UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería de Software

Desarrollo de un videojuego serio para la enseñanza y aplicación de la teoría de conjuntos mediante mecánicas interactivas en un entorno lúdico.

Development of a serious video game for teaching and applying set theory through interactive mechanics in a playful environment.

Realizado por
Mario Cortés Herrera

Tutorizado por
Antonio José Fernández Leiva

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA DEL SOFTWARE

Desarrollo de un videojuego serio para la enseñanza y aplicación de la teoría de conjuntos mediante mecánicas interactivas en un entorno lúdico

Development of a serious video game for teaching and applying set theory through interactive mechanics in a playful environment

Realizado por
Mario Cortés Herrera

Tutorizado por
Antonio José Fernández Leiva

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2025

Fecha defensa: septiembre de 2025

"Mathematics is the art of giving the same name to different things."

— Henri Poincaré

"Video games are the only art form where the audience becomes the creator."

— Will Wright

"Dedicado a todo aquel que ame los videojuegos, el desarrollo de software y las matemáticas, y que quiera hacer de este mundo un lugar mejor a través de ellos."

— Mario Cortés Herrera

Agradecimientos

Un agradecimiento especial a mi pareja, Paula, por su apoyo incondicional, su paciencia y comprensión durante todos estos años, animándome a seguir adelante y a dar lo mejor de mí siempre, por conocerme mejor que yo mismo y confiar en mí cuando yo no lo hacía, por sus grandes ideas e inspiraciones, gracias por ser mi compañera de vida.

Otro a mi familia, en especial a mi madre, por apoyarme y preocuparse siempre en lo que he querido dedicarme a ser y por todos los táperes de comida que me he llevado a la universidad.

Otro agradecimiento a Javi, por haber estado desde el primer año de carrera conmigo, por todos los trabajos que hicimos juntos, por habernos graduado, por crear juntos videojuegos y compartiendo esta pasión, y ahora con muchas ganas de continuar este camino contigo haciendo el máster.

Gracias también a Álvaro, por ser la primera persona que conocí en la universidad, por todas nuestras charlas en la cafetería, por enseñarme cosas que desconocía y por siempre querer probar los juegos que creé.

Otro más a Nico y Andrés, por formar junto con Javi y yo nuestro grupo *Magnet Studio*, por todos los cafés que tomamos, por el desarrollo tan ambicioso de *PokéRoll* y *My Life As A*, y por todas aquellas horas de estudio que acabaron siendo tan amenas con vosotros.

Un último agradecimiento a mi tutor, Antonio José Fernández Leiva, por su orientación y apoyo a lo largo del desarrollo de este trabajo, por confiar en la viabilidad del proyecto y por compartir su experiencia y conocimientos en el ámbito del desarrollo de videojuegos.

Abstract

This work presents **Set Wars: Cardinal Bounds**, a serious 2D video game that combines *shooter arcade* mechanics with strategic elements based on mathematical operations from set theory.

The project was developed using **Godot Engine 4.4.1** and **C#**, implementing a gameplay system in which the player controls a spaceship representing a set and must face enemies by applying set operations. These operations not only determine the effectiveness of attacks and defenses but also serve as the core of progression and challenge resolution in each level.

The proposal aims to provide a playful and interactive environment to facilitate the understanding of abstract concepts from set theory, integrating learning naturally into the gaming experience.

Various levels, enemies, and mechanics were designed to require the conscious and strategic use of these operations, fostering both learning and the player's tactical creativity.

The results show that it is possible to apply complex mathematical concepts in a video game context without sacrificing gameplay. *Set Wars: Cardinal Bounds* maintains the dynamic pace of the arcade genre while reinforcing the learning of mathematical fundamentals through interactive practice and immediate feedback.

Keywords: Set theory, Serious game, Godot Engine, C#, Game-based learning

Resumen

Este trabajo presenta **Set Wars: Cardinal Bounds**, un videojuego serio en 2D que combina mecánicas de disparos tipo *shooter arcade* con elementos estratégicos basados en operaciones matemáticas de la teoría de conjuntos.

El proyecto se desarrolló utilizando **Godot Engine 4.4.1** y **C#**, implementando un sistema de juego en el que el jugador controla una nave espacial que representa un conjunto y debe enfrentarse a enemigos aplicando operaciones de conjuntos. Estas operaciones no solo determinan la efectividad de los ataques y defensas, sino que también sirven como núcleo de la progresión y resolución de desafíos en cada nivel.

La propuesta busca ofrecer un entorno lúdico e interactivo para facilitar la comprensión de conceptos abstractos de la teoría de conjuntos, integrando la enseñanza de forma natural en la experiencia de juego.

Se diseñaron distintos niveles, enemigos y mecánicas que requieren el uso consciente y estratégico de dichas operaciones, fomentando tanto el aprendizaje como la creatividad táctica del jugador.

Los resultados obtenidos demuestran que es posible aplicar conceptos matemáticos complejos en un contexto de videojuego sin sacrificar la jugabilidad. *Set Wars: Cardinal Bounds* logra mantener un ritmo dinámico propio del género arcade, a la vez que refuerza el aprendizaje de fundamentos matemáticos mediante la práctica interactiva y la retroalimentación inmediata.

Palabras clave: Teoría de conjuntos, Videojuego serio, Godot Engine, C#, Aprendizaje basado en juegos

Índice

1. Introducción **23**

1.1. Estructura del documento . . . 23

1.2. Motivación . . . 23

1.3. Objetivos . . . 24

1.4. Tecnologías usadas . . . 24

2. Contexto **27**

2.1. Género . . . 27

2.2. Referentes . . . 27

2.2.1. Beat Hazard Ultra . . . 28

2.2.2. Geometry Wars: Retro Evolved 2 . . . 29

2.2.3. Asteroids . . . 30

2.2.4. Animation vs Math — Alan Becker . . . 32

2.3. Teoría de conjuntos . . . 32

2.3.1. Origen y desarrollo histórico . . . 33

2.3.2. Definición y notación básica . . . 33

2.3.3. Cardinalidad y conjuntos especiales . . . 34

2.3.4. Operaciones fundamentales . . . 34

2.3.5. Aplicaciones . . . 35

2.4. Godot Engine . . . 35

2.4.1. Orígenes y contexto de Godot . . . 36

2.4.2. Lenguajes de programación soportados . . . 36

2.4.3. Comparativa con Unity . . . 37

2.4.4. Sistema de nodos y herencia . . . 38

2.4.5. *Signals* y comunicación entre nodos . . . 39

2.4.6. Funciones base nativas de los nodos . . . 40

2.4.7. Versión 4.2 y migración a 4.4.1 . . . 41

2.5. Serious Games . . . 41

2.5.1. Características y valor educativo . . . 42
2.5.2. Aplicación en el aprendizaje matemático . . . 42
2.5.3. Set Wars como serious game . . . 43
2.6. Metodología Scrum . . . 44
2.6.1. Fundamentos de Scrum . . . 44
2.6.2. Roles en Scrum . . . 45
2.6.3. Eventos de Scrum . . . 45
2.6.4. Artefactos de Scrum . . . 46
2.7. Game Experience Questionnaire (GEQ) . . . 46

3. Análisis y Diseño de la Solución **49**
3.1. Metodología aplicada . . . 49
3.1.1. Sprints . . . 50
3.1.2. Diagrama de Gantt . . . 52
3.2. Primer concepto . . . 54
3.3. Requisitos . . . 54
3.3.1. Requisitos funcionales . . . 54
3.3.2. Requisitos no funcionales . . . 57
3.4. Diagramas de flujo . . . 59
3.4.1. Diagramas de flujo de las interfaces . . . 60
3.4.2. Diagramas de flujo del comportamiento de los enemigos . . . 61
3.5. Bocetos de las interfaces . . . 63
3.6. Bocetos de los personajes . . . 67
3.7. Estilo audiovisual y experiencia de inmersión . . . 68
3.8. Arquitectura general del proyecto . . . 69

4. Implementación y Desarrollo **73**
4.1. Estructura de las carpetas . . . 73
4.2. Configuración del proyecto . . . 74
4.3. *Game:* La clase raíz . . . 77
4.3.1. *Game Inputs* . . . 79
4.3.2. *Game Language* . . . 80

4.3.3. *Game Display* . . . 81
4.3.4. *Game Data* . . . 82
4.3.5. *Game Settings* . . . 83
4.3.6. *Game Audio* . . . 83
4.3.7. *Game Levels* . . . 85
4.3.8. *Game UI* . . . 86
4.3.9. *Gameplay* . . . 87
4.3.10. *Background* . . . 88
4.4. Utilidades y extras . . . 90
4.4.1. Utilidades para el audio . . . 90
4.4.2. Utilidades para los colores . . . 91
4.4.3. Utilidades para los grupos . . . 92
4.4.4. Utilidades para las capas . . . 92
4.4.5. Utilidades para los nodos y procesamiento . . . 93
4.4.6. *Logger* . . . 94
4.4.7. Rutas del proyecto y traducciones . . . 94
4.5. *Shaders* del proyecto . . . 94
4.5.1. *Shader* de desenfoque . . . 95
4.5.2. *Shader* de resplandor (*Glow*) . . . 96
4.6. Reproductores de audio . . . 97
4.6.1. Gestor de audio . . . 98
4.6.2. Gestores de audio globales . . . 99
4.7. Entidades . . . 100
4.8. Gestor de salud . . . 102
4.8.1. Corazones y elementos de un conjunto . . . 103
4.9. Entidad muerta . . . 104
4.10. Proyectiles . . . 106
4.10.1. Daño . . . 108
4.10.2. Tipos de proyectiles . . . 110
4.11. Gestor de proyectiles . . . 111
4.11.1. Puntos de disparo . . . 112

4.11.2. Estrategias de disparo . . . 113
4.12. Gestor de contacto . . . 115
4.13. Gestor de comportamiento . . . 115
4.13.1. Comportamiento . . . 117
4.13.2. Movimiento . . . 118
4.13.3. Ataque . . . 120
4.13.4. Visión . . . 121
4.14. Gestor de *sprites* . . . 121
4.14.1. Animador de *sprites* . . . 123
4.14.2. Animaciones para *shaders* . . . 124
4.15. Propulsores . . . 125
4.16. Tipos de entidades . . . 127
4.16.1. Jugador . . . 128
4.16.2. Enemigos . . . 129
4.17. Cámara . . . 132
4.18. Mirilla (*Scope*) . . . 133
4.19. Interfaces de usuario . . . 135
4.19.1. Componentes . . . 135
4.19.2. Menús . . . 142
4.19.3. Transición de carga . . . 149
4.19.4. HUD (*Head-Up Display*) . . . 151
4.19.5. Interfaz seguidora . . . 157
4.20. Niveles . . . 158
4.20.1. Configuración de niveles . . . 159
4.21. Gestor de datos guardados . . . 160
4.21.1. Archivo de datos guardados . . . 161
4.21.2. Datos de configuración . . . 162
4.21.3. Datos de progreso . . . 162
4.22. *Triggers* . . . 163
4.22.1. *Triggered Trigger* . . . 167
4.22.2. *Trigger* puerta lógica . . . 173

4.23. Elementos del escenario 174
4.23.1. Límites del mapa 175
4.23.2. *Items* 176
4.23.3. *Spawner* del jugador 177
4.23.4. Portal de fin del nivel 178

5. Pruebas y Correcciones **181**
5.1. Respuestas al cuestionario 182
5.2. Retroalimentación y mejoras 184
5.3. Análisis de resultados y conclusiones 186

6. Conclusiones y Líneas Futuras **189**
6.1. Conclusiones generales 189
6.2. Líneas futuras 190
6.3. Aprendizaje personal adquirido 192

Referencias **193**

Apéndice A. High Concept Document **195**

Apéndice B. Game Design Document **201**

Apéndice C. Manual de Instalación **257**

Índice de figuras

1. Logo de Beat Hazard Ultra. . . . 28
2. Ejemplo del gameplay de Beat Hazard Ultra. . . . 29
3. Portada de Geometry Wars: Retro Evolved 2. . . . 29
4. Ejemplo del gameplay de Geometry Wars: Retro Evolved 2. . . . 30
5. Portada de Asteroids. . . . 31
6. Ejemplo del gameplay de Asteroids. . . . 31
7. Captura del vídeo Animation vs Math — Alan Becker. . . . 32
8. Diagramas de Venn de cada operación de conjuntos [5]. . . . 35
9. Ejemplo de árbol de jerarquía de nodos en Godot. . . . 38
10. Parte del árbol de herencia de nodos de Godot. . . . 39
11. Esquema de la metodología Scrum [13]. . . . 46
12. Diagrama de Gantt de la planificación del proyecto. . . . 53
13. Diagrama de flujo de la navegación en el menú de selección de nivel. . . . 60
14. Diagrama de flujo de la navegación global entre todos los menús del juego. . . . 61
15. Diagrama de flujo del comportamiento de los enemigos. . . . 62
16. Boceto del menú principal. . . . 63
17. Boceto de la selección de niveles. . . . 63
18. Boceto del menú de ajustes. . . . 64
19. Boceto del menú de pausa. . . . 64
20. Boceto del menú de *Game Over*. . . . 64
21. Boceto del menú de nivel completado. . . . 65
22. Boceto de los Créditos. . . . 65
23. Boceto de la salud del jugador en el HUD. . . . 65
24. Boceto del selector de proyectiles en el HUD. . . . 66
25. Boceto de la interfaz del diálogo. . . . 66
26. Boceto de la interfaz de leyenda de controles. . . . 66
27. Boceto del enemigo básico. . . . 67
28. Boceto del jugador. . . . 67

