de las convenciones tradicionales.

6.1.3. Aspectos de la implementación actual

El videojuego actualmente encierra gran parte de la esencia que habíamos imaginado desde el principio, aunque en este momento sigue siendo una versión preliminar en comparación con el resultado final que esperamos alcanzar. Hasta ahora, nos hemos centrado en garantizar que los sistemas fundamentales y el combate estén bien implementados, relegando en cierta medida los aspectos artísticos y narrativos para asegurar una base sólida sobre la cual construir.

La versión de demostración que hemos preparado actualmente incluye un menú principal donde los jugadores pueden iniciar una partida, ajustar las opciones del juego o salir del mismo. Si un jugador decide comenzar una partida, será llevado a una sala de entrenamiento mediante una pantalla de carga. Allí encontrará un maniquí que le permitirá poner a prueba los movimientos que tenga equipados. Además, podrá utilizar un menú de selección de movimientos para asignarse nuevos si es que los tiene disponibles. Una vez que el jugador se sienta preparado para el combate, podrá enfrentarse a un enemigo en una zona de combate diferente. Si el jugador logra vencer al oponente, obtendrá nuevos movimientos y volverá a la zona de entrenamiento para equiparse con ellos y probarlos. En caso de que el jugador sea derrotado, regresará al menú principal donde podrá iniciar otra partida.

Mientras el jugador se encuentre en la zona de combate o entrenamiento, podrá cambiar las opciones o regresar al menú principal siempre que lo desee mediante un menú de pausa.

6.2. Sistema de Inputs

Cuando estamos desarrollando un juego de lucha, el sistema de *inputs* debe ser altamente responsivo, ya que necesitamos respuestas precisas y rápidas. Además, debe admitir una amplia variedad de movimientos, lo que incluye interacciones como mantener pulsado o hacer doble pulsación. La precisión también es fundamental en este contexto, y el sistema de *inputs* debe tener la capacidad de respaldar la creación de un sistema de combate profundo, en caso de que se desee implementar.

6.2.1. Estructura

Durante la implementación y utilización de este sistema, hemos observado que el acceso a través de *GameManager* a diversos *scripts* permite la posibilidad de acceder a funcionalidades que pueden no ser necesarias desde el exterior. Además, actualmente existen tres *scripts* distintos del sistema de inputs incorporados en nuestro *GameManager*, una práctica que consideramos que no es completamente óptima. Consideramos que es más apropiado adoptar un enfoque donde cada sistema esté independizado mediante interfaces, siguiendo el modelo aplicado en el sistema de guardado. Esta estructura permitiría ofrecer acceso únicamente a las funciones específicas de cada *script*, evitando el exceso de acoplamiento. Abordamos este desafío previamente al discutir el patrón de diseño fachada, y como solución sugerimos la implementación del principio de Segregación de Interfaces (ISP) de SOLID [42].

6.2.2. InputSystem/InputReader

La utilización del plugin *InputSystem* para establecer nuestro sistema de *inputs* resulta sumamente beneficiosa en el contexto de un juego de lucha, ya que nos brinda las herramientas necesarias para ajustar parámetros y agregar diversas interacciones a cada una de las acciones que hayamos definido [103]. Además, los eventos que se generan al presionar una tecla en cada acción incluyen un parámetro *CallbackContext* [104], el cual nos proporciona información esencial acerca de la acción y los *bindings* en nuestra clase *InputReader*. Este enfoque simplifica considerablemente la obtención de información relacionada con los *inputs*. Como ejemplo, al definir un evento en *InputReader* destinado a activar el movimiento del personaje, podemos fácilmente transmitir información valiosa, como la dirección del joystick que el jugador está utilizando.

Asimismo, nuestra elección de crear un manejador de eventos mediante *ScriptableObject* demuestra ser muy acertada. La asignación de funciones a los *inputs* es directa y eficiente. Mi compañero únicamente necesita establecer una referencia al *InputReader* definido en la estructura de directorios, lo que le permite acceder a él y suscribir las funciones necesarias, siempre teniendo en cuenta los parámetros requeridos por el evento. Dado que la conexión entre el *InputReader* y el *PlayerInput* puede realizarse de manera independiente, se logra una adecuada separación entre las acciones, los *bindings* y las funciones específicas de nuestro juego (como se ilustra en la Figura 4.7).

En consecuencia, obtenemos un resultado que nos brinda una efectiva definición de nuevas

funcionalidades, ya sean más complejas o no. Esta implementación se traduce en una alta escalabilidad y su integración con cualquier segmento de código es eficiente. Con un enfoque altamente reactivo, nuestro código no depende en ningún momento del ciclo de juego, aprovechando los eventos para lograr un buen nivel de precisión.

6.2.3. Vibración/Reasignación de inputs

La implementación de la vibración se realiza a través de un único *script*, el cual es accesible mediante *GameManager* en cualquier parte del juego. Esto nos brinda un acceso rápido y directo sin necesidad de referencias adicionales, ya que tanto *GameManager* como *RumbleController* son instancias estáticas. La facilidad de acceso a la funcionalidad de vibración nos permite incorporarla en distintos contextos, ya sea en menús, combates o secuencias cinematáticas, enriqueciendo la inmersión del jugador en nuestra propuesta de juego.

En contraste, la reasignación de *inputs* no es un *script* de acceso global. Sin embargo, esto no supone un problema, dado que en la mayoría de los juegos, incluido el nuestro, las opciones de reasignación se configuran en un entorno seguro, como un menú de opciones. Para llevar a cabo esta reasignación, simplemente debemos crear una instancia de la clase correspondiente en el lugar donde necesitemos realizar los ajustes. Esta funcionalidad resulta esencial para asegurar que todos los jugadores puedan disfrutar de nuestro juego a través de su propio esquema de controles personalizado, contribuyendo a una jugabilidad altamente adaptable y personalizable.

6.3. Sistema de Audio

El sistema de audio desempeña un papel fundamental en la experiencia de juego de un título de lucha, especialmente en un entorno tridimensional. Una buena implementación contribuye significativamente a la inmersión del jugador, permitiendo que los impactos de los golpes y las acciones resuenen de manera más contundente. En este contexto, la capacidad de generar audio nítido y preciso adquiere relevancia, ya que en los juegos de lucha, los sonidos pueden ser sutiles pero impactantes. Ejemplos de esto incluyen el siseo del aire durante un movimiento rápido o el sonido característico de un puñetazo fallido. A pesar de su brevedad, estos efectos de sonido contribuyen a enriquecer la percepción auditiva del jugador durante un combate.

6.3.1. Estructura

En nuestra implementación de la gestión de sonido, hemos desarrollado un enfoque que permite organizar los efectos de audio en distintos grupos, cada uno con su propio control de volumen y efectos. Optamos por dividir nuestros sonidos en tres grupos (*SoundStructures*):

1. *GameSoundEffects*: Esta estructura alberga los efectos de sonido que deben ser inicializados en una escena de combate, es decir, los efectos sonoros inherentes al propio juego.
2. *UISoundEffects*: Esta estructura engloba los efectos de sonido de los menús de nuestro juego, y su inicialización se realiza en cualquier escena que incluya elementos de interfaz.
3. *Music*: En esta estructura, gestionamos la música del juego, la cual se activa en todas las escenas.

Para la creación de estos grupos, no fue necesario intervenir en el código en ningún momento. Bastó con crear tres instancias de *SoundStructure* en la jerarquía de directorios y proceder a la inicialización de los sonidos en la estructura correspondiente. De esta forma tenemos todos los sonidos inicializados desde un inicio, independizados para su variación por separado, es decir, nos facilita mucho la creación de opciones de audio en nuestro menú de opciones. Además, esta flexibilidad nos brinda la capacidad de modificar las opciones de cada sonido de manera individual, un aspecto crucial para parametrizar de forma precisa los efectos sonoros en un juego de lucha, donde la precisión es esencial.

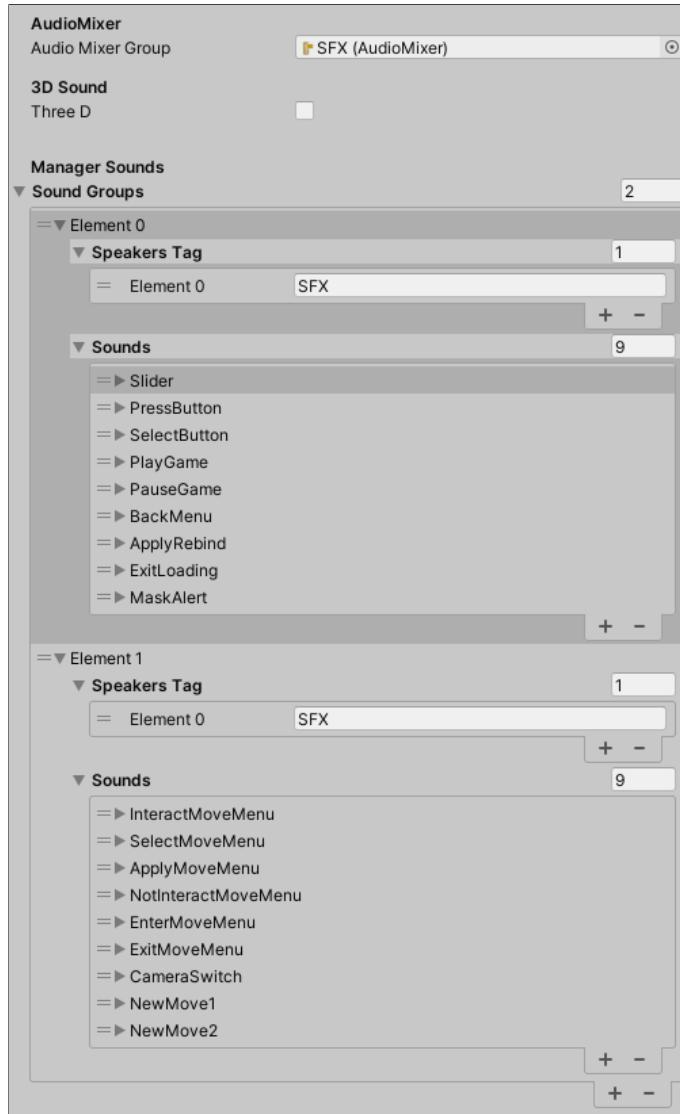


Figura 6.1 – Definición de *UISoundEffects*

La [Figura 6.1](#) ilustra cómo se define la estructura *UISoundEffects*. Si examinamos la parte inicial, notaremos que el primer parámetro es el *AudioMixerGroup*, que hemos configurado de manera idéntica para tanto *GameSoundEffects* como *UISoundEffects*. Esta elección nos permite controlar conjuntamente el volumen de todos los efectos de sonido, aunque los mantengamos separados para diferenciar entre audio en 3D y 2D.

En *GameSoundEffects*, hemos establecido los sonidos generados por jugadores y enemigos, que actualmente son los únicos elementos presentes en la escena de lucha, además de la música.

En cuanto a la etiqueta *Speakers Tag*, se utiliza para indicar que estos sonidos se inicializarán en los objetos de la escena que estén etiquetados como *PLAYER_SOURCES* y *ENEMY_SOURCES*. Dado que en nuestro juego tanto el jugador como el enemigo comparten los mismos efectos de sonido, hemos asignado ambos grupos conjuntamente. No obstante, siempre existe la opción de definir un SoundGroup distinto para el jugador y el enemigo, lo que permitiría configuraciones separadas para los mismos sonidos.

En relación a *UISoundEffects*, es relevante mencionar otra de las funcionalidades de esta estructura. Todos los efectos de sonido de la interfaz de usuario se inicializan en un único objeto de la escena etiquetado como *SFX*. Esto se debe a que, al tratarse de sonidos en 2D, el objeto específico de la escena donde se realiza la inicialización es irrelevante. Incluso, es posible tener dos SoundGroups diferentes que se inicialicen en el mismo objeto, simplemente con el propósito de mejorar la organización visual en el inspector o con fines de una división lógica de sonidos.

La creación de la estructura de sonidos y su posterior inicialización se presenta como un proceso sumamente sencillo y flexible, otorgándonos las opciones necesarias para incorporar efectos de sonido que enriquezcan nuestra experiencia de juego desde las primeras etapas de desarrollo. Sin embargo, es importante señalar un aspecto limitante que hemos identificado: hasta el momento, la implementación no contempla la creación de objetos con sonido durante la ejecución del juego. No obstante, esta limitación no es difícil de superar. Por ejemplo, podríamos diseñar un *script* de inicialización que, al recibir una estructura de sonido (*SoundStructure*), la aplique al objeto que aloje dicho *script*, al mismo tiempo que notifica al controlador de que se han añadido nuevos sonidos.