29. Diagrama de la arquitectura general del proyecto. 71
30. Icono del juego (Windows). 75
31. Pantalla de carga del juego al iniciarlo (*boot splash*). 75
32. Grupos (*tags*) creados para el juego. 76
33. Capas de renderizado nombradas. 76
34. Capas de físicas nombradas. 77
35. Algunas entradas y sus asignaciones del (*input map*) del juego. 77
36. Árbol de nodos de la escena principal del juego en Godot. 78
37. Representación UML de la clase *Game*. 79
38. Representación UML de la clase *GameInputs*. 80
39. Representación UML de la clase *GameLanguage*. 81
40. Inicio del archivo *.csv* de traducciones del juego. 81
41. Representación UML de la clase *GameDisplay*. 82
42. Representación UML de la clase *GameData*. 82
43. Representación UML de la clase *GameSettings*. 83
44. Configuración de los buses de audio en Godot. 84
45. Representación UML de la clase *GameAudio*. 84
46. Código de *AudioBuses*. 85
47. Representación UML de la clase *GameLevels* y *LevelData*. 85
48. Vista del editor de Godot para asignar los datos de cada nivel en *GameLevels*. . 86
49. Ejemplo del árbol de nodos de *GameUI* durante la ejecución del juego. 87
50. Representación UML de la clase *GameUI*. 87
51. Representación UML de la clase *Gameplay*. 88
52. Árbol de nodos de la escena de *Background* en Godot. 89
53. Representación UML de la clase *Background*. 89
54. Escena de *Background*. 90
55. Representación UML de la clase *AudioType* con su método de extensión en *AudioUtils*. 91
56. Representación UML de la clase *GameColor* y los métodos de extensión de *GameColorUtils*. 91

57. Representación UML de la clase *GameGroup* y su método de extensión en *GameGroupUtils*. . . . 92
58. Representación UML de la clase *GamePhysicsLayer* y los métodos de extensión en *GamePhysicsLayerUtils*. . . . 93
59. Representación UML de la clase *ProcessUtils*. . . . 93
60. Representación UML de la clase *Logger*. . . . 94
61. *Shader* de desenfoque aplicado como material desde el editor de Godot. . . . 95
62. Demostración en el juego del *Shader* de desenfoque. . . . 95
63. *Shader* de resplandor aplicado como material desde el editor de Godot. . . . 96
64. Demostración en el juego del *Shader* de resplandor. . . . 97
65. Representación UML de la clase *AudioPlayer* y *AudioPlayer2D*. . . . 98
66. Ejemplo de un *AudioManager* con varios reproductores en Godot. . . . 99
67. Diagrama de clase de *AudioManager* y sus derivados. . . . 100
68. Árbol de nodos de los gestores de audio globales en Godot. . . . 100
69. Representación UML de la clase *Entity*. . . . 101
70. Representación UML de la clase *HealthManager*. . . . 102
71. Representación UML de la clase *Heart* y su clase base *SetElement*. . . . 104
72. Diagrama de clase de *DeadEntity* y sus derivadas. . . . 105
73. Demostración en el juego de la animación de muerte de un asteroide. . . . 106
74. Árbol de nodos de la escena de *DeadPlayer* en Godot. . . . 106
75. Ejemplo del árbol de nodos de la escena de un *Projectile* en Godot. . . . 108
76. Diagrama de clase de *Damage* y sus derivados. . . . 109
77. Diagrama de clase de *Projectile* y sus derivados. . . . 110
78. Diagrama de clase de *ProjectileManager* y sus derivados. . . . 112
79. Ejemplo de puntos de disparo en Godot. . . . 113
80. Diagrama de clase de *FiringStrategy* y sus derivados. . . . 114
81. Representación UML de la clase *ContactManager* y sus derivados. . . . 115
82. Diagrama de clase de *BehaviorManager* y sus derivados. . . . 116
83. Diagrama de clase de *Behavior* y sus derivados. . . . 118
84. Representación UML de la clase *SpriteManager*. . . . 122
85. Representación UML de la clase *SpriteAnimator* y su clase base *CanvasAnimator*. 124

86. Representación UML de la clase *ShaderMaterialAnimation*. 125
87. Representación UML de la clase *Thrusters*. 126
88. Ejemplo del árbol de nodos de los *Thrusters* del jugador. 127
89. Ejemplo visual de los *Thrusters* del jugador desde el editor. 127
90. Árbol de nodos del *Player*. 129
91. Árbol de nodos del *Asteroid*. 130
92. Árbol de nodos del *BasicEnemy*. 131
93. Diagrama de clase de *Entity* y sus derivados. 132
94. Representación UML de la clase *LevelCamera*. 133
95. Representación UML de la clase *Scope*. 134
96. Representación UML de la clase *BasicLabel*. 136
97. Representación UML de la clase *BasicButton* y sus derivados. 137
98. Representación UML de la clase *BasicVerticalSlider*. 138
99. Representación UML de la clase *InputIcon*. 139
100. Ejemplo del árbol de nodos de una *BasicSelection*. 140
101. Representación UML de la clase *BasicSelection*. 140
102. Representación UML de la clase *BasicVerticalScrollBar*. 141
103. Árbol de nodos del *MainMenu*. 143
104. Árbol de nodos del *SelectableLevel*. 144
105. Árbol de nodos del *LevelSelection*. 144
106. Árbol de nodos del *PauseMenu*. 145
107. Árbol de nodos del *SettingsMenu*. 146
108. Árbol de nodos del *GameOver*. 147
109. Árbol de nodos del *LevelCompletedMenu*. 147
110. Árbol de nodos del *CreditLine*. 148
111. Árbol de nodos del *CreditsMenu*. 148
112. Diagrama de clase de *Menu* y sus derivados. 149
113. Árbol de nodos del *LoadingTransition*. 150
114. Representación UML de la clase *LoadingTransition*. 150
115. Representación UML de la clase *LevelUI*. 151
116. Árbol de nodos de la *HealthUI*. 152

117. Árbol de nodos del componente *HeartUI*. . . . 153
118. Representación UML de la clase *HealthUI* y *HeartUI*. . . . 153
119. Árbol de nodos de la *ProjectilesUI*. . . . 154
120. Árbol de nodos del componente *ColorSquareUI*. . . . 154
121. Representación UML de la clase *ProjectilesUI* y *ColorSquareUI*. . . . 154
122. Árbol de nodos del componente *DialogueUI*. . . . 156
123. Representación UML de la clase *DialogueUI* y *Dialogue*. . . . 156
124. Árbol de nodos de la *InputLegendUI*. . . . 157
125. Representación UML de la clase *InputLegendUI* e *InputLegend*. . . . 157
126. Representación UML de la clase *FollowerUI*. . . . 158
127. Ejemplo del árbol de nodos de un *Level*. . . . 159
128. Representación UML de la clase *Level*. . . . 160
129. Representación UML de la clase *DataManager* y *SavedData* y los derivados de ambos. . . . 161
130. Diagrama de clase de *Trigger* y sus derivados. . . . 164
131. Árbol de nodos de una *BoundaryWall*. . . . 175
132. Representación UML de la clase *BoundaryWall*. . . . 176
133. Árbol de nodos de un *ProjectileGrantingItem* mostrando la composición de *triggers*. . . . 177
134. Diagrama de clase de *Item* y sus derivados. . . . 177
135. Árbol de nodos del *PlayerSpawner*. . . . 178
136. Árbol de nodos del *EndSpiral*. . . . 179
137. Representación UML de la clase *Spiral* y sus derivados. . . . 180
138. Escala de valoración para el cuestionario GEQ. . . . 182
139. GEQ - Competence: afirmaciones valoradas por los *testers*. . . . 182
140. GEQ - Sensory & Imaginative Immersion: afirmaciones valoradas por los *testers*. 183
141. GEQ - Flow: afirmaciones valoradas por los *testers*. . . . 183
142. GEQ - Tension / Annoyance: afirmaciones valoradas por los *testers*. . . . 183
143. GEQ - Challenge: afirmaciones valoradas por los *testers*. . . . 184
144. GEQ - Positive Affect: afirmaciones valoradas por los *testers*. . . . 184
145. GEQ - Negative Affect: afirmaciones valoradas por los *testers*. . . . 184

1 Introducción

En esta primera sección se encuentra un resumen de la estructura del documento y se explica la motivación que impulsó la creación del proyecto, los objetivos planteados para este, y finalizando con las tecnologías empleadas durante todo el proceso de desarrollo.

1.1. Estructura del documento

La presente memoria está estructurada de la siguiente forma:

- En la **Introducción** se incluye la motivación, los objetivos y las tecnologías empleadas.
- El **Contexto** recoge los fundamentos teóricos y referencias que sustentan la idea del proyecto.
- El **Análisis y Diseño de la Solución** detalla la metodología utilizada, los requisitos del sistema y los primeros diseños conceptuales.
- En la sección de **Implementación y Desarrollo** se describe el proceso técnico y creativo de construcción del videojuego.
- La sección de **Pruebas** presenta los métodos utilizados para verificar la calidad y propósito del juego, así como para identificar errores y aspectos a mejorar en la experiencia.
- Finalmente, en **Conclusiones y Líneas Futuras** se resumen los logros alcanzados y las posibles ampliaciones del proyecto.

1.2. Motivación

En este documento se presenta la memoria del Trabajo de Fin de Grado titulado *"Desarrollo de un videojuego serio para la enseñanza y aplicación de la teoría de conjuntos mediante*

mecánicas interactivas en un entorno lúdico", cuyo nombre es ***Set Wars: Cardinal Bounds***. El proyecto consiste en el desarrollo de un videojuego serio en 2D que combina mecánicas de disparos tipo *shooter arcade* con elementos estratégicos basados en operaciones matemáticas propias de la teoría de conjuntos.

La motivación principal de este trabajo surge de la necesidad de explorar nuevas formas de enseñanza para conceptos abstractos y, en ocasiones, complejos, como los que abarca la teoría de conjuntos. Esta rama fundamental de las matemáticas presenta aplicaciones clave en informática, lógica y otras ciencias, pero su aprendizaje suele apoyarse principalmente en explicaciones teóricas y ejercicios escritos, lo que puede dificultar la comprensión por parte del estudiantado. Integrar dichos conceptos en un entorno de juego interactivo pretende ofrecer una experiencia más dinámica, atractiva y efectiva.

1.3. Objetivos

El videojuego está diseñado para que el jugador asuma el control de una nave espacial que representa un conjunto. A través de distintos niveles, deberá enfrentarse a enemigos y superar desafíos empleando operaciones como unión, intersección, diferencia y complemento, tanto para atacar como para defenderse. Esta combinación de mecánicas busca que el jugador interiorice la lógica de la teoría de conjuntos de manera natural y progresiva, sin perder la diversión propia del medio.

1.4. Tecnologías usadas

Para el desarrollo de este proyecto se han empleado diversas tecnologías, cada una seleccionada por sus características específicas y su adecuación a los objetivos del trabajo:

1. **Godot Engine** - Motor de videojuegos de código abierto que proporciona un entorno completo para el desarrollo. Inicialmente se planteó usar la versión 4.2 o posterior, con lo que finalmente se implementó con la versión 4.4.1 para aprovechar mejoras de rendimiento, estabilidad y nuevas funcionalidades útiles agregadas. Se seleccionó por su facilidad de uso, su curva de aprendizaje accesible y su rendimiento optimizado para juegos 2D. Además, su filosofía de nodos y escenas permite una estructuración modular ideal para este tipo de proyectos.

2. **C#** - Lenguaje de programación orientado a objetos utilizado para implementar toda la lógica principal del juego. Ofrece un equilibrio óptimo entre rendimiento y facilidad de desarrollo, además de aportar ventajas de tipado fuerte que facilitan la detección temprana de errores.

3. **Godot Shader Language** - Lenguaje específico de Godot empleado para la programación de *shaders*[1], ya que esta funcionalidad no está disponible con C# en Godot. Permite crear efectos visuales personalizados dentro del motor.

4. **Krita y Photoshop** - Softwares de edición de imágenes combinados para la creación original de todos los elementos visuales del juego. Todos los *sprites*[2], elementos de interfaz, partículas y efectos visuales fueron diseñados completamente desde cero sin utilizar recursos externos, demostrando un compromiso con la originalidad artística del proyecto. Ambas herramientas aportaron funcionalidades complementarias para la creación de pixel art, permitiendo mantener una estética consistente y personalizada a lo largo de toda la experiencia de juego.

5. **GitHub** - Sistema de control de versiones y plataforma de alojamiento que permitió gestionar los cambios en el código. Los mensajes de commit sirvieron como diario de desarrollo, documentando adiciones y modificaciones relevantes durante todo el proceso, lo que facilitó la redacción posterior de esta memoria.

6. **Visual Studio Code** - Editor de código empleado para todo el desarrollo de *scripts* y archivos de configuración. Sus extensiones especializadas para C# y Godot Engine aumentaron significativamente la productividad, permitiendo depuración en tiempo real y análisis estático de código.