6.3.2. Controlador y acceso

Ya vimos que todos nuestros sonidos, sin importar la estructura en la que se encuentren, están enlazados a una tabla hash que almacena sus referencias mediante un nombre único. Además, hemos creado métodos en el controlador, accesibles a través de *GameManager*, que abarcan las funciones más comunes de la gestión de sonido, tales como reproducción, detención, pausa o modificación de parámetros. La utilización de estas funciones solo demanda el conocimiento del nombre del sonido que deseamos manipular. Dado que el controlador es estático, si mi compañero quisiera que los movimientos de su personaje emitieran un sonido, solo necesita acceder al controlador y emplear el método con el nombre correspondiente. Para evitar definir los nombres de los sonidos directamente en el código, la opción óptima es crear una variable serializable en el inspector, lo que permite ingresar y alterar el nombre del sonido desde fuera sin necesidad de modificar los *scripts*.

En nuestra perspectiva, esta estructura brinda un acceso global que es tanto limpio como eficiente una vez que los sonidos han sido inicializados. Sin embargo, el principal inconveniente reside en que el controlador debe llevar a cabo la inicialización de los sonidos en cada escena mediante el uso de etiquetas [59]. Aunque no se recomienda el empleo de esta función [61], su implementación no debería representar un obstáculo para proyectos de tamaño moderado. No obstante, hemos optado por investigar más a fondo la complejidad de esta función y su eficiencia real. Para ello, hemos utilizado este Benchmark interactivo de código abierto [105], que evalúa diversas herramientas de Unity con el propósito de comparar su eficiencia.

6.3.2.1. Benchmark y posible solución

El benchmark se configura con una estructura en la escena que consta de cincuenta objetos principales, cada uno de los cuales contiene cincuenta objetos secundarios, y dentro de estos últimos hay cinco objetos adicionales. Durante la prueba, se intentará realizar la búsqueda de alguno de estos objetos una cantidad X de iteraciones, siendo posible ajustar el número de iteraciones según nuestras necesidades.

En la [Tabla 6.1](#) y la [Figura 6.2](#) se exponen los resultados obtenidos para un máximo de 1000 iteraciones de las funciones *FindGameObjectsWithTag* y *FindObjectOfType*. Se optó por utilizar *FindObjectOfType* para poder realizar una comparación con otra funcionalidad de Unity. Como se puede observar, la búsqueda mediante etiquetas no es tan lenta como la búsqueda por tipos, ya que esta última itera a través de cada uno de los objetos definidos en la escena, lo que la convierte en una búsqueda más exhaustiva. Por el contrario, la búsqueda mediante etiquetas se limita únicamente a los objetos que poseen dichas etiquetas.

Iteraciones	Etiquetas	Tipo de objeto
100	0,024	0,39
200	0,041	0,766
300	0,064	1,153
400	0,082	1,545
500	0,103	1,89
600	0,125	2,085
700	0,149	2,499
800	0,165	3,234
900	0,185	3,499
1000	0,202	3,789

Cuadro 6.1 – Benchmark de búsqueda mediante etiquetas

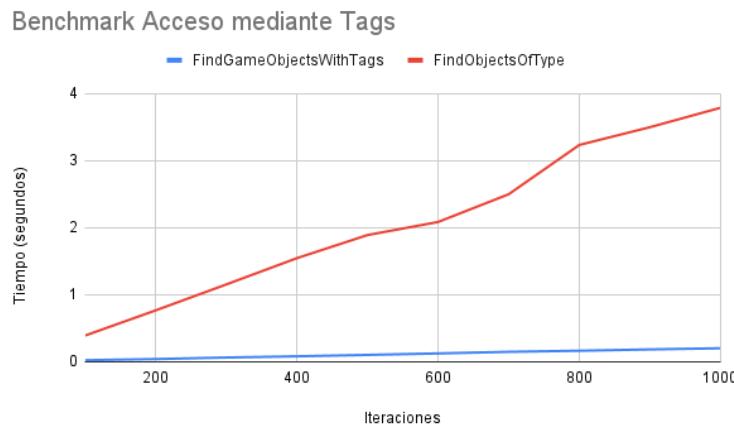


Figura 6.2 – Benchmark de búsqueda mediante etiquetas

A pesar de que mil iteraciones en una sola escena representan una cantidad significativa de objetos para inicializar, especialmente en proyectos muy extensos, observamos que el tiempo requerido para llevar a cabo esta operación no es excesivo y, en caso de que el proyecto continúe expandiéndose, es posible abordar esta cuestión mediante la implementación de una pantalla de carga. Asimismo, hemos concebido una solución viable que afrontaría tanto este problema como

el mencionado en la sección anterior: la delegación del proceso de inicialización a cada uno de los objetos que requieren sonidos. En otras palabras, en lugar de que el controlador busque activamente estos objetos, estos últimos dispondrían de una referencia a la *SoundStructure* necesaria y se inicializarían de forma independiente al inicio de la escena. Esta propuesta resulta totalmente compatible con nuestro controlador y evitaría cualquier tipo de algoritmo de búsqueda.

6.4. Sistema de Guardado

En cualquier videojuego, un sistema de guardado sólido resulta esencial para conservar información relevante entre diferentes ejecuciones. Aunque es cierto que los juegos de lucha, en su mayoría, no incorporaban sistemas de guardado debido a sus raíces en la era de los salones recreativos, gradualmente este género se ha ido incorporando a nuevas narrativas que requieren la funcionalidad de guardado debido a su mayor extensión.

6.4.1. Estructura y funcionamiento

En este apartado, emplearemos la misma estructura que se presentó en la implementación del sistema de guardado local. Consta de dos clases que heredan de la clase abstracta *SaveSlot*:

1. *OptionSlot*: Esta clase almacena toda la información relevante para la configuración del juego, como opciones de audio, visualización, controles y accesibilidad.
2. *GameSlot*: Esta clase almacena la información relacionada con el progreso del juego. En un máximo de tres slots de guardado del juego, se guarda la información de movimientos desbloqueados por el jugador y aquellos que se encuentran actualmente equipados.

Con la implementación del sistema de guardado que tenemos, hacer que este funcione es muy sencillo:

1. Definimos las variables que representan todos los datos que necesitamos guardar en las clases *OptionSlot* y *GameSlot*.
2. Inicialización de los objetos *ScriptableObject*s con los valores por defecto.
3. Transferir estos objetos al *GameManager* para permitir la inicialización del *DataSaver* a través de la interfaz.
4. Acceder a los métodos *Load* o *Save* de la interfaz para cargar o guardar elementos en los archivos.
5. Usar los métodos *GetGameSlot* o *GetOptionsSlot* de la interfaz para leer o escribir en el almacenamiento local.

El proceso de acceso y almacenamiento en archivos XML se realiza de manera automática a través de la interfaz. Durante la ejecución del juego, simplemente se requiere actualizar las instancias de las clases *GameSlot* y *OptionSlot* con los cambios pertinentes y, posteriormente, llamar a la función *Save* una vez que el jugador decida guardar su progreso o en situaciones de autoguardado.

La opción de guardado local a través de estas dos clases estáticas resulta especialmente conveniente cuando trabajamos con múltiples escenas. Esto se debe a que no solo permite a los jugadores conservar su progreso, sino que también nos brinda a los programadores un acceso constante a la información más actualizada del juego, sin importar en qué escena nos encontramos.

Código 6.1 – Archivo de guardado de opciones XML

```

1  <?xml version="1.0"?>
2  <SaveSlot xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="OptionsSlot">
4    <name>GameSettings</name>
5    <mute>false</mute>
6    <masterVolume>0.60004</masterVolume>
7    <musicVolume>0.0001</musicVolume>
8    <sfxVolume>1</sfxVolume>
9    <vSync>true</vSync>
10   <fullscreen>true</fullscreen>
11   <resolution>1920x1080</resolution>
12   <quality>2</quality>
13   <rumble>false</rumble>
14 </SaveSlot>
```

Como estamos desarrollando el proyecto en Windows, el guardado de los archivos se realiza en “C:\Users \<user>\AppData\LocalLow\<company name>” [68]. Si queremos consultar que todo se está almacenando correctamente podemos ir a la ubicación y comprobarlo, la estructura de los archivos será como la de la Código 6.1.

6.4.1.1. Guardado de Inputs

Durante el desarrollo, nos percatamos de la necesidad de almacenar la información relacionada con la reasignación de *inputs* realizada por el usuario. Esta información es crucial para que, en ocasiones posteriores de juego, el usuario pueda cargar su mapeado de controles personalizado junto con las demás opciones configuradas. Para abordar esta necesidad, optamos por emplear una función que nos permite guardar y cargar todas las acciones asociadas a un *PlayerInput* en formato JSON [106]. El resultado de esta función es una cadena de caracteres en formato JSON que almacenamos como una variable de tipo *string* en nuestra clase *OptionSlot*. Esta integración se adapta perfectamente a nuestro sistema de almacenamiento en formato XML. La estructura resultante de esta cadena *string* se asemeja a la que se define en la Código 6.2.

Código 6.2 – Guardado de *inputs* mediante JSON

```

1  {"bindings": [
2    {"action": "Main Movement/Action1", "id": "2277528b-a95c-4d01-bfc0-bbb5a15b965e",
3     "path": "<Gamepad>/rightShoulder", "interactions": "", "processors": ""},
4
5    {"action": "Main Movement/Action0", "id": "037d21b1-ecc3-48d8-9143-93c8231d5156",
6     "path": "<Gamepad>/leftShoulder", "interactions": "", "processors": ""},
7
8    {"action": "Main Movement/Action0", "id": "cefe336d-d3bc-40d4-84ec-153d147f63c7",
9     "path": "<Keyboard>/u", "interactions": "", "processors": ""}
10  ]}
```

6.4.2. Aplicar los cambios de opciones

Consideramos pertinente explicar en esta sección, ya que está estrechamente relacionada con el guardado, cómo implementamos la aplicación de cambios realizados en las opciones de nuestro juego. Con “aplicar cambios” nos referimos a que cuando el usuario modifique la resolución o habilite la sincronización vertical en las opciones, estos cambios se reflejen de manera efectiva en el juego. Para lograrlo, hemos adoptado la creación de una interfaz que contenga los métodos necesarios para la aplicación de cambios (*IApplier*). Esta interfaz debe ser implementada por una clase que leerá la versión más actualizada de las opciones desde *OptionSlot* y aplicará los cambios utilizando las funcionalidades proporcionadas por Unity.

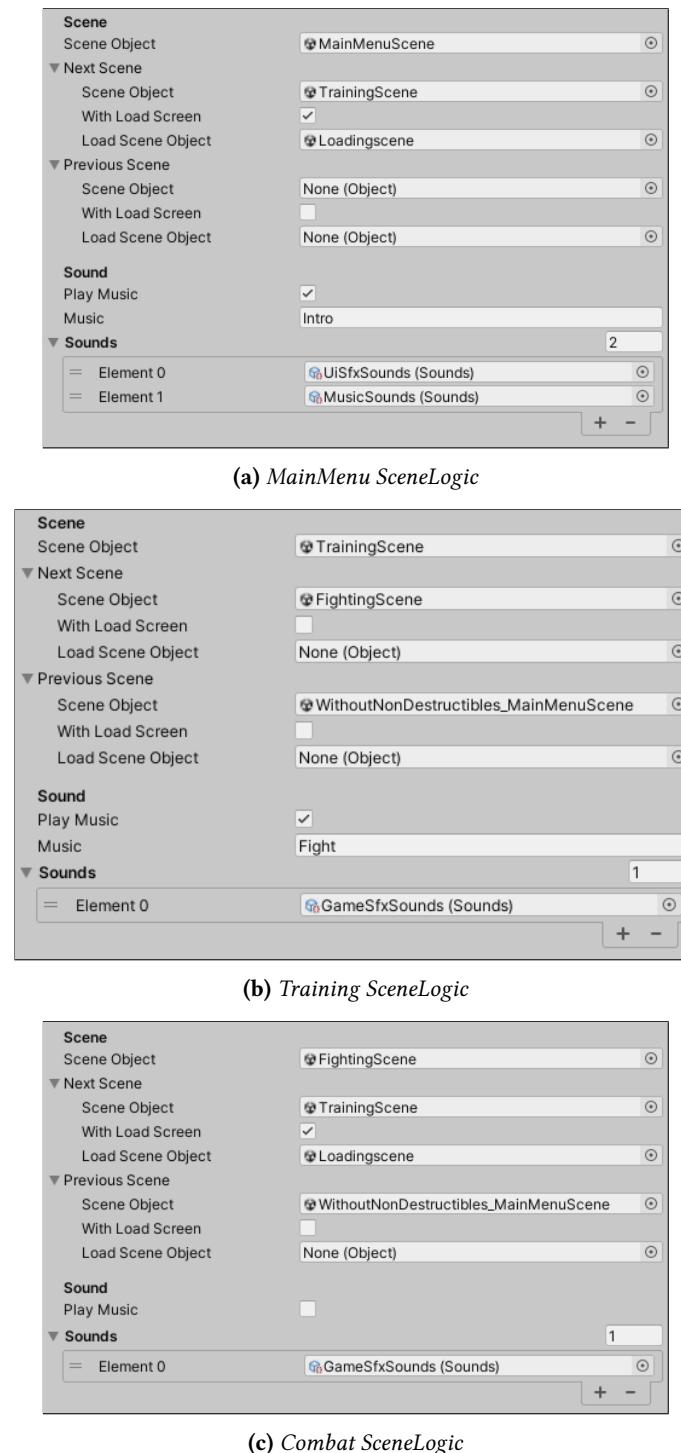
6

- Aplicar opciones de sonido: Para asegurar que los cambios se reflejen en el sonido, accedemos al componente *AudioMixer* [58], el cual contiene todos los grupos de sonido que pueden tener sus opciones alteradas.
- Aplicar cambios visuales: Para cambiar las opciones de calidad visual y la tasa de refresco, accedemos a la variable estática *vSyncCount* y la función estática *SetQualityLevel* de la clase *QualitySettings* [107]. También podemos utilizar la función estática *SetResolution* de la clase *Screen* [108] para ajustar la resolución.

Finalmente, en el menú de opciones, donde se realizan los cambios, se crea una instancia de la interfaz con el “aplicador” correspondiente. También es necesario aplicar los cambios al inicio de la ejecución en *GameManager*, después de cargar los datos de guardado. Es importante destacar que algunas opciones no requieren de esta función, por ejemplo, cuando se activa o desactiva la vibración del mando, esta verificación se realiza directamente en el script de *RumbleController*. Esta función se utiliza solo como un paso intermedio para acceder a las funcionalidades de Unity.

6.5. Gestión de escenas

Este sistema es fundamental en cualquier videojuego independientemente de su género ya que es muy difícil contener el videojuego entero en una única escena y sería contraproducente. Al tratarse de un juego de lucha, nuestra gestión de escenas no será muy extensa, ya que solo necesitamos movernos entre las distintas zonas de combate, el menú principal y la zona de entrenamiento. Sin embargo, a pesar de su relativa simplicidad, es crucial que esta gestión se lleve a cabo de manera adecuada.

Figura 6.3 – Definición de *SceneLogics*

6

6.5.1. Estructura y funcionamiento

En la fase actual del desarrollo, hemos definido tres escenas principales con la lógica estructural que se explicó en la implementación, denominada *SceneLogic*. En estas escenas, establecemos el orden de transición entre ellas, el tipo de transición (síncrona o asíncrona), y los *SoundStructures* que deben inicializarse en cada una. También hemos añadido un campo de tipo *string* que indica el nombre de la canción con la que inicia la escena para lograr una reproducción automática.

tica. Esta definición se muestra en la [Figura 6.3](#).

1. *MainMenu*: Esta es la escena inicial, que transiciona a la escena de entrenamiento a través de una pantalla de carga, utilizando una transición asíncrona.
2. *Training*: En esta escena, los jugadores pueden entrenar sus movimientos de combate. Tienen la opción de regresar al *MainMenu* o iniciar un combate cuando lo deseen.
3. *Combat*: En esta escena, los jugadores se enfrentan a sus oponentes. Si ganan, transicionan directamente a la escena de entrenamiento, donde obtienen recompensas y se preparan para el siguiente combate. Si pierden, vuelven a la escena del menú principal para comenzar una nueva partida. También tienen la opción de regresar al menú principal desde el menú de pausa.

Cabe señalar que, por el momento, la escena de combate permanece constante, ya que no hemos diseñado otras. Este ciclo se repite hasta que el jugador detenga la ejecución del juego. La inicialización de los sonidos también se define en la lógica de las escenas ([Figura 6.3](#)), junto con la música de fondo.

Como se mostró en la implementación, nuestro controlador de escenas accede a esta lógica a través de un [diccionario](#) que vincula el nombre de cada escena con su lógica correspondiente. Automáticamente, el controlador reproduce la escena siguiente o anterior, según lo definido en la lógica. Desde el *GameManager*, mi compañero solo necesita llamar a las funciones *NextScene* o *PreviousScene* sin necesidad de especificar detalles adicionales. Si en la lógica no se ha definido ninguna escena a la cual realizar una transición, esto está contemplado y no se ejecuta ninguna acción. Esta funcionalidad resulta útil, por ejemplo, en la primera escena donde no existe una escena previa a la cual transicionar.

Resulta evidente que abstraer la estructura de las escenas de su uso conlleva a un código más legible y evita la creación de variables superfluas. Consideramos que no hemos aprovechado completamente esta utilidad y que existen posibilidades de mejora. Por ejemplo, podríamos expandir la lógica de las escenas para incluir opciones de autoguardado, otras configuraciones de inicialización, y no solo los aspectos relacionados con el sonido. Asimismo, podemos explorar la posibilidad de indicar el modo de carga de las escenas [\[109\]](#).

6.5.2. Objetos permanentes entre escenas

No queremos concluir esta sección sin abordar algunos aspectos que podrían resultar relevantes. A lo largo de todo el proceso de implementación, hemos asumido que el objeto *GameManager* y todo lo relacionado con él constituyen una única instancia estática durante toda la ejecución del juego. Sin embargo, aún no hemos discutido cómo se gestiona esta instancia al cambiar de escenas. El hecho es que cualquier objeto instanciado en una escena, como el propio *GameManager*, se autodestruye cuando cambiamos a otra escena, a menos que carguemos la siguiente escena en modo aditivo [\[109\]](#), lo cual todavía no hemos contemplado. Por lo tanto, necesitamos una forma de asegurarnos de que este objeto permanezca disponible entre diferentes escenas, de manera que pueda ser accesible en cualquier momento durante la duración del juego. Afortunadamente, Unity ya previó esta necesidad y proporciona una función llamada *DontDes-*

`troyOnLoad` [110] que simplifica enormemente este proceso.

En nuestro juego, hemos creado un objeto llamado *SaveData* en la primera escena, y todos los objetos que necesitamos mantener entre escenas se convierten en descendientes de este objeto.

- *GameManager*: Indispensable para toda la funcionalidad del juego.
- *UI Input Module* y *Event System*: Estos *scripts* definen la funcionalidad de los inputs en la interfaz de usuario y la navegación en los menús, respectivamente. Dado que no cambian entre escenas, no tiene sentido definirlos repetidamente.
- *Transition*: Este objeto gestiona las animaciones de transición entre escenas. Su función es visualizar la transición de una escena a otra, y por lo tanto, no tiene sentido tener una instancia de este objeto en cada escena.
- *PermanentSounds*: Este objeto alberga los sonidos en 2D, como la música y los efectos de sonido de la interfaz de usuario. Dado que estos sonidos se utilizan en varias escenas, no es necesario reinicializarlos en cada ocasión, lo que ahorra tiempo y recursos.

Con la implementación de objetos permanentes, emerge una nueva problemática: si decidimos regresar a la primera escena, nos encontraremos con duplicados de los objetos. Esto se origina en los objetos que se instancian en la primera escena para ser conservados en *DontDestroyOnLoad*, junto con las copias de estos objetos que persisten entre escenas. Unity nos alertará sobre esta situación, y es probable que nuestro código no funcione de manera adecuada. La solución que consideramos más efectiva para abordar este inconveniente es generar una nueva escena, idéntica a la primera, pero exenta de los objetos que deben preservarse. Únicamente necesitamos asegurarnos de que la lógica de escenas dirija el regreso al menú principal a través de esta nueva escena, como se ilustra en la [Figura 6.3](#).

Otra alternativa para resolver este problema, que muchos juegos optan por seguir, es incorporar una escena de título. Esta escena mostraría exclusivamente el título de nuestro juego y quizás algún botón para continuar. En esta escena, podríamos cargar todos estos recursos. Dado que esta escena se despliega solamente al comienzo de la ejecución y no se retorna a ella posteriormente, sería una manera efectiva de superar el problema.

6.6. Controlador de cámara

Un sólido controlador de cámara resulta crucial en cualquier juego de combate, ya que la cámara desempeña un papel fundamental en la inmersión del jugador en cada golpe, movimiento y esquiva que lleva a cabo. Los impactos que el personaje reciba se sentirán más contundentes, y los golpes ejecutados serán aún más satisfactorios. En términos generales, todo juego de lucha requiere, como mínimo, una cámara estática que siga tanto al personaje principal como al enemigo. Esta disposición permite que los jugadores tengan siempre una percepción clara de la distancia que los separa y cómo enfrentar el combate en curso. Además, resulta esencial incorporar una variedad de efectos de cámara que intensifiquen la inmersión en movimientos específicos y otros aspectos relevantes del juego.

Dado que la cámara es un elemento intrínsecamente vinculado a la experiencia del juego, esta sección se desglosará en varias partes, donde analizaremos los diversos componentes de la cámara que hemos empleado a lo largo del juego. Abordaremos tanto su función dentro del juego como el uso de los *scripts* del controlador de cámara para su implementación.

Todas las figuras de los comportamientos de la cámara se han creado utilizando capturas de pantalla tomadas directamente desde la escena o el juego en Unity. Estas imágenes se han ajustado visualmente para mejorar la claridad y comprensión. En las representaciones, el jugador se muestra en color azul, mientras que el enemigo se representa en color rojo. La cámara se representa mediante un cuadrado de color negro.

6.6.1. Uso de *Cinemachine*

Como ya se explicó en la sección de implementación, estamos utilizando el complemento *Cinemachine* de Unity. En consecuencia, todas nuestras cámaras son instancias de *Cinemachine-VirtualCamera*, lo que nos permite definir los componentes característicos de cualquier cámara en un videojuego. Es importante destacar que un componente de *Cinemachine* no crea una nueva cámara por se, sino que configura su comportamiento y anima su posición para satisfacer los requisitos que le hemos establecido; en otras palabras, se asemeja más a un camarógrafo que a una cámara propiamente dicha [111]. De esta manera, disponemos de una amplia gama de parámetros que podemos ajustar en una cámara virtual para lograr el comportamiento deseado.

- **Follow** y **LookAt**. imprescindibles para que la cámara sepa qué objeto debe seguir y hacia qué dirección debe orientarse.
- **Body**. aquí especificamos el algoritmo que determina cómo la cámara virtual se desplaza en la escena [112].
- **Aim**: este parámetro nos permite definir cómo rota la cámara virtual en la escena [113].
- **Noise**: con este ajuste podemos simular la vibración de la cámara, agregando un toque de realismo a la experiencia visual [114].

Cada tipo de *Body*, *Aim* o *Noise* contiene sus propios parámetros que nos ayudan a establecer con total precisión la configuración de cámara que deseamos.

6.6.2. Cámara de combate

En Carnem Levare, contamos con una única cámara principal diseñada específicamente para los combates. Esta cámara incorpora una serie de detalles y elementos de inmersión que consideramos esencial mencionar para demostrar la versatilidad de nuestro controlador en la implementación de nuevas funcionalidades.

6.6.2.1. Inicialización de objetivos (Targeting)

Tanto en el caso del jugador como en el del enemigo, es necesario disponer de ciertos objetos que marquen las posiciones que la cámara debe seguir constantemente. Para llevar a cabo esta tarea, hemos encontrado que es beneficioso emplear un *script* personalizado que permita definir

estos objetivos. En nuestro proyecto, hemos creado un *script* llamado *CameraTargets*, que contiene una lista de tuplas. Cada tupla representa un objetivo principal y un objetivo alternativo para la cámara, concepto que exploraremos más adelante. De este modo, en cada entidad del juego, ya sea jugador o enemigo, sus objetivos son accesibles públicamente, generalmente mediante la búsqueda de etiquetas. El único requisito estructural es que los índices de esta lista coincidan con las definiciones de nuestro enumerador *CameraType*, lo cual simplifica su integración posterior. Esto significa que, por ejemplo, si la cámara principal es el primer elemento en *CameraType*, el primer objetivo en la lista debería ser el objetivo de la cámara principal.