7. **Overleaf** - Plataforma colaborativa de edición LaTeX en línea utilizada para la redacción y composición de esta memoria, facilitando la estructuración del documento y la gestión de referencias bibliográficas.

[1] Un *shader* es un programa que se ejecuta en la GPU y controla el proceso de renderizado de los gráficos, permitiendo crear efectos visuales avanzados y personalizados.

[2] Un *sprite* es una imagen bidimensional utilizada para representar visualmente elementos en videojuegos. Normalmente se gestiona y dibuja de forma independiente sobre el fondo, permitiendo animaciones y cambios de estado.

8. **Audacity** - Software de edición de audio empleado para modificar archivos de audio obtenidos de fuentes externas. Aunque los recursos sonoros no fueron creados desde cero, esta herramienta permitió personalizarlos según las necesidades específicas del juego, principalmente para recortar secciones y ajustar volúmenes que mejoraran la experiencia auditiva.

9. **TypeLight** - Herramienta especializada en la edición de fuentes tipográficas. Si bien se utilizaron fuentes de origen externo, esta aplicación permitió personalizarlas mediante la adición de caracteres específicos y modificaciones que aseguraron su correcta integración y legibilidad dentro de la interfaz del juego, manteniendo la coherencia estética.

10. **GanttProject** - Herramienta gratuita, muy potente para la creación de diagramas de Gantt, necesario para la aplicación de la metodología *Scrum* en este proyecto.

11. **Frame0** - Software fácil de usar, utilizado para la creación de bocetos de las interfaces de usuario. Siendo una altenativa a *Balsamiq*.

12. **Draw.io** - Herramienta online que ha sido utilizada para la creación de los diagramas de flujo. Permite vincularla con Google Drive, exportar a múltiples formatos y ofrece numerosas opciones de diseño que facilitan la creación de diagramas profesionales.

Esta combinación de tecnologías ha permitido desarrollar un producto de buena calidad y educativo, manteniendo al mismo tiempo un proceso de desarrollo estructurado y eficiente.

2
Contexto

En este capítulo se presentan los fundamentos conceptuales y las bases lúdicas que sustentan el desarrollo de *Set Wars: Cardinal Bounds*.

En primer lugar, se contextualiza el género al que pertenece el videojuego y se analizan ejemplos como referentes clave de otros juegos que han inspirado el diseño y la jugabilidad del proyecto.

Seguidamente, se abordan los conceptos esenciales de la teoría de conjuntos, fundamentales para comprender la mecánica principal del videojuego.

Finalmente, se explica el desarrollo en Godot Engine, detallando aspectos técnicos y funcionales que proporcionan un mejor contexto para entender la estructura y funcionamiento del juego.

2.1. Género

El género al que pertenece el videojuego es el **space shooter 2D con vista cenital**, caracterizado por la acción continua, el control de movimiento libre en dos ejes y la posibilidad de disparar en cualquier dirección. Este tipo de juegos sitúa al jugador en un espacio bidimensional visto desde arriba, ofreciendo una visión completa del campo de juego y permitiendo tomar decisiones rápidas para esquivar, atacar o reposicionarse.

2.2. Referentes

Dentro del género, *Set Wars: Cardinal Bounds* se alinea con propuestas que combinan un ritmo de juego elevado con una presentación visual clara y estilizada. Estas influencias no se han replicado de manera literal, sino reinterpretadas con el estilo y propósito del juego.

Además de su base arcade, el proyecto se enmarca dentro de la categoría de **videojuegos serios** (*serious games*), cuyo objetivo no se limita al entretenimiento, sino que incluye la trans-

misión de conocimientos o el entrenamiento de habilidades. En este caso, el aprendizaje de la teoría de conjuntos se produce de forma indirecta y progresiva, favoreciendo la asimilación mediante la práctica y la experimentación en un entorno interactivo y motivador.

Entre sus principales referentes se encuentran los citados a continuación.

2.2.1. Beat Hazard Ultra

Beat Hazard Ultra (Cold Beam Games, 2011) destaca por su dinamismo visual y la intensidad de sus combates, con el jugador en el centro de la acción enfrentándose a oleadas continuas de enemigos y estímulos visuales.

Figura 1: Logo de Beat Hazard Ultra.

Otro factor importante de este juego es la música; según el ritmo e intensidad, los ataques y la iluminación de las partículas son mayores. Se puede jugar con canciones que el juego ofrece o cualquier archivo de audio que tengas en el dispositivo; un nivel se acaba cuando la canción se termina, así que la duración de este depende de su longitud.

El juego presenta diferentes modos de juego, entre los más destacables se encuentran el modo **Standard**, en el que el objetivo es sobrevivir y obtener la mayor puntuación posible mientras dura una canción; el modo **Boss Rush**, donde solo aparecen jefes, uno tras otro, durante toda la canción; el modo **Survival**, que consiste en jugar canciones de forma continua, una tras otra, hasta perder todas las vidas; y el modo **Chill Out**, un modo sin enemigos ni puntuación, diseñado únicamente para disfrutar la música y los efectos visuales.

Figura 2: Ejemplo del gameplay de Beat Hazard Ultra.

2.2.2. Geometry Wars: Retro Evolved 2

Geometry Wars: Retro Evolved 2 (Bizarre Creations, 2008) aporta una estética abstracta y minimalista, en la que los elementos en pantalla, aunque abundantes, mantienen una alta legibilidad y transmiten información relevante mediante formas geométricas y partículas. El objetivo principal es sobrevivir el mayor tiempo posible y alcanzar la máxima puntuación eliminando oleadas de enemigos que aparecen de manera constante y en patrones variados.

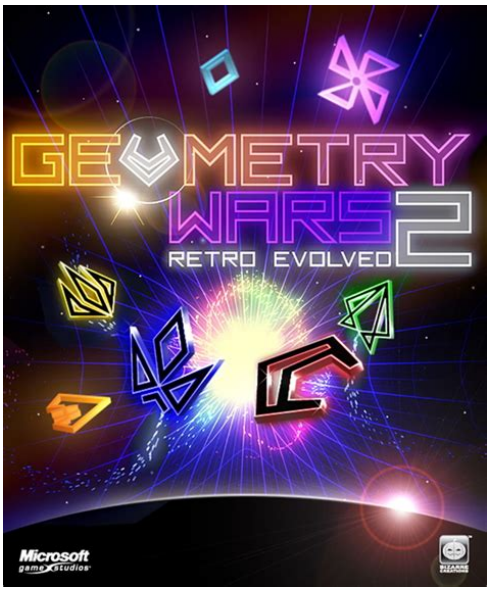


Figura 3: Portada de Geometry Wars: Retro Evolved 2.

La jugabilidad se caracteriza por su ritmo frenético y la necesidad de tomar decisiones rápidas para esquivar enemigos y proyectiles, así como para aprovechar los potenciadores que aparecen en el campo de juego. Cada tipo de enemigo presenta comportamientos y patrones de movimiento distintos, lo que obliga al jugador a adaptar continuamente su estrategia. Además, el juego ofrece varios modos, cada uno con reglas y objetivos específicos, lo que incrementa la rejugabilidad y el desafío.

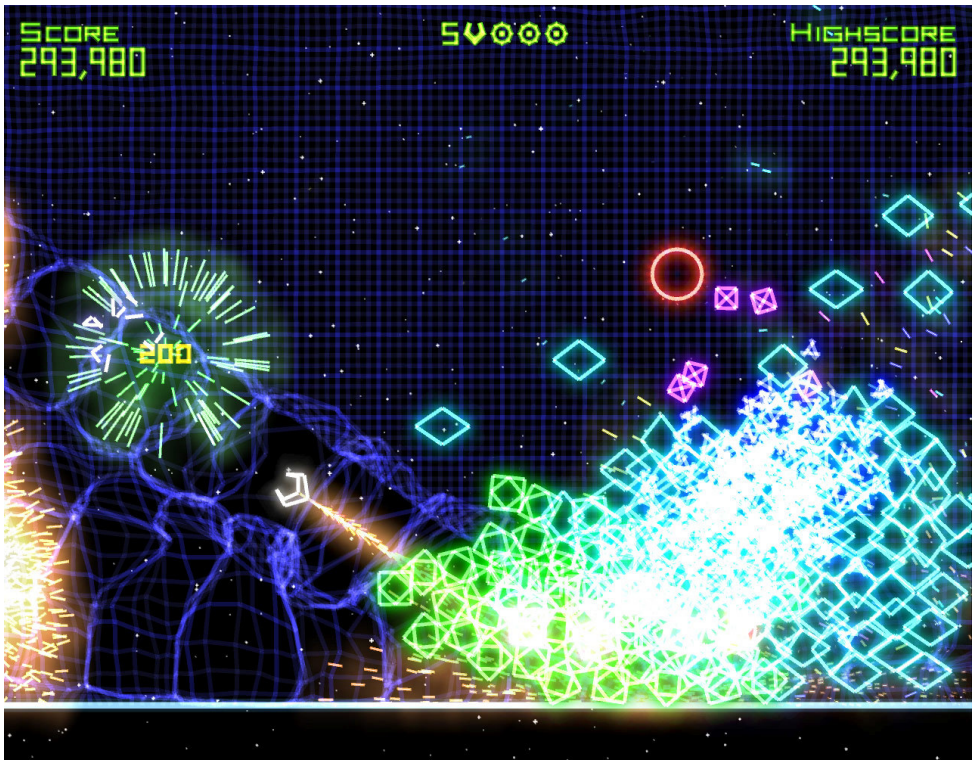


Figura 4: Ejemplo del gameplay de Geometry Wars: Retro Evolved 2.

2.2.3. Asteroids

Asteroids (Atari, 1979) es uno de los precursores más influyentes del género de los shooters espaciales con vista cenital. En este clásico arcade, el jugador controla una nave triangular en un espacio bidimensional, con el objetivo de destruir asteroides y platillos voladores mientras evita colisiones. La nave puede rotar, avanzar y disparar en la dirección en la que apunta.

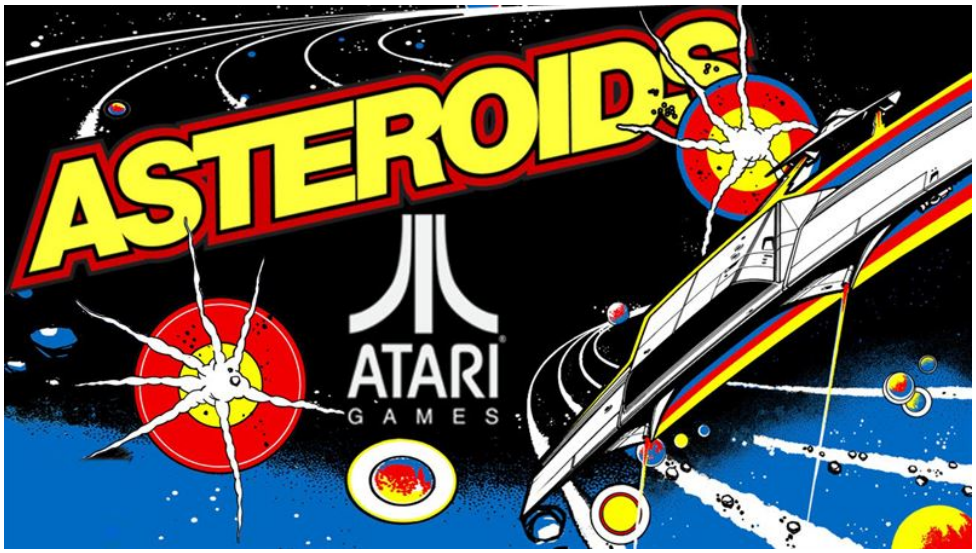


Figura 5: Portada de Asteroids.

El juego se caracteriza por su mecánica basada en la física de inercia: la nave continúa moviéndose en la dirección en la que fue impulsada hasta que el jugador aplique un nuevo impulso o frene. Los asteroides, al ser destruidos, se fragmentan en piezas más pequeñas, aumentando la dificultad y la necesidad de reflejos rápidos. La simplicidad visual, con gráficos vectoriales y una interfaz minimalista, contribuye a la claridad de la acción y a la concentración del jugador en la jugabilidad.

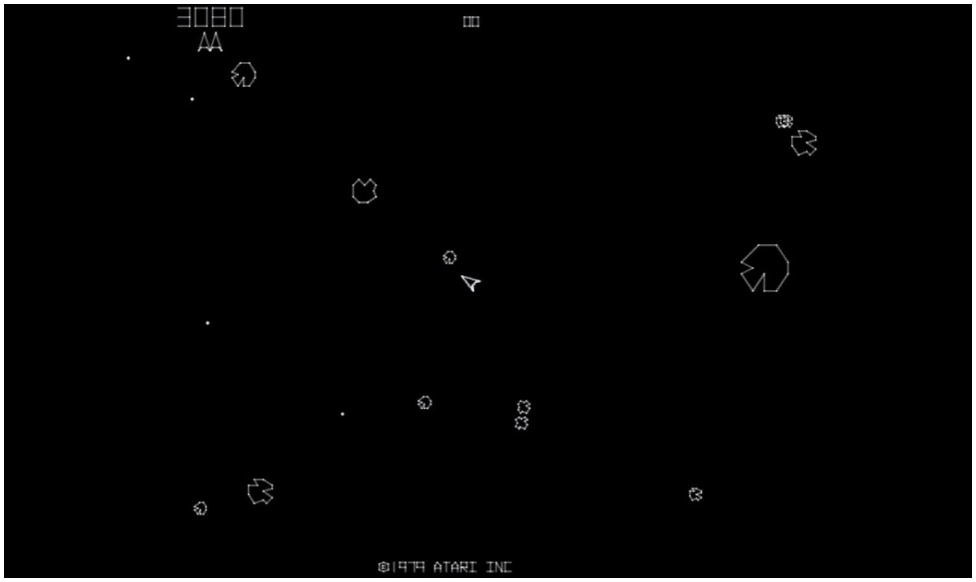


Figura 6: Ejemplo del gameplay de Asteroids.

2.2.4. Animation vs Math — Alan Becker

Es un corto animado de Alan Becker, lanzado en junio de 2023, se ha convertido en un referente creativo internacional sobre cómo las matemáticas pueden convertirse en protagonistas visuales dentro de un relato artístico. Aunque no se trata de un videojuego de por sí, su narrativa muestra cómo las operaciones matemáticas cobran vida de forma dinámica y visual: un *stickman* interactúa con símbolos matemáticos (como sumas, exponentes, raíces y la identidad de Euler) que se transforman en escenarios, armas o puertas dimensionales.

Esta animación ejemplifica cómo conceptos tradicionalmente abstractos pueden representarse de forma clara, atractiva y emocional, despertando interés sin necesidad de explicaciones técnicas explícitas. Sirve como antecedente inspirador para *Set Wars: Cardinal Bounds*, donde la teoría de conjuntos se convierte en mecánica jugable e imagen visual, otorgándole un significado natural dentro del contexto comprendido por el jugador.

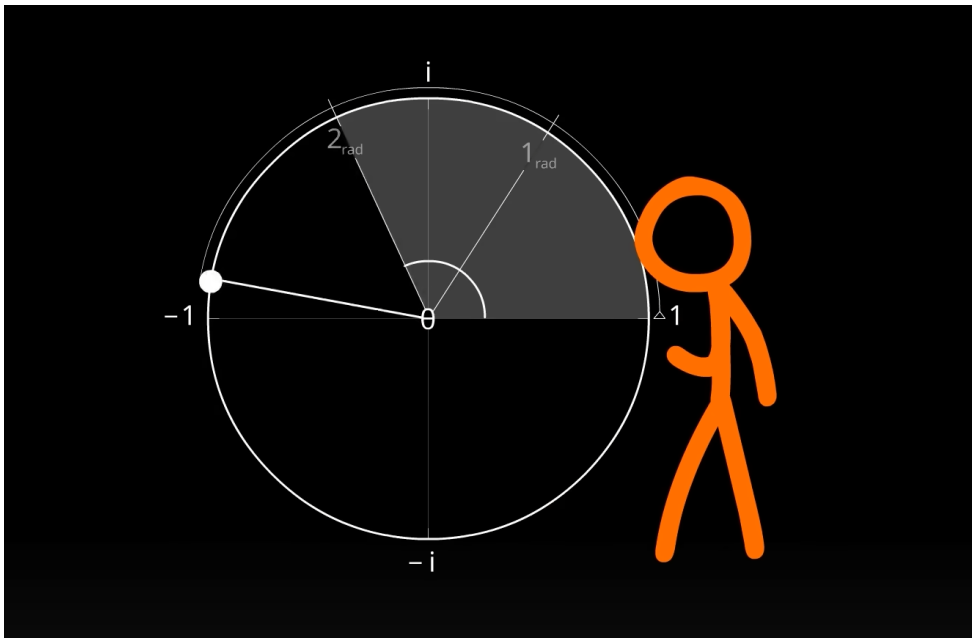


Figura 7: Captura del vídeo Animation vs Math — Alan Becker.

2.3. Teoría de conjuntos

La teoría de conjuntos es una rama de las matemáticas que estudia las colecciones de objetos [1, 2], denominadas conjuntos, y las relaciones y operaciones que se pueden establecer entre ellos. Constituye una de las bases conceptuales más sólidas de la matemática moderna,

pues proporciona un lenguaje formal para definir, clasificar y manipular entidades matemáticas de cualquier naturaleza. Desde la aritmética y el álgebra hasta el análisis y la topología, gran parte de la estructura matemática contemporánea se fundamenta en los principios de la teoría de conjuntos [3].

2.3.1. Origen y desarrollo histórico

Los orígenes formales de la teoría de conjuntos se remontan a finales del siglo XIX con los trabajos de Georg Cantor (1845–1918), matemático alemán que introdujo el concepto de conjunto infinito y desarrolló una teoría sistemática de estos objetos [4]. Aunque el término "conjunto" había sido utilizado anteriormente de manera informal en matemáticas, fue Cantor quien lo dotó de rigor y generalidad.

Cantor estableció nociones como la igualdad de cardinalidades para comparar conjuntos infinitos y mostró que no todos los infinitos son del mismo tamaño, introduciendo la distinción entre conjuntos numerables y no numerables [2]. Sus ideas provocaron un gran impacto en la comunidad matemática de la época, no exento de controversia, pues desafiaban la concepción tradicional de infinito.

A principios del siglo XX, el desarrollo de la teoría encontró ciertos problemas lógicos, como la famosa paradoja de Russell, que surge al considerar el conjunto de todos los conjuntos que no se contienen a sí mismos. Para resolver estas dificultades se formularon diferentes sistemas axiomáticos, siendo el más aceptado hoy en día la teoría axiomática de conjuntos de Zermelo-Fraenkel (ZFC), que junto con el axioma de elección constituye el marco de referencia habitual para las matemáticas modernas [2].

2.3.2. Definición y notación básica

Un conjunto se define como una colección bien determinada de elementos u objetos, llamados miembros del conjunto [1]. Estos elementos pueden ser cualquier cosa: números, puntos, letras, funciones, e incluso otros conjuntos.

La notación habitual para denotar un conjunto utiliza llaves:

$$A = \{1, 2, 3\}$$

indica que el conjunto A contiene los elementos 1, 2 y 3.

La pertenencia de un elemento a un conjunto se denota con el símbolo "$\in$":

$$2 \in A$$

significa que el 2 es un elemento de A, mientras que

$$4 \notin A$$

indica que el 4 no pertenece a A.

2.3.3. Cardinalidad y conjuntos especiales

La cardinalidad de un conjunto, denotada $|A|$, es el número de elementos que contiene. Para conjuntos finitos, la cardinalidad es un número natural. Para conjuntos infinitos, la cardinalidad se describe utilizando la noción de números cardinales transfinitos, introducida por Cantor [4, 2].

Algunos conjuntos tienen un papel fundamental en matemáticas [3]:

- **Conjunto vacío** ($\emptyset$): no contiene elementos.
- **Conjunto universal** (U): contiene todos los elementos bajo consideración en un contexto dado.
- **Subconjunto**: A es subconjunto de B si todo elemento de A está en B, lo que se escribe $A \subseteq B$.

2.3.4. Operaciones fundamentales

En teoría de conjuntos, las operaciones permiten combinar o transformar conjuntos para obtener nuevos conjuntos. Entre las más importantes se encuentran [1, 3]:

- **Unión** ($A \cup B$): conjunto de elementos que están en A, en B, o en ambos.
- **Intersección** ($A \cap B$): conjunto de elementos comunes a A y B.
- **Diferencia** ($A - B$): conjunto de elementos que están en A pero no en B.
- **Diferencia simétrica** ($A \triangle B$): conjunto de elementos que están en A o en B, pero no en ambos simultáneamente.

- **Complemento** (A' o $U - A$): conjunto de elementos que no están en A, tomando como referencia el conjunto universal U.

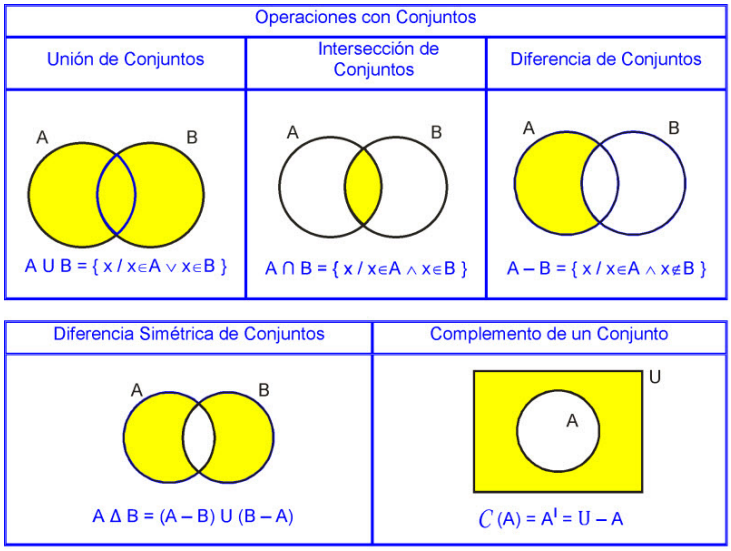


Figura 8: Diagramas de Venn de cada operación de conjuntos [5].

2.3.5. Aplicaciones

La teoría de conjuntos tiene aplicaciones directas en múltiples campos [3, 6]:

- **Matemáticas puras**: base de estructuras como espacios topológicos, grupos o funciones.
- **Informática**: modelado de bases de datos, álgebra relacional, diseño de algoritmos.
- **Lógica y filosofía**: formalización de proposiciones y razonamiento deductivo.
- **Probabilidad y estadística**: definición de espacios muestrales y eventos.

En definitiva, la teoría de conjuntos no es solo una rama más de las matemáticas, sino el lenguaje y la estructura subyacente en las que se apoya una parte esencial de la ciencia moderna.

2.4. Godot Engine

Godot Engine es un motor de videojuegos libre y de código abierto, diseñado para el desarrollo de aplicaciones interactivas tanto en 2D como en 3D. Su enfoque modular, su ligereza y

su filosofía de accesibilidad lo han convertido en una alternativa cada vez más popular frente a soluciones comerciales de gran escala como Unity o Unreal Engine.

Desde su publicación como proyecto de código abierto en 2014, Godot ha experimentado un crecimiento sostenido gracias a una comunidad activa y a un modelo de desarrollo transparente. Esta característica no solo permite que cualquier desarrollador examine, modifique y adapte el código fuente, sino que también garantiza que las mejoras y correcciones sean fruto de una colaboración continua.

En el contexto de *Set Wars: Cardinal Bounds*, Godot ha proporcionado una base sólida para desarrollar un videojuego en 2D con mecánicas complejas, manteniendo una estructura clara y escalable.

2.4.1. Orígenes y contexto de Godot

Godot fue creado inicialmente por los desarrolladores argentinos Juan Linietsky y Ariel Manzur, quienes comenzaron su desarrollo en 2007 como una herramienta interna para sus propios proyectos. En 2014 fue lanzado Godot 1.0 en GitHub bajo licencia MIT.

A diferencia de otros motores cuyo desarrollo está fuertemente dirigido por intereses corporativos, Godot se rige por un modelo comunitario: las decisiones de diseño y las nuevas funcionalidades suelen debatirse de forma abierta en foros, repositorios y sistemas de propuestas públicos. Este enfoque favorece la inclusión de características solicitadas por desarrolladores independientes y estudios pequeños.

2.4.2. Lenguajes de programación soportados

Godot ofrece soporte para varios lenguajes de programación, entre los cuales destacan:

- **GDScript**: Lenguaje propio del motor, con sintaxis inspirada en Python, diseñado para integrarse de forma nativa con todas las características de Godot. Su ejecución es interpretada, y su estrecha relación con el motor lo convierte en la opción más directa para *scripts* rápidos y prototipado.

- **C#**: Lenguaje de tipado fuerte basado en la plataforma .NET, que ofrece mayor rendimiento en operaciones complejas y una sintaxis ampliamente conocida en la industria

de los videojuegos. Godot integra C# a través de *Mono*, lo que permite aprovechar bibliotecas externas y patrones de programación avanzados.

En este proyecto se ha optado por utilizar C#, dado que facilita la escritura de código más estructurado y robusto, con un control más estricto de tipos y una gestión más predecible de errores. Esta elección es especialmente beneficiosa en un videojuego que combina mecánicas matemáticas precisas con lógica de juego compleja.

No obstante, ciertas partes del desarrollo requieren el uso del lenguaje de *shaders* de Godot para implementar efectos visuales avanzados, ya que esta funcionalidad no está disponible con C# ni con GDScript. Se llama **Godot Shader Language** y su sintaxis es similar a GLSL, pero adaptada a las particularidades del motor.

2.4.3. Comparativa con Unity

Aunque Unity y Godot comparten el objetivo general de proporcionar un entorno integral para el desarrollo de videojuegos, existen diferencias significativas en su filosofía y funcionamiento:

- **Licencia y coste**: Unity requiere un modelo de suscripción o pago según ingresos, mientras que Godot es completamente gratuito y de código abierto.
- **Estructura del proyecto**: Unity organiza sus elementos en escenas y *GameObjects*, mientras que Godot utiliza una estructura basada en nodos, donde cada nodo cumple un rol específico.
- **Peso y rendimiento**: Godot es considerablemente más ligero en términos de instalación y recursos, lo que facilita su ejecución en equipos menos potentes.
- **Lenguajes soportados**: Unity utiliza principalmente C#, mientras que Godot ofrece múltiples opciones, incluyendo su propio lenguaje.
- **Documentación**: Godot cuenta con una completa documentación online disponible, bastante limpia y ordenada, lo que facilita su uso y la búsqueda de información. Además, la documentación es editable por cualquiera que quiera contribuir, está disponible en varios idiomas y se puede consultar desde el propio motor sin necesidad de acceder a la página web [7].