6.6.2.2. Apuntado y seguimiento

Esto se refiere a la configuración de los parámetros *Follow* y *LookAt* de nuestra cámara virtual. En nuestro caso, debido a la naturaleza tridimensional del juego, hemos optado por una cámara en tercera persona que se coloca detrás del jugador, asegurando al mismo tiempo que siempre tenga en cuenta la posición del enemigo. Esto nos lleva a la cuestión de cómo conocer en todo momento las posiciones del jugador y del enemigo en la pantalla. Dado que solo hay un enemigo en pantalla en todo momento, no es necesario complicarnos con una función de fijado, ya que no hay un cambio de objetivo.

Para mantener un seguimiento constante de las posiciones del jugador y el enemigo, recuperamos sus respectivos *CameraTargets* al inicio de nuestro controlador de cámara (*CameraController*), utilizando el método de búsqueda por etiquetas que ya hemos empleado en otras ocasiones. Una vez que tenemos estos objetivos, los asignamos al *script* de nuestra cámara principal para que esta los siga y los enfoque.

6.6.2.2.1. Problema de seguimiento En principio, podríamos pensar que simplemente tenemos que establecer el objeto obtenido del *CameraTargets* del jugador como el punto al cual la cámara debe apuntar. Sin embargo, esto puede dar lugar a un problema potencial en nuestro juego. Si establecemos la posición del jugador como el punto al cual la cámara apunta, obtendremos un efecto en el cual la cámara sigue al jugador incluso si este gira sobre sí mismo, ya que estamos anclando la cámara directamente al vector tridimensional de la posición del jugador. En caso de que el jugador cambie su dirección, la cámara seguirá esa rotación, lo que implica siempre observar la espalda del jugador. Aunque este enfoque puede ser adecuado para muchas cámaras en tercera persona en juegos de disparos u otros géneros (*Resident Evil 2 Remake* [115]), no es lo que buscamos en *Carnem Levare*. Nuestra intención es que, sin importar la dirección en que el jugador esté mirando, la cámara siempre se mantenga centrada en el enemigo.

Para abordar esta cuestión, hemos implementado una solución en la que un objeto sigue constantemente la posición del objetivo al que la cámara debe seguir, pero sin replicar sus transformaciones de cambio de dirección. En resumen, la cámara siempre se orienta hacia un objeto que sigue la posición del jugador y mantiene la mirada fija en el enemigo. Esto implica que estamos estableciendo una línea recta de dirección entre la cámara, el jugador y el enemigo. Esta configuración se mantiene constante, independientemente de la dirección en la que el jugador esté mirando localmente ([Figura 6.4](#)).

6.6.2.2.2. Apuntado mediante *CinemachineTargetGroup* Mediante un *TargetGroup*, hemos logrado que nuestra cámara enfoque un punto intermedio entre el jugador y el enemigo. Esto

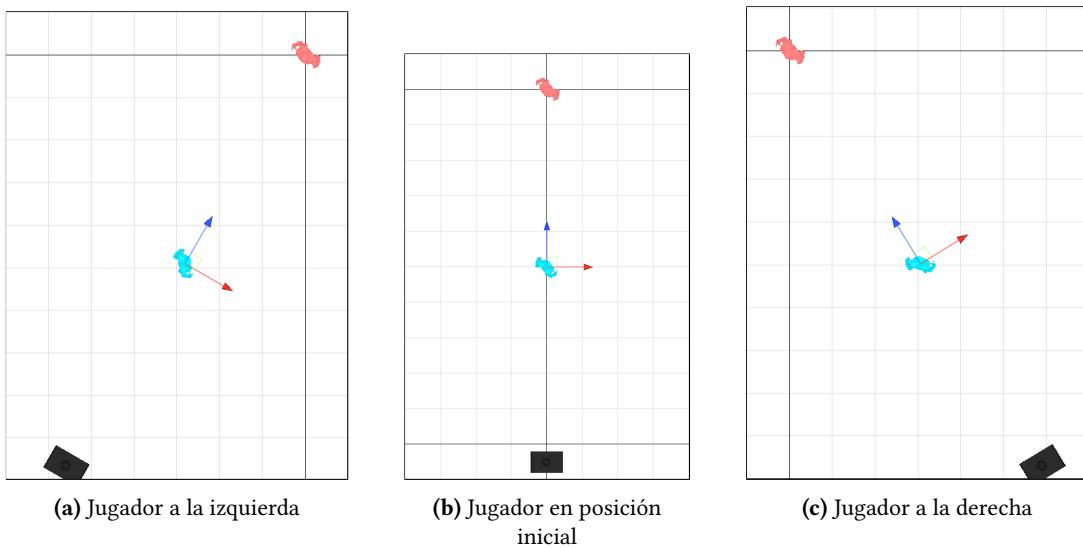


Figura 6.4 – Objeto de seguimiento de jugador

6

garantiza que ambos estén siempre dentro del encuadre de la cámara, de manera que la cámara siga al jugador mientras mantiene el punto medio entre ambos en su mira. En términos análogos, podría compararse a cómo una persona sigue a alguien que camina, pero su cabeza mira en una dirección diferente. Para lograr esto, hemos creado una instancia de *CinemachineTargetGroup* y la pasamos desde nuestro controlador al *script* de la cámara. De esta forma, se inicializa con los objetivos adecuados. En este caso, la dirección exacta del vector en el *TargetGroup* resulta irrelevante, ya que solo necesitamos un punto en el espacio hacia el cual dirigir la mirada de nuestra cámara virtual ([Figura 6.5](#)).

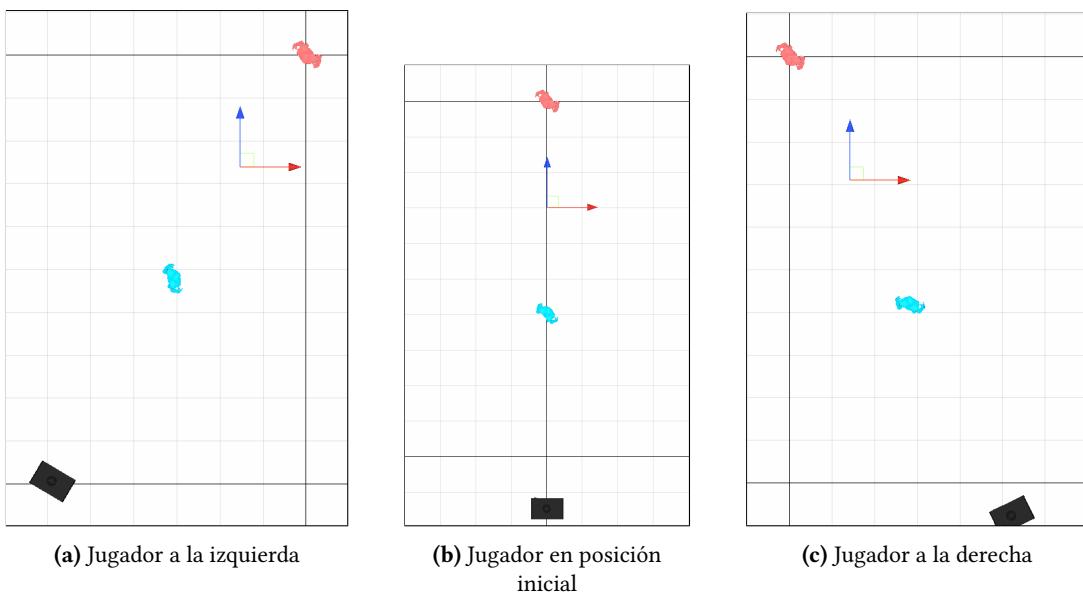


Figura 6.5 – Posición de *CinemachineTargetGroup*

6.6.2.3. Orbital Transposer

Para poder llevar a cabo la funcionalidad de la cámara deseada en Carnem Levare, hemos optado por emplear un algoritmo de desplazamiento (*Body*) denominado *Orbital Transposer*. Si

bien no es necesario entrar en todos los detalles de su funcionamiento, su documentación está disponible para su consulta [116], lo hemos seleccionado debido a que nos brinda la capacidad de establecer un círculo de órbita alrededor de nuestro objetivo a seguir ([Figura 6.6](#)).

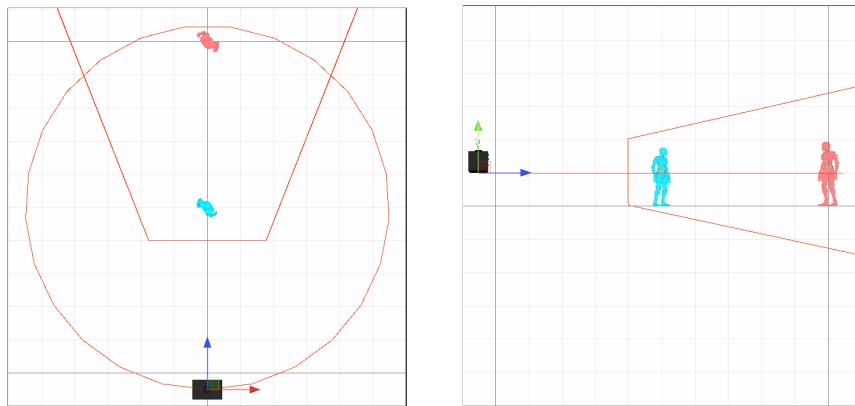


Figura 6.6 – Orbital Transposer

6

Mediante esta técnica, podemos mover la cámara hacia la izquierda y derecha del jugador sin perderlo de vista en ningún momento, y luego realinearla según sea necesario. Nuestro enfoque visual no persigue que puedas girar completamente en torno al jugador, sino que busca un efecto más sutil. Queremos que si te desplazas a la izquierda o derecha, el enemigo permanezca siempre en el campo de visión. Por lo tanto, si el jugador se mueve a la izquierda, debemos orillar la cámara hacia la derecha, y viceversa ([Figura 6.7](#)).

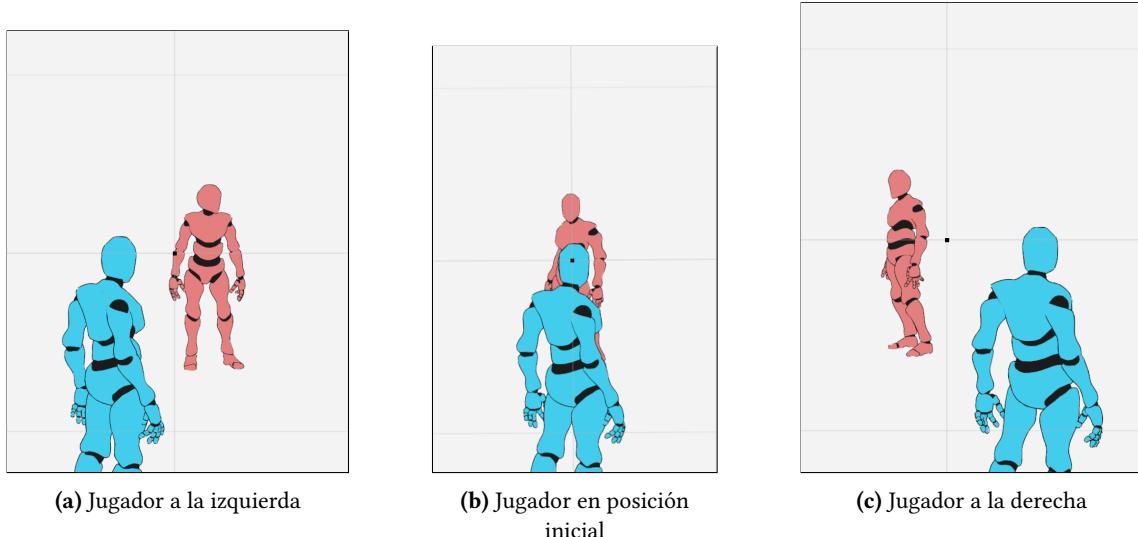


Figura 6.7 – Uso de Orbital Transposer para encuadre de jugador y enemigo

Dentro de este algoritmo de desplazamiento (*Body*), también debemos definir otros parámetros, como el *offset*. Por defecto, la cámara seguirá al objetivo de manera que se sitúe en la misma posición, es decir, dentro del propio jugador. No obstante, para lograr una visualización adecuada, necesitamos especificar a qué distancia debe mantenerse con respecto al objetivo en los diferentes ejes del espacio. En la [Figura 6.7](#), podemos observar un *offset* positivo en el eje Z y en el eje Y.