- **Patrones de diseño**: Unity favorece el uso de componentes y *scripts* asociados a objetos, siguiendo el patrón de entidad-componente. Godot, en cambio, promueve la composición mediante nodos, facilitando la reutilización y modularidad del código. Una característica distintiva de Godot es que los *scripts* se asignan directamente a nodos existentes, y estos heredan de otros, permitiendo crear jerarquías de comportamiento y lógica. Así, cada nodo puede tener su propio script, y la herencia entre *scripts* permite extender o especializar funcionalidades de manera clara y estructurada, aprovechando la flexibilidad del sistema de nodos.

2.4.4. Sistema de nodos y herencia

La estructura de Godot se basa en la composición jerárquica de **nodos**. El árbol de nodos organiza la escena, permitiendo que cada nodo cumpla una función específica y se relacione con otros nodos como parte de una jerarquía visual y funcional. De este modo, es posible construir escenas complejas a partir de la combinación y anidación de nodos más simples.

Además, Godot introduce el concepto de ***PackedScene***, que permite guardar una escena (compuesta por uno o varios nodos y su configuración) como un recurso reutilizable. Este sistema facilita la creación de plantillas (*prefabs* en Unity) que pueden ser instanciadas múltiples veces dentro de otras escenas. Es importante destacar que, al instanciar una *PackedScene*, no se crea una nueva escena independiente, sino que se generan nuevos nodos hijos que se añaden al árbol de nodos de la escena principal en tiempo de ejecución. Esto permite reutilizar estructuras complejas y gestionar dinámicamente la aparición de elementos en el juego, manteniendo la flexibilidad y modularidad del proyecto.

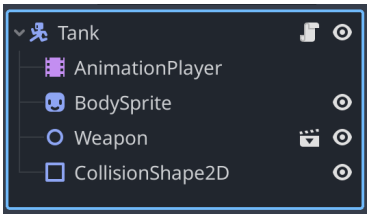


Figura 9: Ejemplo de árbol de jerarquía de nodos en Godot.

Por otro lado, cada nodo pertenece a una jerarquía de clases de herencia. Esto significa que los diferentes tipos de nodos (como *Sprite2D*, *CollisionShape2D*, *AudioStreamPlayer*, etc.)

heredan propiedades y comportamientos de clases base. Así, un nodo derivado incorpora automáticamente las características y métodos definidos en las clases superiores de la jerarquía de herencia, independientemente de su posición en el árbol de nodos de la escena.

Además, mediante scripting, Godot permite extender esta herencia: puedes asociar *scripts* a los nodos para crear nuevas clases personalizadas, añadiendo o sobrescribiendo funcionalidades según las necesidades del proyecto.

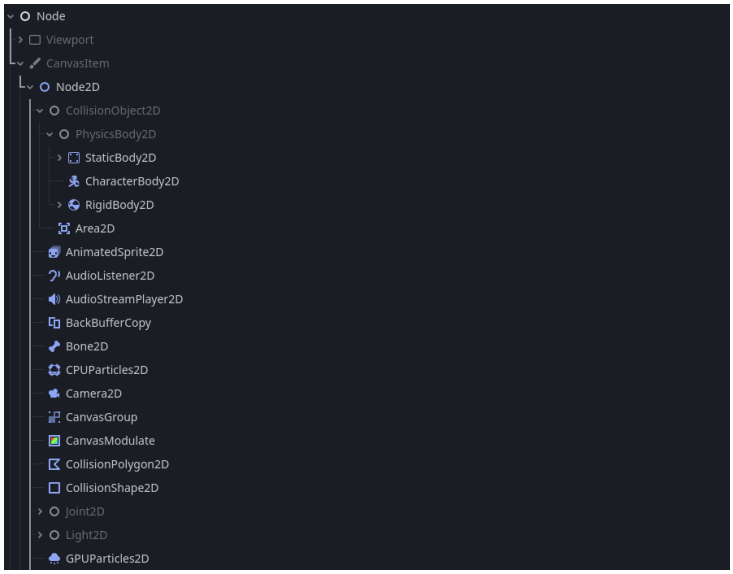


Figura 10: Parte del árbol de herencia de nodos de Godot.

2.4.5. *Signals* y comunicación entre nodos

Godot implementa un sistema de comunicación entre nodos basado en ***signals*** (señales), que permite a los nodos emitir eventos y a otros nodos suscribirse para recibir notificaciones cuando dichos eventos ocurren. Este mecanismo facilita la interacción entre diferentes partes del juego sin necesidad de acoplar directamente los nodos, promoviendo un diseño más modular y flexible.

Cuando un nodo emite una señal, todos los nodos que están conectados a esa señal reciben una notificación y pueden ejecutar una función específica en respuesta. Esta conexión se puede establecer tanto en el editor de Godot como mediante código, lo que permite una gran versatilidad en la gestión de eventos.

En el caso de C#, las señales se manejan mediante delegados y eventos, lo que proporciona una sintaxis familiar para los desarrolladores acostumbrados a este lenguaje. La integración

de señales en C# permite aprovechar las ventajas del sistema de eventos propio del lenguaje, facilitando la comunicación entre nodos de manera eficiente y estructurada.

En este proyecto, como es evidente, el uso de señales ha sido fundamental para gestionar eventos, manteniendo el código limpio y evitando dependencias innecesarias entre los diferentes componentes del juego.

2.4.6. Funciones base nativas de los nodos

Godot ofrece una serie de funciones base que pueden ser sobrescritas en los *scripts* asociados a los nodos para definir comportamientos específicos. Entre las más relevantes se encuentran:

- **_EnterTree()**: Se llama cuando el nodo es añadido al árbol de nodos, antes de que sus nodos hijos sean añadidos. Es útil para configuraciones que deben realizarse antes de que el nodo esté completamente activo.

- **_Ready()**: Se llama cuando el nodo y todos sus nodos hijos han sido añadidos al árbol de nodos. Es el lugar ideal para inicializar variables, configurar estados iniciales o establecer conexiones de señales.

- **_Process(delta)**: Se ejecuta en cada fotograma (*frame*) del juego, recibiendo como parámetro el tiempo transcurrido desde el último fotograma (*delta time*). Es adecuado para actualizar la lógica del juego que debe evaluarse constantemente, como movimientos o verificaciones de estado.

- **_PhysicsProcess(delta)**: Similar a _Process, pero se llama en intervalos fijos, sincronizados con el motor de físicas. Es ideal para manejar la lógica relacionada con colisiones o movimientos basados en fuerzas.

- **_Input(event)**: Se invoca cada vez que se recibe un evento de entrada (teclado, ratón, mando, etc.). Permite gestionar la interacción del usuario de manera directa y personalizada. Se recomienda utilizarse para manejar *inputs* relacionados con interfaces de usuario.

- **_UnhandledInput(event)**: Similar a _Input, pero se llama solo si el evento no ha sido consumido por otros nodos. Se recomienda utilizarse para manejar *inputs* que no sean para interfaces de usuario, es decir, para la lógica dentro del *gameplay*.
- **_ExitTree()**: Se ejecuta cuando el nodo es removido del árbol de nodos. Es un buen lugar para limpiar recursos o desconectar señales.
- **_Notification(what)**: Se llama para notificar al nodo sobre diversos eventos del motor, como cambios de visibilidad o modificaciones en el árbol de nodos. El parámetro *what* indica el tipo de notificación recibida.

Existen otras funciones base adicionales, pero las mencionadas son las que se han utilizado en este proyecto.

2.4.7. Versión 4.2 y migración a 4.4.1

El desarrollo de *Set Wars: Cardinal Bounds* comenzó en Godot 4.2, una versión estable que incorporaba mejoras significativas respecto a la serie 3.x, especialmente en rendimiento 2D, herramientas de depuración y flexibilidad en el sistema de nodos.

Durante el transcurso del proyecto, se optó por migrar a la versión 4.4.1. Esta actualización proporcionó:

- Mejoras de rendimiento tanto en renderizado como en ejecución de *scripts*.
- Nuevas funcionalidades que facilitaron la implementación de mecánicas y la optimización de recursos.
- Agregación de que cuando ejecutas el juego en modo debug este se encuentre *embedded* a la ventana del motor.

El cambio de versión supuso un proceso de adaptación mínimo, gracias a la compatibilidad mantenida entre ambas versiones y a la planificación modular del proyecto desde su inicio.

2.5. Serious Games

Los juegos serios (*serious games*) son aplicaciones interactivas que, más allá del entretenimiento, tienen como objetivo principal la educación, formación o concienciación sobre un

tema específico [8, 9, 10]. Estos juegos mantienen los elementos de diversión y compromiso característicos de los videojuegos, pero están diseñados intencionalmente para cumplir objetivos pedagógicos concretos.

2.5.1. Características y valor educativo

Las características principales de los serious games incluyen:

- **Objetivos educativos explícitos:** Diseñados para transmitir conocimientos o habilidades específicas.
- **Retroalimentación inmediata:** Permiten a los usuarios aprender de sus errores en tiempo real.
- **Progresión adaptativa:** La dificultad se ajusta gradualmente para mantener el desafío y el interés.
- **Motivación intrínseca:** Utilizan mecánicas de juego para mantener el compromiso del usuario.

Boyle et al. [11] demostraron que los juegos educativos presentan ventajas significativas en entornos de aprendizaje, incluyendo mayor retención de conocimiento, desarrollo de habilidades cognitivas y aumento de la motivación del estudiante.

2.5.2. Aplicación en el aprendizaje matemático

En el ámbito matemático, los serious games han demostrado ser particularmente efectivos en un contexto donde las matemáticas representan una de las asignaturas más problemáticas para los estudiantes. Según el informe PISA 2022, España se sitúa por debajo de la media de la OCDE en competencia matemática, y aproximadamente un 25 % de los estudiantes españoles no alcanza el nivel básico de competencia en esta materia [12].

En este contexto, los serious games ofrecen ventajas sustanciales:

- **Transformación de conceptos abstractos:** Convierten principios matemáticos teóricos y aparentemente desconectados de la realidad en experiencias visuales e interactivas tangibles, permitiendo a los estudiantes manipular y experimentar con ideas que de otro modo resultarían difíciles de visualizar o comprender.

- **Reducción de la ansiedad matemática:** Crean un entorno seguro donde el error forma parte natural del proceso de aprendizaje, eliminando la presión asociada a las evaluaciones tradicionales y permitiendo que los estudiantes afronten los desafíos matemáticos sin el miedo al fracaso que suele generar bloqueos cognitivos.

- **Práctica significativa:** Facilitan la repetición necesaria para consolidar conceptos matemáticos sin caer en la monotonía de los ejercicios tradicionales, integrando la práctica en contextos narrativos o competitivos que mantienen el interés y la motivación a lo largo del tiempo.

- **Desarrollo del pensamiento crítico:** Fomentan habilidades de análisis, síntesis y resolución de problemas en situaciones que requieren aplicar conocimientos matemáticos de forma creativa, preparando a los estudiantes para enfrentarse a problemas del mundo real que no siempre siguen patrones predefinidos.

2.5.3. Set Wars como serious game

Set Wars: Cardinal Bounds se enmarca en esta categoría al combinar la acción frenética de un space shooter 2D con objetivos educativos relacionados con la teoría de conjuntos. A diferencia de una simple capa temática, el diseño fundamental del juego:

- **Integra conceptos matemáticos en la mecánica central:** Las operaciones con conjuntos como intersección, unión y complemento no son simplemente mencionadas, sino que constituyen el núcleo de la jugabilidad, donde el jugador debe aplicar estos conceptos en tiempo real para superar los desafíos que se le presentan durante las batallas espaciales.

- **Estimula el razonamiento matemático intuitivo:** A través de representaciones visuales dinámicas y decisiones rápidas, el juego desarrolla una comprensión intuitiva de las relaciones entre conjuntos, promoviendo una forma de pensamiento lógico-matemático que trasciende la memorización de definiciones.

- **Implementa una curva de aprendizaje cuidadosamente diseñada:** Los niveles y desafíos están estructurados para introducir gradualmente conceptos más complejos,

permitiendo que el jugador construya su comprensión sobre bases sólidas antes de enfrentarse a situaciones que requieren un dominio más profundo de la teoría.

- **Crea un contexto narrativo significativo:** Los elementos matemáticos están integrados en un universo coherente donde tienen sentido dentro de la historia y objetivos del juego, evitando la sensación de estar frente a un «ejercicio disfrazado» y generando un entorno donde el aprendizaje surge como consecuencia natural de la inmersión en el juego.

La fusión entre la intensidad de un shooter espacial y los fundamentos matemáticos de la teoría de conjuntos permite que *Set Wars: Cardinal Bounds* trascienda la dicotomía entre entretenimiento y educación, demostrando que ambos aspectos pueden reforzarse mutuamente cuando están diseñados como partes integradas de una misma experiencia interactiva.

2.6. Metodología Scrum

Scrum es un marco de trabajo ágil que permite abordar problemas complejos adaptativos, a la vez que se entregan productos de forma eficiente y creativa con el máximo valor posible [13]. A diferencia de metodologías tradicionales basadas en planificaciones extensas y documentación exhaustiva, Scrum se centra en la entrega iterativa e incremental, priorizando la adaptación al cambio y la retroalimentación continua.

2.6.1. Fundamentos de Scrum

La metodología Scrum se fundamenta en tres pilares esenciales [14]:

- **Transparencia:** Todos los aspectos significativos del proceso son visibles para quienes son responsables del resultado. La transparencia se manifiesta en un lenguaje común para referirse al proceso y en la definición compartida de «terminado».

- **Inspección:** Los usuarios de Scrum deben inspeccionar frecuentemente los artefactos y el progreso hacia los objetivos para detectar variaciones indeseadas. Esta inspección no debe ser tan frecuente que interfiera con el trabajo.

- **Adaptación:** Si se determina que uno o más aspectos del proceso se desvían de los límites aceptables, el proceso o el material que se está procesando debe ajustarse lo antes posible para minimizar desviaciones adicionales.

2.6.2. Roles en Scrum

El equipo Scrum consta de tres roles principales [15]:

- **Product Owner:** Responsable de maximizar el valor del producto resultante del trabajo del equipo de desarrollo. Gestiona el Product Backlog (sección 2.6.4), asegurando que sea claro y refleje las necesidades del cliente.
- **Scrum Master:** Facilitador que asegura que el equipo siga las prácticas y reglas de Scrum. Ayuda a eliminar impedimentos y facilita eventos según sea necesario.
- **Equipo de Desarrollo:** Profesionales que realizan el trabajo de entregar un incremento de producto «terminado» al final de cada Sprint. Son autoorganizados y multifuncionales.

2.6.3. Eventos de Scrum

Scrum define varios eventos formales para inspección y adaptación [14]:

- **Sprint:** Bloque de tiempo (generalmente de 1 a 4 semanas) durante el cual se crea un incremento de producto «terminado» utilizable y potencialmente desplegable.
- **Sprint Planning:** Reunión donde se planifica el trabajo a realizar durante el Sprint actual.
- **Daily Scrum:** Evento diario de 15 minutos donde el equipo de desarrollo sincroniza actividades y crea un plan para las próximas 24 horas.
- **Sprint Review:** Se lleva a cabo al final del Sprint para inspeccionar el incremento y adaptar el Product Backlog si es necesario.
- **Sprint Retrospective:** Oportunidad para el equipo de inspeccionarse a sí mismo y crear un plan de mejoras para el próximo Sprint.

2.6.4. Artefactos de Scrum

Los artefactos de Scrum representan trabajo o valor y están diseñados para maximizar la transparencia [15]:

- **Product Backlog:** Lista ordenada de todo lo que podría ser necesario en el producto, única fuente de requisitos para cualquier cambio a realizarse.
- **Sprint Backlog:** Conjunto de elementos del Product Backlog seleccionados para el Sprint, más un plan para entregar el incremento de producto y conseguir el objetivo del Sprint.
- **Incremento:** Suma de todos los elementos del Product Backlog completados durante un Sprint y el valor de los incrementos de todos los Sprints anteriores.

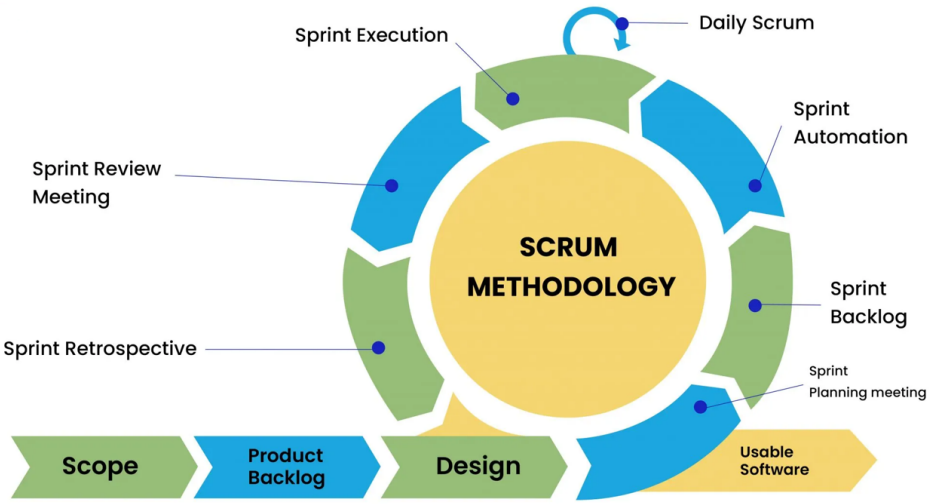


Figura 11: Esquema de la metodología Scrum [13].

2.7. Game Experience Questionnaire (GEQ)

Con el objetivo de evaluar la calidad de la experiencia de juego una vez alcanzado un prototipo funcional, y tras permitir que dos *testers* probaran el juego, se optó por emplear el Game Experience Questionnaire (GEQ), desarrollado por IJsselsteijn, de Kort y Poels en la Universidad Tecnológica de Eindhoven [16]. Este cuestionario es ampliamente empleado en la

investigación de videojuegos y ha sido diseñado específicamente para medir distintos aspectos de la experiencia del jugador mediante escalas estandarizadas.

El GEQ está organizado en módulos; el *Core Module* es el más utilizado en pruebas de juegos individuales y permite evaluar dimensiones clave de la experiencia lúdica, tales como:

- **Competence**: percepción de habilidad y dominio del juego.
- **Sensory and Imaginative Immersion**: grado de inmersión sensorial y narrativa.
- **Flow**: nivel de concentración y absorción en la actividad.
- **Tension/Annoyance**: frustración o incomodidad experimentada.
- **Challenge**: percepción de dificultad en el juego.
- **Positive Affect** y **Negative Affect**: emociones positivas y negativas derivadas de la sesión.

Gracias a estas dimensiones, el GEQ permite obtener un perfil detallado de la experiencia del jugador, combinando aspectos emocionales, cognitivos y de desempeño. Su amplia adopción en estudios académicos lo convierte en una herramienta válida y reconocida para investigaciones sobre diseño y evaluación de videojuegos.

La aplicación concreta del GEQ y el análisis de los resultados obtenidos en las pruebas realizadas con *Set Wars: Cardinal Bounds* se detallan en el Capítulo 5.

3

Análisis y Diseño de la Solución

En este capítulo se presentan los aspectos fundamentales del análisis y diseño del proyecto. En primer lugar, se describe la metodología de desarrollo adaptada, basada en principios ágiles de Scrum modificados para un proyecto individual, detallando la organización en sprints y la planificación temporal mediante un diagrama de Gantt. A continuación, se expone la evolución conceptual del proyecto, desde una idea inicial más amplia hasta el enfoque específico que finalmente definió el juego.

El capítulo continúa con un análisis detallado de requisitos, tanto funcionales como no funcionales, que establecen las características esenciales del videojuego. Seguido de diversos diagramas que ilustran el flujo de las interfaces, la dinámica de juego y el comportamiento de los enemigos.

Finalmente, se muestran los bocetos iniciales de interfaces y personajes, así como las decisiones tomadas respecto a la dirección artística y diseño sonoro que definen la estética del juego y su accesibilidad. El capítulo concluye con una descripción de la arquitectura general en la que se ha construido todo el proyecto.

3.1. Metodología aplicada

Para la gestión del proyecto Set Wars, adopté la metodología ágil Scrum adaptada al contexto de un desarrollo individual. Si bien Scrum está diseñado originalmente para equipos, sus principios fundamentales de desarrollo iterativo, entregables frecuentes y adaptabilidad resultaron valiosos incluso para un proyecto unipersonal.

Implementé una versión personalizada de Scrum, manteniendo la esencia de sus elementos clave. Establecí sprints de 1-3 semanas de duración, adaptando los tiempos según la compleji-

dad de las funcionalidades y mi disponibilidad entre otras obligaciones académicas y laborales.

Al inicio de cada sprint, realizaba mi propia versión de la planificación, reflexionando sobre qué aspectos del juego eran prioritarios y estableciendo objetivos concretos y alcanzables para ese periodo. Aunque no contaba con un equipo para las reuniones diarias tradicionales de Scrum, mantenía un seguimiento personal mediante listas de tareas para cada sprint.

Esta adaptación de Scrum me proporcionó estructura sin sacrificar flexibilidad, permitiéndome visualizar el progreso, ajustar prioridades cuando era necesario y, lo más importante, mantener el objetivo de tener versiones jugables a partir de cierto punto al final de cada sprint para probar las mecánicas implementadas y responder ágilmente a los cambios requeridos.

Cada ciertos periodos se realizaban reuniones con el tutor para que observase el progreso y pudiese valorar los resultados que había hasta el momento.

3.1.1. Sprints

Para organizar el desarrollo, implementé una estructura de sprints de una a tres semanas de duración, siguiendo los principios de Scrum. Esta planificación permitió mantener un ritmo constante de desarrollo y obtener incrementos funcionales que pudieran evaluarse periódicamente. A continuación, se detalla el contenido y objetivos de cada uno de los sprints realizados:

- **Sprint 0 - Fase preparatoria:** Dedicado a la formalización y aprobación de la propuesta con el tutor. Este sprint sentó las bases conceptuales y técnicas para el desarrollo. También incluye la familiarización con las tecnologías seleccionadas, aprovechando el conocimiento previo que ya se tenía de ellas y profundizando en aspectos específicos necesarios para este proyecto.

- **Sprint 1 - Conceptualización:** Elaboración del documento High Concept y Game Design Document (GDD), definiendo las mecánicas principales y estructura del juego. También se realizaron bocetos de las interfaces y algunos elementos como los personajes. Recopilación de canciones, sonidos y fuentes de texto.

- **Sprint 2 - Planificación técnica:** Análisis detallado de requisitos funcionales y no funcionales. Elaboración de diagramas de clase para definir la arquitectura del software

y diagramas de flujo para la navegación por las interfaces y comportamiento de los enemigos.

- **Sprint 3 - Primera iteración de la implementación:** Creación del proyecto base en Godot, configuración del repositorio y establecimiento de la estructura fundamental del proyecto. Desarrollo de la mecánicas del juego, incluyendo el personaje del jugador y sus controles, salud, proyectiles, enemigos, límites del mapa, *background*, etc. También se crearon los diseños artísticos de los elementos que se iban requiriendo.

- **Sprint 4 - Segunda iteración de la implementación:** Enfoque mayor en el desarrollo de las interfaces del juego (menús y HUD[3]) y de los principales *módulos* del juego, como la instanciación/eliminación de interfaces y niveles, el idioma, los controles (mando y teclado), el sistema de guardado, los ajustes, el sonido, la pausa, etc. Se crearon todos los diseños necesarios artísticos para las interfaces que fueran necesarios.

- **Sprint 5 - Tercera iteración de la implementación:** Enfoque mayor en la creación de los niveles del juego una vez se tenían el resto de elementos para montarlos como si fuesen piezas. También se crearon *triggers*[4] a medida que se iban necesitando en los niveles.

- **Sprint 6 - Pruebas y ajustes:** Fase intensiva de pruebas para identificar y corregir errores. Se dió a probar a algunas personas el juego para encontrar los aspectos que se podían mejorar y cuáles eran acertados. Tras ello se hicieron los cambios necesarios finales.

- **Sprint 7 - Documentación:** Desarrollo de la memoria del proyecto, documentando tanto el proceso de creación como los aspectos técnicos y de diseño más relevantes.

- **Sprint 8 - Finalización:** Finalización de la memoria y preparación de la presentación del proyecto. Revisión y últimos ajustes posibles.

[3]El HUD (*Head-UP Display*) es una interfaz gráfica que muestra información relevante al jugador en tiempo real, como la salud, munición y otros datos importantes.

[4]Los *triggers* son eventos en el juego que activan ciertas acciones o comportamientos y que se activan de diversas formas. Son un sistema de comandos internos dentro de un juego para gestionar la lógica del mismo.

3.1.2. Diagrama de Gantt

Acompañado de los sprints, se elaboró un *diagrama de Gantt* para visualizar la planificación temporal del proyecto. El cual permitió identificar las fases de desarrollo, los hitos importantes y las dependencias entre tareas, facilitando una gestión más efectiva del tiempo y los recursos disponibles. Cabe destacar que el diagrama presenta tres fases, dividiendo los sprints por grupos:

- **Fase de Conceptualización:** Comprende del sprint 0 al 2, en esta fase se establecen las bases conceptuales y técnicas del proyecto. Se crearon bocetos, documentación del juego, diagramas y todo lo necesario para poder comenzar con su implementación.

- **Fase de Implementación:** Comprende del sprint 3 al 5, esta fase se centra en la creación y desarrollo del juego. Se implementan las mecánicas, interfaces y niveles, construyendo el juego de manera iterativa y asegurando que cada componente funcione correctamente. Es la fase más extensa.

- **Fase de Evaluación:** Comprende del sprint 6 al 8, esta fase se dedica a la revisión, pruebas y documentación posterior del proyecto. Se identifican y corrigen errores, se realizan ajustes basados en el feedback recibi, creado con y se documenta todo el proceso de desarrollo para futuras referencias.