6.6.2.4. Efectos de cámara

Ahora abordaremos los efectos de cámara que hemos añadido para activarse en respuesta a ciertos eventos del juego. Hasta este punto, toda la funcionalidad de la cámara estaba integrada en el complemento *Cinemachine* y en el *script* de nuestra cámara de combate, donde inicializamos los componentes y gestionamos el *Orbital Transposer*.

Para componentes más específicos, hemos creado la clase abstracta *CameraEffect* y la hemos empleado para desarrollar seis efectos de cámara. Todos estos efectos se activan de manera reactiva, tal como se explicó en su funcionamiento. Esto significa que asignamos funciones a los eventos del comportamiento del jugador para que ejecuten los efectos de cámara. Para lograr la interpolación de valores a lo largo del ciclo de juego, utilizamos funciones *Lerp* [117].

6.6.2.4.1. Alineación de cámara orbital El objetivo de este efecto es lograr una interpolación del valor del movimiento orbital hacia cero, de manera que la cámara se ubique en una posición completamente centrada. Esta transición se lleva a cabo cuando el jugador eleva su guardia, lo que proporciona una sensación de cobertura y aumenta el impacto de las esquivas en guardia.

6.6.2.4.2. Movimiento Lineal Este efecto consiste en aplicar una interpolación entre el *offset* (*Vector3*) actual de desplazamiento de la cámara y un nuevo *offset* proporcionado como parámetro. Asignamos este comportamiento a los eventos asociados con levantar y bajar la guardia del jugador. De esta forma, cuando el jugador levanta la guardia, además de centrar la cámara orbital, se genera un desplazamiento lineal suave entre los dos *offsets*.

Hemos introducido un tiempo de respuesta para evitar movimientos continuos si el jugador pulsa repetidamente el botón de la guardia. En nuestro caso, hemos aplicado este efecto específico para lograr un ligero acercamiento al jugador en el eje Z, creando así un efecto de zoom cuando el jugador levanta la guardia, añadiendo intensidad a la acción. Este movimiento se podría cambiar simplemente alterando el parámetro de *offset* proporcionado.

Es posible lograr movimientos que involucren más de dos puntos si obtenemos varios *offsets* como parámetros y empleamos Curvas de Bézier cuadráticas o cúbicas [118] (Figura 6.8). Esto nos permite generar movimientos mucho más complejos, permitiendo la creación de trayectorias curvilíneas y reactivas para la cámara.

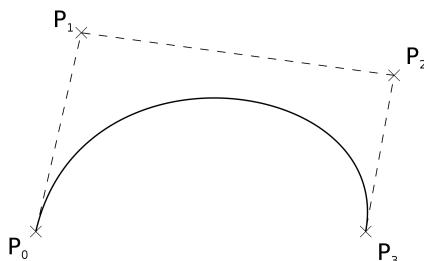


Figura 6.8 – Curva cónica de Bézier - Obtenido de [118]

6.6.2.4.3. Suavizado de seguimiento Este efecto implica la modificación del parámetro *Yaw Damping* [116] del desplazamiento, llevándolo de un valor a otro. Cuando el jugador no se encuentra en guardia, la rotación de la cámara es menos sensible, lo que otorga al jugador mayor libertad de movimiento. En contraste, cuando el jugador se encuentra en guardia, la rotación se vuelve más ágil, creando la impresión de un mayor control sobre las acciones.

6.6.2.4.4. Ruido de cámara El ruido de cámara consiste en añadir movimientos suaves y aleatorios, dando la sensación de que alguien está sujetando la cámara. Esta técnica puede aumentar el realismo en ciertas situaciones. Para lograrlo, intervenimos en el componente *Noise* de *Cinemachine* en nuestro efecto. De esta forma, cuando estamos en guardia, se genera un poco de ruido en la cámara, mientras que en el estado no guardia, se evita cualquier tipo de ruido.

6.6.2.4.5. Vibración de cámara (Camera Shake) La vibración de cámara es uno de los efectos más recurrentes en los videojuegos de lucha. Consiste en aumentar el ruido aparente de la cámara durante breves períodos, como al golpear o recibir un golpe. Esta técnica intensifica la sensación de impacto en los movimientos del jugador y enemigo. Cuando se efectúa un golpe o se recibe uno, la cámara vibra durante unos segundos, dando al espectador la impresión de haber experimentado la fuerza del impacto. Si se quiere conocer más acerca de este efecto se puede consultar este vídeo de Masahiro Sakurai, creador de Super Smash Bros [119]

Para lograr este efecto de manera precisa, los parámetros de vibración son proporcionados por el propio jugador, ya que cada movimiento deberá tener sus propios ajustes, considerando que hay golpes de mayor o menor intensidad. En consecuencia, suscribo las funciones de vibración de cámara con los eventos de golpe y de recibir daño del jugador. Cuando se activan estos eventos, también es necesario extraer de las propiedades del jugador los parámetros de vibración específicos del movimiento en curso.

6.6.2.4.6. Cambio de target Este efecto implica cambiar los objetivos que la cámara sigue en el *CinemachineTargetGroup*. Aunque este ajuste puede tener diversas aplicaciones, en nuestro caso lo utilizamos para incrementar la fuerza percibida de los golpes, permitiendo que la cámara siga ligeramente la cabeza del jugador cuando recibe un golpe. Por ello establecimos un *target* alternativo en el script *CameraTargets*, brindandonos la capacidad de alternar entre los dos objetivos durante el efecto.

6.6.2.4.7. Hitstop El efecto de *Hitstop*, ampliamente utilizado en juegos de lucha al igual que *Camera Shake*, consiste en detener momentáneamente el juego y generar una leve vibración en el personaje que ha sido golpeado. Esto crea la ilusión de un impacto contundente y acentúa la intensidad de la colisión. Para obtener más información, se pueden consultar los videos de Masahiro Sakurai [120] [121].

Aunque el efecto en sí no está a cargo del diseñador de la cámara, sí afecta al comportamiento de la misma. Si el personaje experimenta vibración debido al *HitStop*, la cámara también vibrará porque lo está siguiendo. Sin embargo, nuestro objetivo es separar la vibración de la cámara durante los golpes (*Camera Shake*) de la vibración del personaje. Para lograr esto, hemos creado

un “efecto de cámara” que desactiva el seguimiento y el apuntado de la cámara durante el breve período de duración del *HitStop*.

6.6.3. Cámara frontal

Además de la cámara de combate decidimos crear otra cámara que nos permita visualizar la parte frontal de nuestro jugador. Esta cámara solo se puede usar durante la escena de entrenamiento y permite visualizar mejor los movimientos que se asigne el usuario ([Figura 6.9](#)).

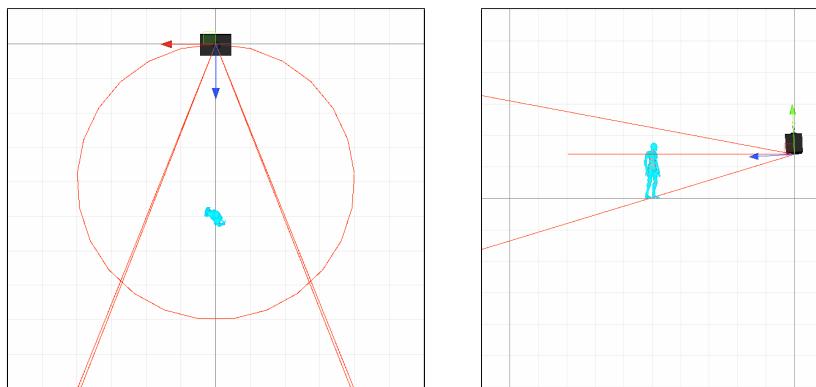


Figura 6.9 – Cámara frontal

6.6.4. Conclusiones y posibles mejoras

Durante el desarrollo de la cámara, nos hemos encontrado con ciertos aspectos que podrían ser mejorados para futuras implementaciones. Aquí se presentan algunas de estas consideraciones:

1. Los efectos de cámara podrían beneficiarse de una implementación por defecto del tiempo de respuesta. Actualmente, en la clase abstracta *CameraEffect*, hemos introducido la variable de tiempo de respuesta, pero no existe una implementación predeterminada para su aplicación. En consecuencia, esta funcionalidad debe ser especificada en cada *script* de efecto que creemos. Sería conveniente definir un comportamiento por defecto para el tiempo de respuesta en la clase base de efectos.
2. Para evitar la necesidad de inyectar dependencias de jugador y enemigo en cada uno de los efectos, sería recomendable abstraer los eventos a los que las funciones de los efectos se suscriben. Actualmente, si deseamos cambiar un evento, esta estructura genera varios puntos de modificación en el código. Una solución viable podría ser emplear un manejador de eventos basado en *ScriptableObject*, similar a cómo lo implementamos en el sistema de *inputs* ([Figura 4.7](#)). De esta manera, los eventos podrían ser invocados por el jugador y el enemigo, mientras que los efectos de cámara se suscriben a estos eventos. Al ser tanto los efectos de cámara como los eventos *ScriptableObject*, se establecería una mejor compatibilidad, permitiendo especificar los eventos como parámetros y facilitando futuras modificaciones en la ejecución de los efectos.
3. Si bien la inyección de dependencias desde el controlador hacia las diferentes cámaras es una práctica beneficiosa, presenta ciertas complicaciones. Por ejemplo, si existen varias cámaras y cada una depende de distintos objetivos, se gestionaría un número excesivo

de variables de objetivos en el código. En lugar de manejar múltiples parámetros para la inicialización de las cámaras, podríamos simplificarlo al abstraerlos en dos variables: *follow* y *lookAt*. Ambas variables serán simplemente un punto en el espacio y contienen toda la información que requiere la cámara para iniciarse. Para manejar este proceso, podríamos mantener una lista de estas variables, en correspondencia con el orden del enumerador *CameraType*, de manera similar a cómo lo implementamos en el *targeting*.

6.7. Sistema de interfaz de usuario

La interfaz de usuario, como se ha mencionado en otras ocasiones, no necesita una introducción exhaustiva, ya que es ampliamente conocida en el ámbito de los videojuegos. Una interfaz de usuario sólida es esencial en cualquier juego, ya que permite a los usuarios realizar una variedad de acciones, como modificar las opciones del juego, pausarlo, guardar su progreso o acceder a las opciones específicas del juego en sí. En nuestro caso, la interfaz de usuario también se utiliza para permitir a los jugadores cambiar y asignar movimientos a su personaje de manera personalizada.

6

Las imágenes que representan el diseño de las interfaces no son totalmente precisas en términos de su apariencia real en el juego. Se optó por aplicar un efecto de negativo a las imágenes para mejorar su visualización en el documento y resaltar la navegabilidad entre los distintos componentes interactivos.

6.7.1. Menú principal

Al iniciar el juego, el punto de partida es el menú principal, el cual debe ofrecer las opciones necesarias para comenzar a jugar, aplicar las configuraciones del juego, cargar una partida existente y salir del juego. En la versión actual de la implementación, no se incluye una función de guardado de partida, ya que el enfoque se centra en una demostración de estilo arcade. A pesar de esto, el resto de opciones se encuentra disponible.

En la construcción del menú principal, seguimos la misma lógica de implementación presentada en el [Capítulo 4](#). Esto implica explicar la creación del menú principal mediante dos estructuras de Modelo-Vista-Controlador (MVC) [78]. Una serie de controladores que heredan de *AbstractMenu* para crear la funcionalidad de cada menú por separado y un controlador para la navegabilidad entre los menús que usa el Framework desarrollado ([Capítulo 5](#)) como modelo.

6.7.1.1. Diseño

El diseño del menú es independiente de la funcionalidad que se implementará posteriormente. Todo este proceso se lleva a cabo utilizando el sistema de interfaz de usuario (UI) de Unity [75]. Este enfoque está diseñado para ser ejecutado por un diseñador con experiencia en otras plataformas de diseño de interfaces. A continuación, se explorará el diseño que hemos elegido para comprender cómo se conecta con la funcionalidad que se desarrollará más adelante.

Nuestra visión es que el menú principal sea simple pero efectivo, al menos en esta etapa de desarrollo. Está compuesto por tres botones principales: uno para iniciar la partida, otro para

acceder al menú de opciones y un tercero para salir del juego ([Figura 6.10](#)).



Figura 6.10 – Menú de inicio de partida

Dentro del menú de opciones, se presenta otro menú de tipo seleccionable. Aquí se utilizan dos teclas del teclado o dos botones del mando para alternar entre los menús asociados. Los menús seleccionables son muy prácticos ya que indican en todo momento en qué sección se encuentra el usuario y le permiten interactuar directamente con ella. Si optáramos por la navegación estándar de menús, tendríamos que regresar a un menú anterior cada vez que quisiéramos cambiar entre las distintas secciones de configuración.

Actualmente, las opciones disponibles incluyen: Audio, Controles, Opciones Visuales y Accesibilidad ([Figura 6.11](#)). Aunque la funcionalidad de esta última aún no ha sido implementada, está planificada para futuras actualizaciones.

6.7.1.2. Navegabilidad

Para establecer la lógica que rige la navegación entre los menús, emplearemos nuestro framework. Dado que estamos tratando con un menú bastante simple, su estructura lógica también lo es. En la [Figura 6.12](#) se muestra el árbol de navegación resultante. Podemos observar que consta únicamente de un nodo raíz y un nodo estándar con una sola transición (botón *options*, [Figura 6.10](#)) hacia un nodo seleccionable. Este último nodo seleccionable se desglosa en 4 hijos, cada uno representando uno de los menús de configuración: audio, controles, opciones visuales y accesibilidad.

Desde nuestro controlador (*MenuController*), se realiza la navegación a través de los menús, accediendo a las funciones del árbol y obteniendo los índices de los menús que deben desplegarse. Si se desea acceder a un menú específico dentro de un nodo seleccionable en lugar de moverse a la izquierda o derecha, se puede lograr proporcionando el ID del menú. Esta opción ha sido especialmente útil para el control del menú con el ratón, que permite hacer clic en los distintos menús para cambiar entre ellos.

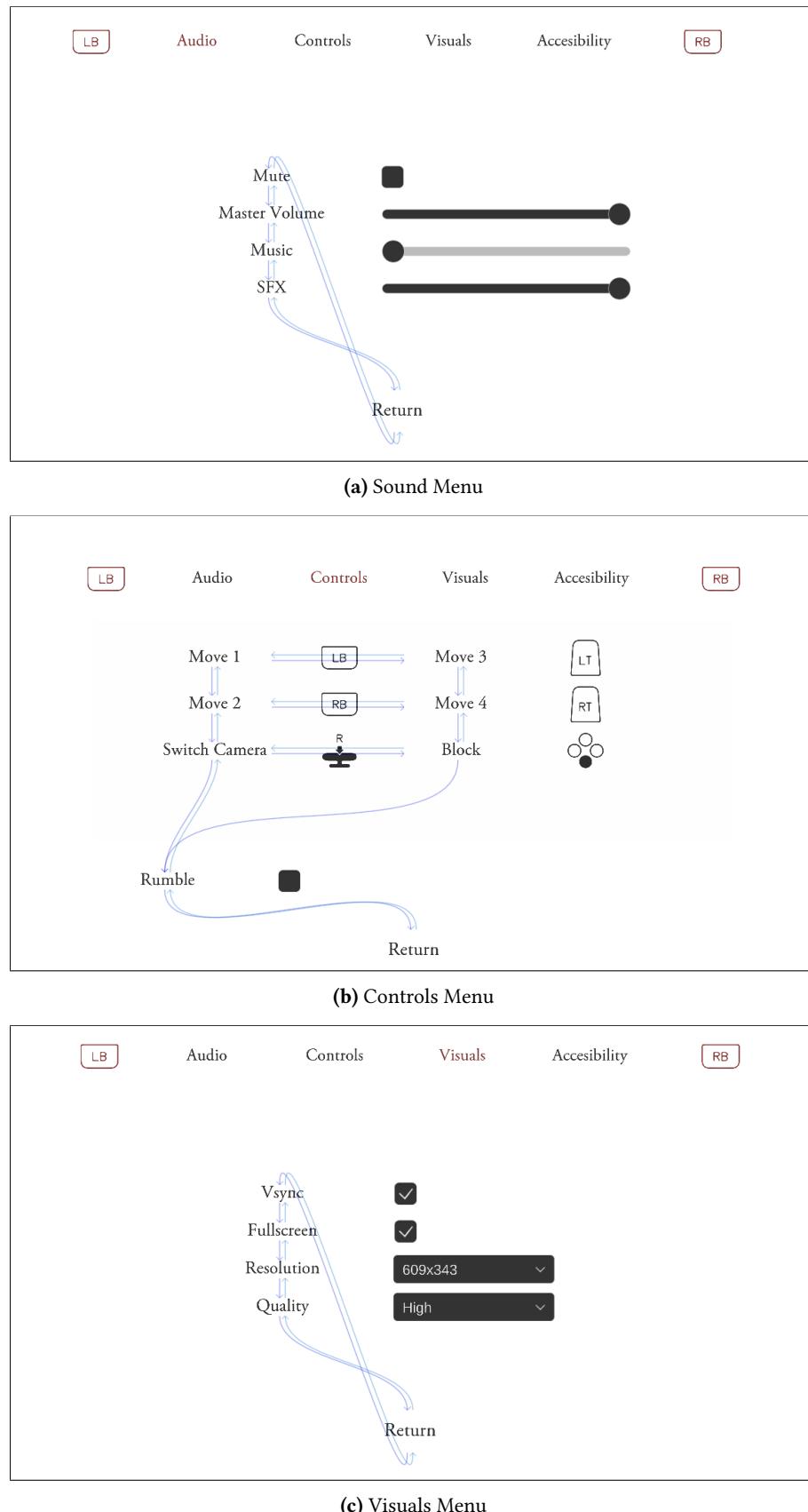


Figura 6.11 – Menú seleccionable para configuración

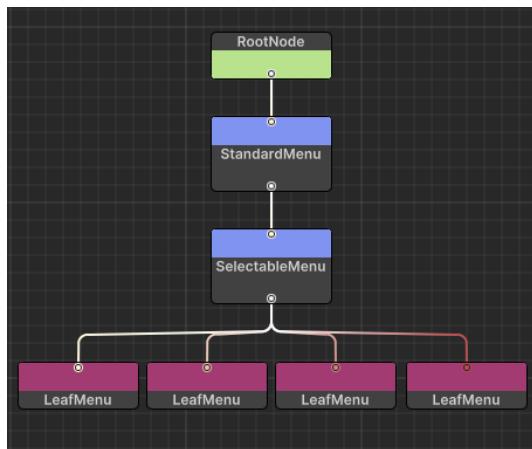


Figura 6.12 – Árbol lógico de menú principal

6.7.1.3. Asignación de funcionalidades

6 Este apartado nos explica los distintos controladores de menús que hemos creado para que todos los botones y utilidades que definen nuestro menú puedan ser interactivos, es decir, sus eventos estén suscritos a las funciones adecuadas. En nuestro caso se han creado cinco menús que heredan de *AbstractMenu* para suplir todas las necesidades.

En cada controlador de menú se accede tanto a los elementos de UI como a las funcionalidades que deben suplir y se suscriben esas funciones a los eventos de los objetos UI. Se va describir cada elemento y la funcionalidad que se suscribe para una mejor comprensión de todo lo que hace el menú.

6.7.1.3.1. Initial Menu

- Botón *Play* → Accede al gestor de escenas a través del *GameManager* para cambiar a la siguiente escena.
- Botón *Options* → Accede al *MenuController* para avanzar al siguiente menú en el árbol lógico.
- Botón *Exit* → Ejecuta la función *Application.Quit()* para cerrar la aplicación.

6.7.1.3.2. Selectable Options Menu Este menú posee una función para recorrer todos los menús asociados y conectarlos con las elecciones de menú en el árbol lógico, habilitando la interacción con el ratón.

- Botones de menú asociados → Acceden al *MenuController* para verificar en el árbol lógico qué menú está actualmente seleccionado y lo resaltan en color.

6.7.1.3.3. Sound Menu

- Botón *Mute* → Utiliza la función *Toggle* heredada de *AbstractMenu*, que guarda el valor en *DataSaver* y aplica los cambios mediante la interfaz *IApplier*.
- Botones *MasterVolume*, *Music* y *SFX* → Cada uno crea una transición con el *Slider* correspondiente.

- *Sliders* → Utilizan la función *Slider* heredada de *AbstractMenu*, que guarda el valor en *DataSaver* y aplica los cambios mediante la interfaz *IApplier*.
- Botón *Return* → Accede al *MenuController* para regresar al menú principal del que se proviene.

6.7.1.3.4. Controls Menu

- Botón *Rumble* → Utiliza la función *Toggle* heredada de *AbstractMenu*, que guarda el valor en *DataSaver* y aplica los cambios mediante la interfaz *IApplier*.
- Botones de remapeo de controles → Acceden a una instancia de la clase *InputRemapping* que llama a la función *Remapping*, usando el parámetro del evento como identificador del *input* a remapear y pasando como parámetro un objeto que representa un *PopUpMenu*, necesario para que el usuario pueda realizar correctamente el remapeo.
- Botón *Return* → Accede al *MenuController* para regresar al menú principal del que se proviene.

6.7.1.3.5. Visuals Menu

- Botones *VSync* y *FullScreen* → Utilizan la función *Toggle* heredada de *AbstractMenu*, que guarda el valor en *DataSaver* y aplica los cambios mediante la interfaz *IApplier*.
- Botones *Resolution* y *Quality* → Cada uno crea una transición con el *DropDown* correspondiente.
- *DropDowns* → Utilizan la función *DropDown* heredada de *AbstractMenu*, que guarda el valor en *DataSaver* y aplica los cambios mediante la interfaz *IApplier*.
- Botón *Return* → Accede al *MenuController* para regresar al menú principal del que se proviene.

6.7.1.4. Asignación de inputs

Para que el menú tenga funcionalidades con teclado/ratón o mando, además de las que ya contempla el *UI Input Module*, debemos asignar las acciones que deseamos que ocurran al presionar ciertos *inputs* en nuestro *InputReader*. Esta asignación se llevará a cabo dentro del *MenuController*, dado que todas estas acciones están relacionadas con la navegabilidad en los menús. En nuestro caso, asignamos un botón del mando para retroceder al menú anterior, que corresponde simplemente al *GoToParent* en nuestro árbol de menús.

En esta misma funcionalidad, también debemos verificar si se ha realizado alguna transición entre objetos interactivos para deshacerla antes de regresar al menú anterior. Por ejemplo, usando el mismo botón del mando, podemos regresar de un *Slider* a su menú principal correspondiente. Con otros dos botones adicionales, podremos cambiar entre menús dentro del menú seleccionable. Esto significa asignar las funciones *MoveLeft* o *MoveRight* al *input*, lo que nos permitirá navegar entre elementos hermanos en la jerarquía de menús.

6.7.2. Menú de Pausa

El menú de pausa es esencialmente idéntico al menú principal, excepto por la adición de un controlador adicional para abarcar las funcionalidades típicas de un menú de pausa. Esto implica asignar un comando del mando para activar y desactivar el menú de pausa, así como la inicialización de los componentes pertinentes. Además, el menú de pausa debe contener dos botones esenciales: uno para reanudar el juego y otro para volver al menú principal, junto con el botón que accede a las opciones. Con el propósito de que otras utilidades puedan rastrear cuándo se ha iniciado o salido del menú de pausa, hemos establecido dos eventos estáticos que notifican estas acciones.

Al pausar un juego, se deben tener en cuenta ciertos aspectos que este controlador debe manejar para evitar cualquier problema:

- La reinicialización del árbol lógico de navegabilidad es esencial. Esto asegura que, aunque se haya cerrado el menú de pausa en un punto específico del árbol, éste se reconfigure y regrese al nodo raíz.
- Se debe detener el flujo del tiempo en el juego, lo cual se logra mediante el ajuste de la propiedad *Time.timeScale* [122].
- El *ActionMap* del *PlayerInput* debe cambiar al mapeo de acciones de la interfaz (UI) para permitir la navegación por el menú. Anteriormente, se utilizaba el mapeo de acciones del juego.
- Activar la animación que produce la transición visual desde el juego al menú de pausa.
- Detener cualquier reproducción de sonido, ya que los sonidos no forman parte del ciclo de juego y podrían continuar en segundo plano durante el menú de pausa.
- Llamar a los eventos que notifican la activación o desactivación del menú de pausa, de modo que otras partes del sistema puedan responder adecuadamente.