Este es el diagrama, creado con la herramienta *GanttProject*:

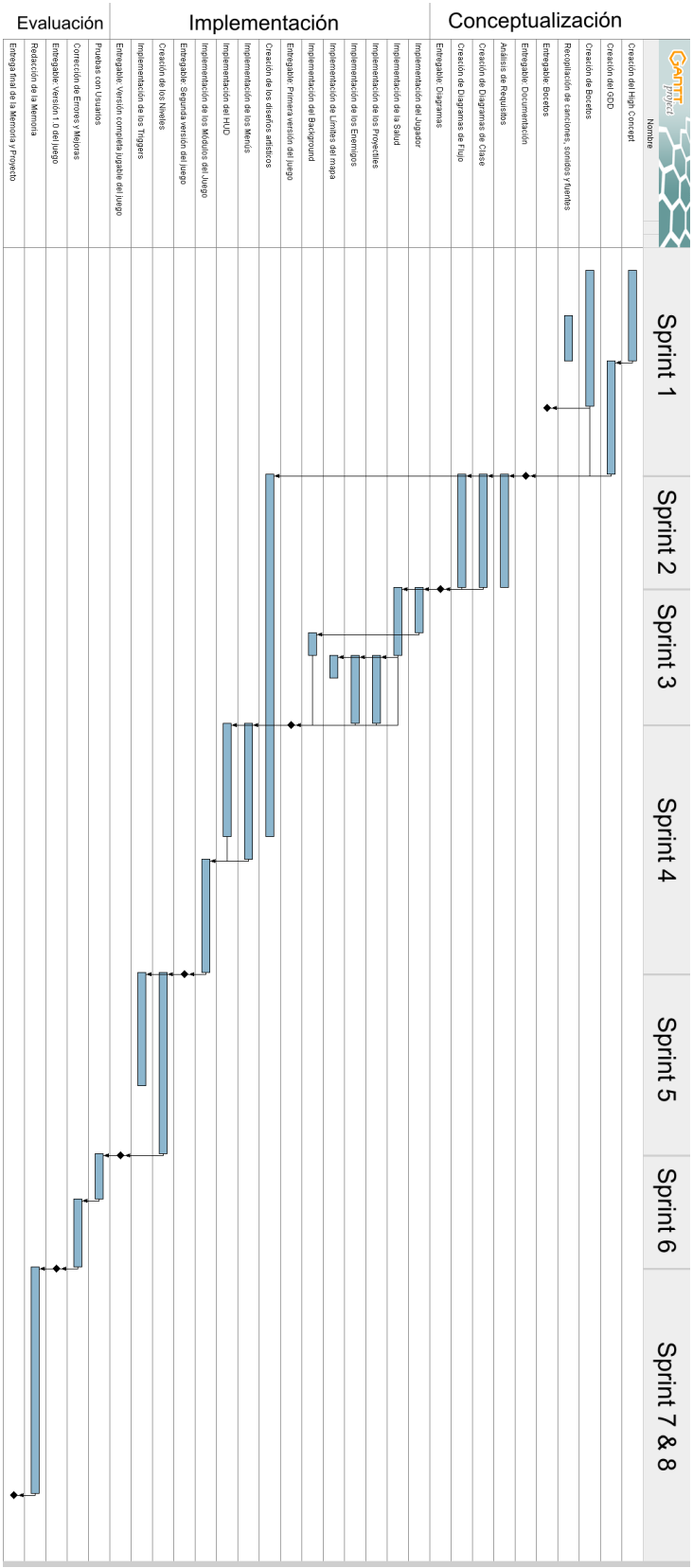


Figura 12: Diagrama de Gantt de la planificación del proyecto.

3.2. Primer concepto

La primera idea para el proyecto consistía en crear un videojuego educativo que abarcara una gran variedad de ramas de las matemáticas. El planteamiento inicial era un juego de plataformas en el que se buscaba aplicar distintos conceptos de forma práctica, ofreciendo al jugador una experiencia diversa y amplia.

No obstante, este enfoque resultaba demasiado ambicioso para los recursos y tiempo disponibles. El desarrollo de un proyecto de tal magnitud era inabarcable, y reducirlo sin una dirección clara habría restado calidad a la experiencia. Por ello, se tomó la decisión de orientar el diseño hacia una propuesta más contenida, que permitiera pulir la jugabilidad y profundizar mejor en los conceptos elegidos, aunque fueran menos en número.

Así nació *Set Wars*, centrado exclusivamente en la teoría de conjuntos. El cambio de género hacia un *shooter* espacial respondió también a esta nueva dirección: se buscaba un entorno y unas mecánicas diferentes que permitieran adaptar de manera más orgánica la enseñanza de los conceptos matemáticos de solo esta rama a un contexto propio y coherente. De esta forma, el juego mantiene la esencia educativa del concepto original, pero con un enfoque mucho más acotado y sólido.

3.3. Requisitos

A partir del concepto y de los objetivos definidos para Set Wars, se establecieron los requisitos que debía cumplir el proyecto. Estos requisitos se organizan en dos categorías principales: por un lado, los requisitos funcionales, que recogen las características y comportamientos específicos que el videojuego debe ofrecer al jugador; y por otro, los requisitos no funcionales, que establecen aspectos relacionados con la calidad, el rendimiento, la usabilidad y otros criterios generales que afectan al sistema en su conjunto.

3.3.1. Requisitos funcionales

RF1: Inicio de partida y selección de nivel: El juego debe permitir al jugador elegir y repetir cualquier nivel que esté desbloqueado.

RF1.1: El menú de selección de nivel debe mostrar todos los niveles disponibles, indi-

cando cuáles están completados, desbloqueados y bloqueados.

RF2: Progreso y desbloqueo de niveles: El sistema debe gestionar el avance del jugador desbloqueando niveles conforme se completen.

RF2.1: Al completar un nivel, el sistema debe desbloquear automáticamente el siguiente nivel.

RF2.2: El progreso debe guardarse automáticamente sin intervención del jugador.

RF3: Pausa y reanudación de partida: El jugador debe poder pausar la partida en cualquier momento y retomarla después.

RF3.1: El menú de pausa debe permitir reanudar la partida.

RF3.2: El menú de pausa debe incluir acceso a ajustes y la opción de abandonar el nivel.

RF4: Menús contextuales: El juego debe disponer de diferentes menús para facilitar la navegación.

RF4.1: Menú principal.

RF4.2: Pantalla de selección de nivel.

RF4.3: Menú de pausa.

RF4.4: Menú de ajustes.

RF4.5: Pantalla de créditos.

RF4.6: Pantalla de nivel completado.

RF4.7: Pantalla de *Game Over*.

RF5: Sistema HUD (interfaz en partida, *Head-Up Display*): Durante la partida, el juego debe mostrar en pantalla la información esencial para el jugador.

RF5.1: Salud actual del jugador.

RF5.2: Proyectiles y colores disponibles y cuál se tiene seleccionado.

RF5.3: Diálogos y mensajes del juego cuando deban aparecer.

RF5.4: Leyendas de ayuda sobre controles cuando deban aparecer.

RF6: Sistema de salud: La salud del jugador y de los enemigos se representa mediante un conjunto de corazones de distintos colores. Cada color representa un elemento único del conjunto, por lo que no pueden repetirse colores en la salud.

- **RF6.1:** Si un jugador o enemigo pierde todos los corazones de su conjunto, se considera que muere. En el caso del jugador, esto provoca el *Game Over* del nivel y obliga a reiniciarlo desde el inicio.
- **RF6.2:** El estado de salud puede modificarse mediante proyectiles disparados por el jugador o los enemigos, los cuales aplican operaciones de teoría de conjuntos que pueden eliminar o restaurar corazones según sus colores.
- **RF6.3:** Los enemigos utilizan colores distintos a los del jugador, para diferenciar claramente la salud y proyectiles.

RF7: Disparos y operaciones de conjuntos: El sistema de combate debe implementar la mecánica principal basada en la teoría de conjuntos.

- **RF7.1:** El jugador puede disparar diferentes tipos de proyectiles, cada uno representando una operación de conjuntos.
- **RF7.2:** Cada proyectil debe aplicarse con un color determinado, el cual se corresponde con el color del corazón que "llevan".
- **RF7.3:** El efecto de un disparo sobre un objetivo depende de la relación entre el color del proyectil, el tipo de este y la salud del objetivo en ese momento.
- **RF7.4:** Los enemigos también deben poder disparar proyectiles hacia el jugador.
- **RF7.5:** El jugador debe poder cambiar entre los distintos tipos y colores de proyectiles disponibles que tenga en ese nivel.
- **RF7.6:** Los proyectiles del jugador no pueden impactar en él mismo. De manera análoga pasa entre los enemigos, pero no se pueden hacer daño entre ellos.

RF8: Configuración de usuario: El juego debe ofrecer un menú de configuración que permita al jugador ajustar la experiencia de juego.

- **RF8.1:** Ajuste de volumen de música y efectos de sonido.

RF8.2: Selección de idioma.

RF8.3: Ajuste de sensibilidad de la mirilla al usar mando.

RF8.4: Configuración de la ventana del programa: completa, completa sin bordes y ventana.

RF9: Soporte multilenguaje: Todos los textos del juego deben estar disponibles en dos idiomas: español e inglés. Mostrandose solo el que se tenga seleccionado desde los ajustes.

RF10: Tutorial interactivo: El juego debe contar con un sistema de tutoriales integrados para la enseñanza progresiva.

RF10.1: Los niveles deben introducir nuevas mecánicas, controles y enemigos de forma progresiva.

RF10.2: El tutorial debe mostrar indicaciones claras mediante HUD y diálogos.

RF11: Sistema de diálogos: El juego debe incluir un sistema de diálogos que se active en determinados eventos o momentos de los niveles.

RF11.1: Los diálogos deben presentarse de manera clara y legible en pantalla.

RF11.2: El jugador debe poder avanzar en los diálogos mediante una acción específica (pulsar una tecla o botón) y en otros casos por eventos específicos.

RF12: Guardado automático: El sistema debe guardar automáticamente el progreso y configuraciones al completar niveles o modificar ajustes.

3.3.2. Requisitos no funcionales

RNF1: Portabilidad: El juego debe poder ejecutarse correctamente en sistemas operativos Windows y Linux. Se estudiará la posibilidad de soportar macOS.

RNF2: Rendimiento: El sistema debe ofrecer un rendimiento estable y fluido durante la ejecución del juego.

RNF2.1: La tasa de frames por segundo (FPS) no debe ser inferior a 60 en equipos de gama media durante la partida.

RNF2.2: Los tiempos de carga entre escenas no deben superar los 3 segundos, garantizando una experiencia ágil y continua.

RNF3: Usabilidad: La interfaz de usuario y la experiencia de navegación deben ser intuitivas y accesibles para todos los jugadores.

RNF3.1: Todos los menús deben mostrar información y opciones relevantes de manera clara y ordenada.

RNF3.2: La interfaz durante la partida (HUD) debe ser legible, mostrando la información crítica sin obstruir la visión del jugador.

RNF3.3: Los elementos interactivos deben responder correctamente a las acciones del jugador, proporcionando retroalimentación visual o sonora.

RNF4: Escalabilidad: El diseño del juego debe ser modular y flexible, permitiendo futuras ampliaciones sin afectar el núcleo del sistema.

RNF4.1: Será posible añadir nuevos niveles, enemigos, proyectiles u operaciones matemáticas sin modificar la lógica base del juego.

RNF4.2: La incorporación de nuevos idiomas en el sistema multilenguaje no deberá requerir cambios en el código central.

RNF5: Mantenibilidad: El código del juego debe ser fácil de mantener, depurar y ampliar.

RNF5.1: La organización del código debe estar basada en carpetas y componentes coherentes según su función.

RNF5.2: Se deben aplicar patrones de diseño que faciliten la gestión de componentes globales, como *singletons*[5] o gestores de escenas.

RNF5.3: Cada módulo y función debe estar debidamente documentado para permitir futuras ampliaciones o correcciones.

RNF6: Accesibilidad, estética y coherencia visual: El diseño visual debe ser funcional, atractivo y seguro para todo tipo de jugadores.

[5]Un *singleton* es un patrón de diseño que restringe la instanciación de una clase a un único objeto pero permite el acceso global de este, facilitando su uso en diferentes partes del código.

RNF6.1: La paleta de colores del juego se basará en blanco, gris y negro, reservando colores vivos únicamente para proyectiles, corazones y elementos de interfaz.

RNF6.2: Los colores de los proyectiles y corazones estarán diferenciados entre jugador y enemigos para evitar confusiones durante la partida.

RNF6.3: Se evitará el uso de flashes o estímulos visuales que puedan afectar a jugadores epilépticos o con sensibilidad a ciertos colores.

RNF6.4: La coherencia visual debe mantenerse en todos los elementos del juego, incorporando animaciones fluidas y efectos visuales que refuercen la jugabilidad.

RNF7: Seguridad: El sistema debe proteger la integridad de los datos del jugador.

RNF7.1: Los archivos de guardado deben almacenarse en una ubicación oculta del sistema para evitar manipulaciones accidentales.

RNF7.2: Los archivos de guardado no deben ser fácilmente editables por el usuario para prevenir trampas o alteraciones del progreso.

RNF8: Audio: El diseño sonoro debe ser coherente con la temática y configurable por el jugador.

RNF8.1: El juego debe incluir música de fondo y efectos sonoros acordes con la temática espacial y el ritmo de la acción.