6.7.3. Conclusiones y posibles mejoras

El desarrollo del menú de Carnem Levare ha sido efectuado de manera exitosa gracias a la estructura que hemos implementado. Esta estructura nos ha permitido administrar cada menú de forma individualizada y asignar la funcionalidad deseada a los elementos de la interfaz. Creemos que este enfoque es fundamental debido a la variabilidad de cambios que pueden surgir en el uso de un menú. Aunque en situaciones previas hemos reconocido la utilidad de un manejador de eventos para coordinar acciones, en este contexto en particular, esta estrategia podría no ser tan adecuada, ya que requeriría que estableciéramos en una única estructura funcionalidades muy específicas de cada elemento interactuable. En cambio, la adopción de la estructura Modelo-Vista-Controlador para cada menú resulta altamente ventajosa, ya que nos permite definir con precisión cómo deseamos que funcionen los diferentes elementos del menú y realizar modificaciones según sea necesario.

En relación al *framework* de navegabilidad, consideramos que ha sido una idea excelente, ya que nos ha permitido abstraer la lógica que conecta a los distintos menús, evitando así la necesidad de seguir jerarquías específicas en la escena. No obstante, aún hay margen de mejora y ciertos aspectos que nos gustaría perfeccionar:

1. Sería deseable que el *framework* incorporara la gestión de *inputs*, lo que nos permitiría asignar directamente comandos para moverse de izquierda a derecha dentro de un nodo seleccionable en el árbol. Además, sería útil incluir otros comandos generales que puedan ser necesarios, como el de regresar al menú anterior.
2. Sería conveniente mejorar la capacidad del *framework* para crear menús más complejos. Nos gustaría poner a prueba el *framework* con varios ejemplos y abordar cualquier defecto o laguna que pueda presentar en la lógica.
3. Sería valioso integrar funciones de transición en el *framework*, lo que permitiría especificar en el árbol si se desea aplicar algún tipo de efecto visual, como un desvanecimiento, durante el cambio de menú.
4. Debido a la estructura interna del *framework*, es factible crear nuevos tipos de nodos que puedan servir para otras funciones, como un menú desplegable o un menú de información que se cierre automáticamente después de un tiempo determinado.

Conclusiones y trabajos futuros

Queremos concluir este proyecto recopilando en este capítulo todo lo que hemos logrado y compartiendo cómo planeamos avanzar en nuestro futuro como desarrolladores de videojuegos.

7.1. Conclusiones

Este trabajo representa un hito significativo en nuestra trayectoria como desarrolladores de videojuegos. Durante el proceso, hemos adquirido un profundo entendimiento del software, la lógica y la estructura subyacente en la creación de productos de esta naturaleza. Esta experiencia nos ha permitido sumergirnos en el mundo del desarrollo de videojuegos y ganar confianza en nuestras habilidades.

Hemos logrado alcanzar los objetivos que nos propusimos; con un sistema de *UI* que se beneficia del patrón de diseño MVC y del *framework* que hemos desarrollado; un sistema de cámara que, gracias a *Cinemachine* y nuestras abstracciones, nos permite crear una amplia variedad de efectos y configuraciones; un sistema de *inputs* que aprovecha al máximo el *InputSystem* para ofrecer la flexibilidad que necesitamos; un sistema de audio fácilmente manipulable por el diseñador y accesible mediante una tabla hash; un sistema de guardado que permite múltiples tipos de almacenamiento gracias al uso de interfaces; y un controlador de escenas que abstraer su lógica para que pueda ser definida por el diseñador. Todo esto se encuentra unificado mediante un software que, gracias a diversos patrones de diseño y al potencial de los *ScriptableObject*s de Unity, nos permite mantener todo independizado, accesible y modular.

7.2. Perspectivas Futuras

A lo largo de todo este proyecto, hemos discutido en los distintos capítulos mejoras y variantes que podríamos aplicar para continuar avanzando. Reconocemos que nuestro trabajo no es perfecto y necesita refinamiento, así como numerosas actualizaciones. Nuestra dirección futura implica la implementación de estas mejoras, así como ampliar gradualmente las capacidades de nuestro framework de menús para abarcar más opciones y mejorar su escalabilidad. Nuestra aspiración es llevarlo a un punto donde pueda ser incluso comercializado.

Del mismo modo, Carnem Levare es un proyecto que concebimos con la intención de lanzarlo comercialmente. Planeamos colaborar con profesionales del ámbito artístico para dotar al juego de un atractivo visual acorde con nuestra visión. Consideramos fundamental contar con la perspectiva de artistas para que nos indiquen sus necesidades y preferencias, y así nosotros, desde la perspectiva del desarrollo de software, podamos cumplirlas, lo que sin duda enriquecerá aún más nuestros sistemas.

Hemos aprendido que el desarrollo de un videojuego es una tarea compleja y multifacética. Aunque hemos abordado numerosos aspectos fundamentales en este trabajo, aún existen áreas pendientes, como la implementación de efectos visuales (VFX), como sistemas de partículas. Tam-

bién deseamos profundizar en la gestión de la música y explorar la integración de software como FMOD [16]. Pretendemos trabajar en opciones de accesibilidad más exhaustivas y considerar la implementación de sistemas de diálogo y narrativa para transmitir eficazmente la historia a nuestros jugadores.

Por último nuestra intención a nivel global es continuar adoptando la misma metodología de desarrollo que hemos aplicado en este proyecto, capitalizando todas las lecciones aprendidas en el proceso. Buscaremos abordar cada necesidad particular de nuestro videojuego, ya sea aplicando soluciones existentes o desarrollando nuevas soluciones cuando sea necesario. Con optimismo, esperamos que nuestra propuesta resuene con un público más amplio y tenga el potencial de tener éxito comercial, lo que a su vez nos permitiría continuar trabajando en futuros proyectos de desarrollo.

Bibliografía

- [1] Wikipedia. «AAA (industria del videojuego).» (2022), dirección: [https://es.wikipedia.org/wiki/AAA_\(industria_del_videojuego\)](https://es.wikipedia.org/wiki/AAA_(industria_del_videojuego)).
- [2] Unity Technologies, *Attributes*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/Attributes.html>.
- [3] Unity, *Order of execution for event functions*, <https://docs.unity3d.com/Manual/ExecutionOrder.html>, 2023.
- [4] La Red Martínez, David L. and Acosta, Julio César and Mata, Liliana E. and Bachmann, Noemí G. and Vallejos, Oscar 2012. «Aprendizaje combinado, aprendizaje electrónico centrado en el alumno y nuevas tecnologías.» (2012), dirección: <http://sedici.unlp.edu.ar/handle/10915/19306>.
- [5] Microsoft, «Multiplicidad de extremo de asociación,» *Microsoft Learn*, 2023. dirección: <https://learn.microsoft.com/es-es/dotnet/framework/data/adonet/association-end-multiplicity>.
- [6] Unity, *Prefabs*, <https://docs.unity3d.com/es/530/Manual/Prefabs.html>, 2023.
- [7] Unity Technologies, *StyleSheet*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.StyleSheet.html>.
- [8] C. Vaccaro, *The importance of computer games for software engineering and scientific research*, https://www.researchgate.net/publication/345896115_The_importance_of_computer_games_for_software_engineering_and_scientific_research.
- [9] Diagrams.net. «Diagrams.net.» (2023), dirección: <https://app.diagrams.net/>.
- [10] Wikipedia, *Pong*, <https://es.wikipedia.org/wiki/Pong>, 2023.
- [11] Leslie Berlin, «The Inside Story of Pong,» *Wired*, 2017. dirección: <https://www.wired.com/story/inside-story-of-pong-excerpt/>.
- [12] R. Mraz, *The history of adaptive music in video games*, <https://splice.com/blog/adaptive-music-video-games/>.
- [13] Jacksynth, *Super Mario Odyssey – Adaptive Audio Analysis*, https://www.youtube.com/watch?v=jQRq10IFw10&ab_channel=Jacksynth, 2022.
- [14] Aaron Souppouris. «'Super Mario Odyssey' is everything it needs to be and more.» (2019), dirección: <https://o.aolcdn.com/hss/storage/midas/d09fc10addeb9d21d98f168fe588ac6e/205800890/scuba.jpg>.
- [15] Simon Brunner. «Super Mario Odyssey.» (2017), dirección: https://www.daily-passions.com/wp-content/uploads/2017/11/Super_Mario_Odyssey_003.jpg.
- [16] FMOD, *Games*, <https://www.fmod.com/games>.
- [17] Playstation, *Project Leonardo para PlayStation 5*, <https://blog.es.playstation.com/2023/01/05/presentamos-project-leonardo-para-playstation-5-un-kit-de-mando-de-accesibilidad-altamente-personalizable/>.
- [18] Wikipedia, *Rumble pak*, https://en.wikipedia.org/wiki/Rumble_Pak, 2023.
- [19] Wikipedia, *Haptic technology*, https://en.wikipedia.org/wiki/Haptic_technology, 2023.
- [20] M. E. Gingerich, *God of War's One-Shot Camera Approach Explained*, <https://gamerant.com/god-of-war-one-shot-unbroken-camera-challenging-film-technique/>.
- [21] G. Vault, *The Cameras of Uncharted*, <https://www.gdcvault.com/play/1015514/The-Cameras-of-Uncharted>.

- [22] PlayStation, *The last of us Part II - Accessibility*, <https://www.playstation.com/en-us/games/the-last-of-us-part-ii/accessibility/>.
- [23] Tom Hall, «The Doom Bible,» 1992. dirección: <https://5years.doomworld.com/doombible/doombible.pdf>.
- [24] Scrum.org, *Scrum.org - The Home of Scrum*, <https://www.scrum.org/>.
- [25] Microsoft, *What is DevOps?* <https://learn.microsoft.com/en-us/devops/what-is-devops>.
- [26] KeepCoding Team, «5 Ventajas de DevOps para tus proyectos,» *keepcoding*, dirección: https://keepcoding.io/wp-content/uploads/2022/02/13429_ILL_DevOpsLoop-1-768x432.webp.
- [27] Wikipedia, *Diseño centrado en el usuario*, https://es.wikipedia.org/wiki/Dise%C3%B1o_centrado_en_el_usuario, 2020.
- [28] Discord. «Discord.» (2023), dirección: <https://discord.com/>.
- [29] GitHub. «GitHub.» (2023), dirección: <https://github.com/>.
- [30] Muhammad Ovais Ahmad and Jouni Markkula and Markku Oivo, «Kanban in Software Development: A Systematic Literature Review,» *ResearchGate*, 2013. dirección: https://www.researchgate.net/publication/260739586_Kanban_in_Software_Development_A_Systematic_Literature_Review.
- [31] Github, *Creating draft issues*. dirección: <https://docs.github.com/en/issues/planning-and-tracking-with-projects/managing-items-in-your-project/adding-items-to-your-project#creating-draft-issues>.
- [32] Github, *Converting draft issues to issues*, 2023. dirección: <https://docs.github.com/en/issues/planning-and-tracking-with-projects/managing-items-in-your-project/converting-draft-issues-to-issues>.
- [33] Valpadasu Hema and Sravanthi Thota and Sripada Naresh Kumar and Ch Padmaja and C Bala Rama Krishna and K. Mahender, «Scrum: An Effective Software Development Agile Tool,» *Researchgate*, 2020. dirección: https://www.researchgate.net/publication/347374023_Scrum_An_Effective_Software_Development_Agile_Tool.
- [34] Rens Kortmann, «Agile game development: Lessons learned from software engineering,» *ResearchGate*, 2009. dirección: https://www.researchgate.net/publication/228985244_Agile_game_development_Lessons_learned_from_software_engineering.
- [35] Donetonic, «Metodología Waterfall vs Metodología Agile,» *Donetonic*, dirección: <https://donetonic.com/es/metodologia-waterfall-vs-metodologia-agile/>.
- [36] GitHub, *GitHub Desktop*, 2023. dirección: <https://desktop.github.com/>.
- [37] GitHub, *Unity.gitignore*, 2023. dirección: <https://github.com/github/gitignore/blob/main/Unity.gitignore>.
- [38] Unity, *Creando y usando scripts*, <https://docs.unity3d.com/es/530/Manual/CreatingAndUsingScripts.html>, 2023.
- [39] Unity, *Unity User Manual 2022.3*, <https://docs.unity3d.com/Manual/index.html>, 2023.
- [40] Wikipedia, *Facade (patrón de diseño)*, [https://es.wikipedia.org/wiki/Facade_\(patr%C3%B3n_de_dise%C3%BDo\)](https://es.wikipedia.org/wiki/Facade_(patr%C3%B3n_de_dise%C3%BDo)).
- [41] D. Parra, *Patrones de diseño – Facade*, <https://thepowerups-learning.com/patrones-de-diseno-facade/>.
- [42] M. Á. Sánchez, *Segregación de Interfaces (ISP)*, <https://medium.com/all-you-need-is-clean-code/segregaci%C3%B3n-de-interfaces-isp-7eb5dec98e57>.
- [43] César Daniel Meneses Guevara, «Principio de Segregación de la Interfaz,» *Cedaniel200*, dirección: <https://cedaniel200.blogspot.com/2018/11/principio-de-segregacion-de-la-interfaz.html>.

- [44] Unity, *SerializeField*, <https://docs.unity3d.com/ScriptReference/SerializeField.html>, 2023.
- [45] Microsoft, *events*, <https://learn.microsoft.com/es-es/dotnet/standard/events/>.
- [46] R. Nystrom, *Observer*, <https://gameprogrammingpatterns.com/observer.html>.
- [47] Unity, «How to Create Modular and Maintainable Code: Observer Pattern», *Unity*, 2023. dirección: <https://unity.com/how-to/create-modular-and-maintainable-code-observer-pattern>.
- [48] Unity, *Input System*, <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6/manual/index.html>, 2023.
- [49] Unity, *FindGameObjectsWithTag*, <https://docs.unity3d.com/ScriptReference/GameObject.FindGameObjectsWithTag.html>, 2023.
- [50] R. Nystrom, *singleton*, <https://gameprogrammingpatterns.com.singleton.html>.
- [51] Unity, *ScriptableObject*, <https://docs.unity3d.com/Manual/class-ScriptableObject.html>, 2023.
- [52] Microsoft, *dictionary*, <https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.dictionary-2?view=net-7.0>.
- [53] Unity, *Input Bindings*, <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.4/manual/ActionBindings.html>, 2023.
- [54] Unity, *PerformInteractiveRebinding*, https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.InputActionRebindingExtensions.html#UnityEngine_InputSystem_InputActionRebindingExtensions_PerformInteractiveRebinding_UnityEngine_InputSystem_InputAction_System_Int32, 2023.
- [55] Microsoft, *asynchronous-programming*, <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>.
- [56] J. French, *Async in Unity (better or worse than coroutines?)*, <https://gamedevbeginner.com/async-in-unity/>.
- [57] Unity, *AudioSource*, [https://docs.unity3d.com/ScriptReference/, 2023.](https://docs.unity3d.com/ScriptReference/<AudioSource.html)
- [58] Unity, *AudioMixerGroup*, <https://docs.unity3d.com/ScriptReference/Audio.AudioMixerGroup.html>, 2023.
- [59] Unity, *FindGameobjectsWithTag*, <https://docs.unity3d.com/ScriptReference/GameObject.FindGameObjectsWithTag.html>, 2023.
- [60] V. Bodurov, *IDictionary Options - Performance Test - SortedList vs. SortedDictionary vs. Dictionary vs. Hashtable*, <http://blog.bodurov.com/Performance-SortedList-SortedDictionary-Dictionary-Hashtable/>.
- [61] Microsoft, *Performance recommendations for Unity*, <https://learn.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-recommendations-for-unity?tabs=openxr>.
- [62] Microsoft, *ICloneable Interfaz*, <https://learn.microsoft.com/es-es/dotnet/api/system.icloneable?view=net-7.0>.
- [63] Wikipedia, *Principio de inversión de la dependencia*, https://es.wikipedia.org/wiki/Principio_de_inversi%C3%B3n_de_la_dependencia.
- [64] Microsoft, *Conversiones de tipos (Guía de programación de C#)*, <https://learn.microsoft.com/es-es/dotnet/csharp/programming-guide/types/casting-and-type-conversions#implicit-conversions>.
- [65] A. Šimec y Magličić, *Comparison of JSON and XML Data Formats*, https://www.researchgate.net/publication/329707959_Comparison_of_JSON_and_XML_Data_Formats, sep. de 2014.
- [66] Microsoft, *FileStream Clase*, <https://learn.microsoft.com/es-es/dotnet/api/system.io.filestream?view=net-7.0>, 2023.

- [67] Microsoft, *Uso de Visual C# para serializar un objeto en XML*, <https://learn.microsoft.com/es-es/troubleshoot/developer/visualstudio/csharp/language-compilers/serialize-object-xml>, 2023.
- [68] Unity, *Application.persistentDataPath*, <https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html>, 2023.
- [69] Unity, *SceneManager*, <https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html>, 2023.
- [70] Unity, *Cinemachine*, <https://unity.com/es/unity/features/editor/art-and-design/cinemachine>, 2023.
- [71] Microsoft, *params (Referencia de C#)*, <https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/params>, 2023.
- [72] Unity, *GameObject.GetComponent*, <https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html>, 2023.
- [73] Unity. «Component.GetComponentInChildren.» (2023), dirección: <https://docs.unity3d.com/ScriptReference/Component.GetComponentInChildren.html>.
- [74] Wikipedia. «Inyección de dependencias.» (2022), dirección: https://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_dependencias.
- [75] Unity. «Unity UI: Unity User Interface.» (2023), dirección: <https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/index.html>.
- [76] Unity. «Selectable Base Class.» (2023), dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/script-Selectable.html>.
- [77] Unity. «EventSystem.» (2023), dirección: <https://docs.unity3d.com/es/2018.4/Manual/EventSystem.html>.
- [78] Wikipedia. «Modelo–vista–controlador.» (2021), dirección: <https://es.wikipedia.org/wiki/Modelo%20vista%20controlador>.
- [79] Unity. «Animation System Overview.» (2023), dirección: <https://docs.unity3d.com/Manual/AnimationOverview.html>.
- [80] ShadowcoastGaming. «Overview of the UI in Baldur's Gate 3.» (2023), dirección: https://www.youtube.com/watch?v=eqPXY5Xvd5I&ab_channel=ShadowcoastGaming.
- [81] Unity Technologies, *Animator Class*, 2022. dirección: <https://docs.unity3d.com/Manual/class-Animator.html>.
- [82] Wikipedia, «Composite (patrón de diseño),» *Wikipedia*, 2020. dirección: [https://es.wikipedia.org/wiki/Composite_\(patr%C3%B3n_de_dise%C3%BAo\)](https://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%BAo)).
- [83] Yoones A. Sekhavat, «Behavior Trees for Computer Games,» *ResearchGate*, 2017. dirección: https://www.researchgate.net/publication/312869797_Behavior_Trees_for_Computer_Games.
- [84] Wikipedia, «Depth-first search,» *Wikipedia*, 2023. dirección: https://en.wikipedia.org/wiki/Depth-first_search.
- [85] Unity Technologies, *UI Builder*, 2022. dirección: <https://docs.unity3d.com/Manual/UIBuilder.html>.
- [86] Unity Technologies, *Structure UI with UXML*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/UIE-UXML.html>.
- [87] Unity Technologies, *Style UI*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/UIE-USS.html>.
- [88] Unity Technologies, *EditorWindow*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/EditorWindow.html>.
- [89] Unity Technologies, *Custom Editors*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/editor-CustomEditors.html>.
- [90] Unity Technologies, *TwoPaneSplitView*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.TwoPaneSplitView.html>.

- [91] Unity Technologies, *VisualElement*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.VisualElement.html>.
- [92] Unity Technologies, *GraphView*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/Experimental.GraphView.GraphView.html>.
- [93] Unity Technologies, *Edge*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/Experimental.GraphView.Edge.html>.
- [94] Unity Technologies, *Node*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/Experimental.GraphView.Node.html>.
- [95] Unity Technologies, *Special folder names*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/SpecialFolders.html>.
- [96] Unity Technologies, *UxmlFactory*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.Foldout.UxmlFactory.html>.
- [97] Unity Technologies, *UxmlTraits*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.UxmlTraits.html>.
- [98] Unity Technologies, *VisualElement.Add*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/ScriptReference/UIElements.VisualElement.Add.html>.
- [99] Unity Technologies, *Immediate Mode GUI (IMGUI)*, 2023. dirección: <https://docs.unity3d.com/2023.2/Documentation/Manual/GUIScriptingGuide.html>.
- [100] Wikipedia. «Parry (fencing).» (2022), dirección: [https://en.wikipedia.org/wiki/Parry_\(fencing\)](https://en.wikipedia.org/wiki/Parry_(fencing)).
- [101] WBN - World Boxing News. «Muhammad Ali dodges Michael Dokes punches in 1977 exhibition.» (2016), dirección: https://www.youtube.com/watch?v=OPr73038ddA&ab_channel=WBN-WorldBoxingNews.
- [102] Wikipedia. «Absolver.» (2023), dirección: <https://en.wikipedia.org/wiki/Absolver>.
- [103] Unity. «Interactions.» (2023), dirección: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6/manual/Interactions.html>.
- [104] Unity. «Struct InputAction.CallbackContext.» (2023), dirección: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.InputAction.CallbackContext.html>.
- [105] Matthew-J-Spencer. «Unity benchmarks by Tarodev.» (2022), dirección: <https://github.com/Matthew-J-Spencer/Unity-Benchmarks>.
- [106] Unity. «SaveBindingOverridesAsJson.» (2023), dirección: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.1/manual/ActionBindings.html>.
- [107] Unity. «QualitySettings.» (2023), dirección: <https://docs.unity3d.com/ScriptReference/QualitySettings.html>.
- [108] Unity. «Screen.» (2023), dirección: <https://docs.unity3d.com/ScriptReference/Screen.html>.
- [109] Unity. «LoadSceneMode.» (2023), dirección: <https://docs.unity3d.com/ScriptReference/SceneManagement.LoadSceneMode.html>.
- [110] Unity. «DontDestroyOnLoad.» (2023), dirección: <https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html>.
- [111] Unity. «CinemachineVirtualCamera.» Unity, 2023. dirección: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/api/Cinemachine.CinemachineVirtualCamera.html>.
- [112] Unity. «Body properties.» Unity, 2023. dirección: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineVirtualCameraBody.html>.

- [113] Unity, «Aim properties,» *Unity*, 2023. dirección: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineVirtualCameraAim.html>.
- [114] Unity, «Noise properties,» *Unity*, 2023. dirección: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineVirtualCameraNoise.html>.
- [115] SHN Survival Horror Network. «Resident Evil 2 Remake.» (2019), dirección: https://www.youtube.com/watch?v=EyYbXNsE2k4&ab_channel=SHNSurvivalHorrorNetwork.
- [116] Unity, «Orbital Transposer,» *Unity*, 2023. dirección: <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineBodyOrbitalTransposer.html>.
- [117] Unity, «Mathf.Lerp,» *Unity*, 2023. dirección: <https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html>.
- [118] Wikipedia. «Curva de Bézier.» (2023), dirección: https://es.wikipedia.org/wiki/Curva_de_B%C3%A9zier.
- [119] Masahiro Sakurai on Creating Games. «Screen Shake [Effects].» (2023), dirección: https://www.youtube.com/watch?v=2JXR7IASog&ab_channel=MasahiroSakuraionCreatingGames.
- [120] Masahiro Sakurai on Creating Games. «Stop for Big Moments! [Design Specifics].» (2023), dirección: https://www.youtube.com/watch?v=OdVkJ0zdCPw&ab_channel=MasahiroSakuraionCreatingGames.
- [121] Masahiro Sakurai on Creating Games. «Eight Hit Stop Techniques [Design Specifics].» (2023), dirección: https://www.youtube.com/watch?v=tycbMSjDDLg&ab_channel=MasahiroSakuraionCreatingGames.
- [122] Unity, «Time.timeScale,» *Unity*, 2023. dirección: <https://docs.unity3d.com/ScriptReference/Time-timeScale.html>.

Gracias por leer este Trabajo Fin de Grado.