RNF8.2: El jugador debe poder ajustar el volumen de la música y efectos, así como desactivarlos completamente si lo desea.

3.4. Diagramas de flujo

La representación gráfica de procesos mediante diagramas de flujo tradicionales constituye una herramienta esencial en el ámbito del desarrollo de software, ya que facilita la visualización de la lógica subyacente y la organización de secuencias complejas sin necesidad de recurrir directamente al código fuente [17, 18].

En este contexto, los diagramas de flujo y, en un sentido más formal, los diagramas de actividad de UML, permiten expresar de manera estandarizada decisiones, bifurcaciones y eventos,

lo cual favorece la comunicación entre desarrolladores y el entendimiento de la dinámica del sistema [19].

Para este proyecto, estos diagramas han resultado particularmente útiles al momento de representar el funcionamiento de las interfaces y el comportamiento de los enemigos, aportando claridad tanto en el diseño como en la posterior implementación [17, 19].

3.4.1. Diagramas de flujo de las interfaces

Se han elaborado dos diagramas para las interfaces. El primero muestra la navegación global entre todos los menús, evidenciando las posibles transiciones y retornos entre ellos. El segundo se centra en el flujo específico del menú de selección de niveles, detallando cómo se gestionan las opciones disponibles.

En conjunto, ambos diagramas ofrecen una visión estructurada del comportamiento de las interfaces, permitiendo anticipar problemas de usabilidad y garantizar una interacción más intuitiva y eficiente para el jugador.

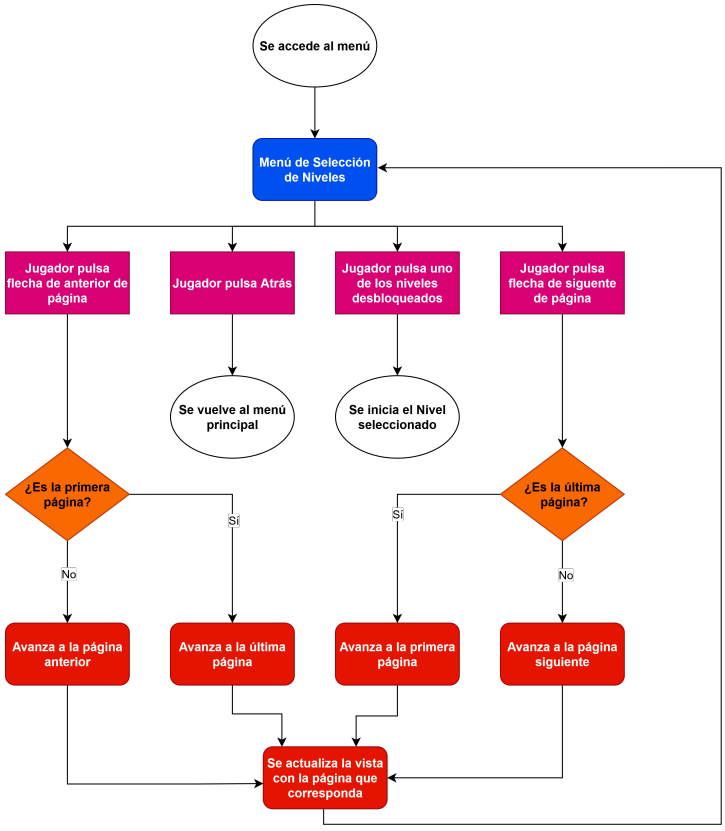


Figura 13: Diagrama de flujo de la navegación en el menú de selección de nivel.

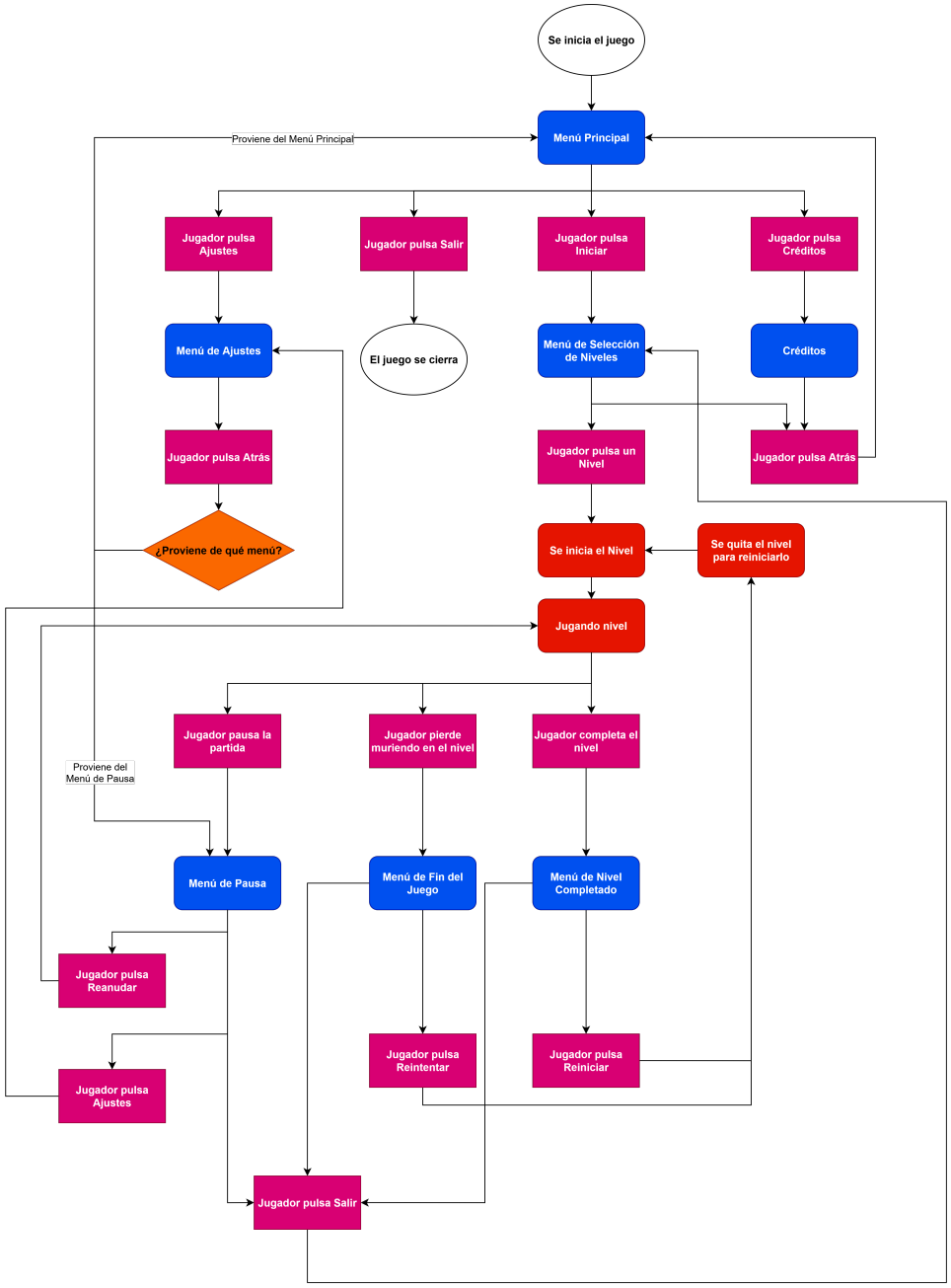


Figura 14: Diagrama de flujo de la navegación global entre todos los menús del juego.

3.4.2. Diagramas de flujo del comportamiento de los enemigos

Todos los enemigos del juego comparten una misma base de comportamiento, definida por un flujo común de acciones. Las diferencias entre ellos se introducen únicamente mediante ajustes de parámetros como la velocidad o el rango de detección, lo que permite dar variedad sin alterar la lógica principal.

Para ilustrar este funcionamiento se ha elaborado un único diagrama, en el que se recogen las fases básicas: detección del jugador, seguimiento, ataque, orbitación y huida. Este esquema refleja la dinámica general a partir de la cual se configuran todos los enemigos (salvo excepciones).

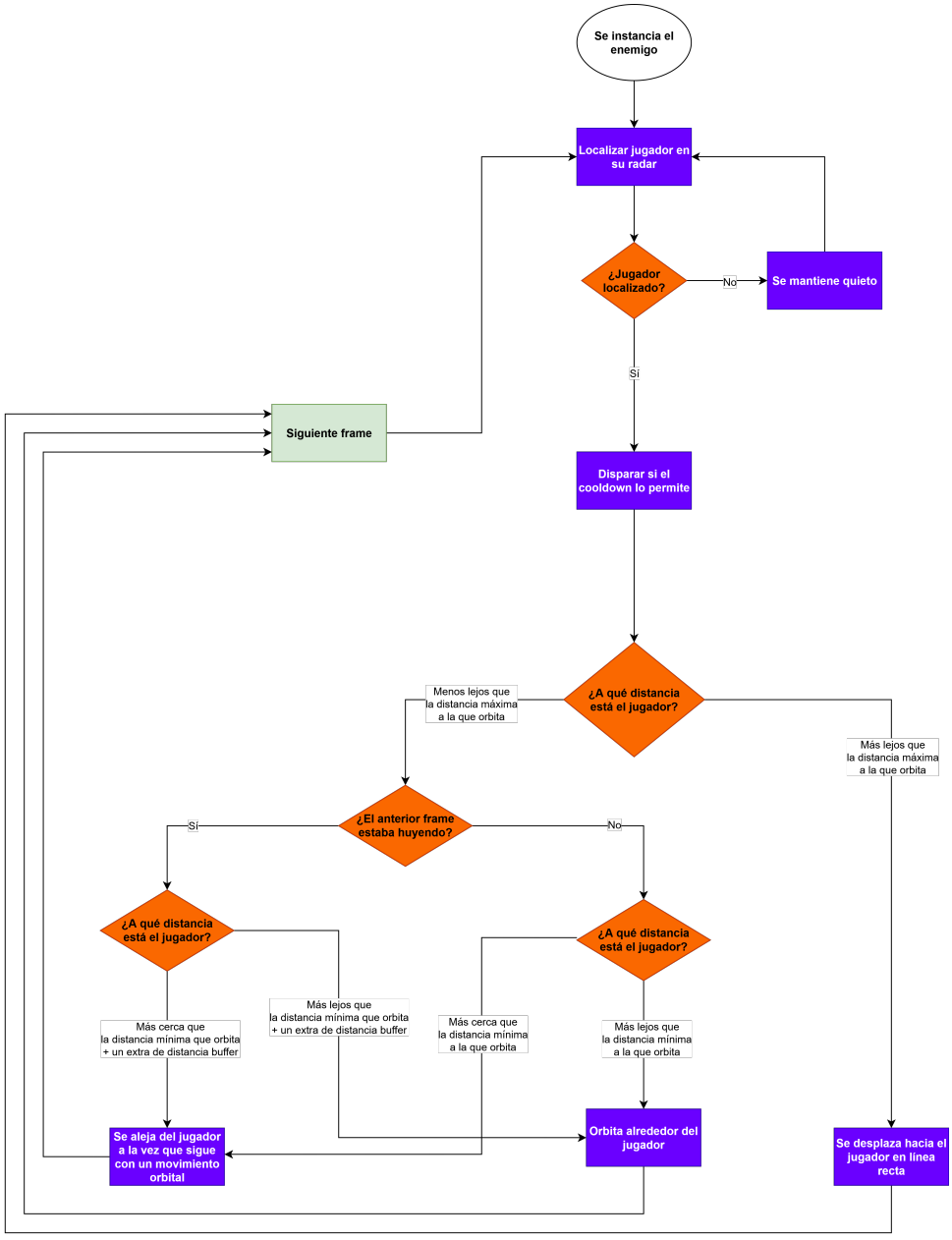


Figura 15: Diagrama de flujo del comportamiento de los enemigos.

3.5. Bocetos de las interfaces

Durante la fase de diseño se realizaron bocetos y prototipos de la interfaz de usuario con el objetivo de organizar los elementos en pantalla de manera lógica y facilitar la comprensión y uso por parte del jugador. Los bocetos sirven principalmente para planificar la estructura y disposición de los elementos, por lo que no reflejan los colores finales del juego: suelen aparecer sobre fondos blancos con objetos y texto en negro, mientras que en la versión definitiva se aplicará la paleta de colores pensada.

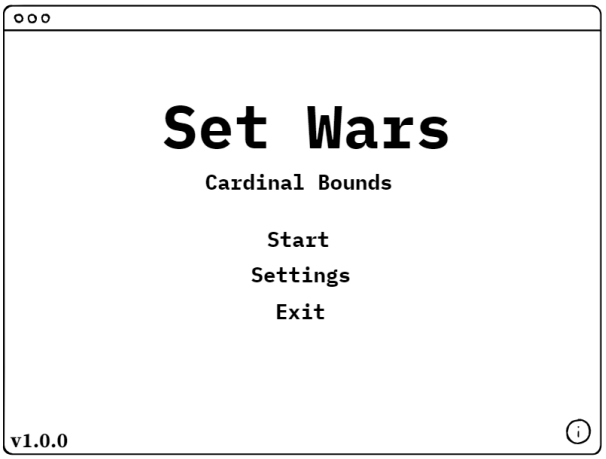


Figura 16: Boceto del menú principal.

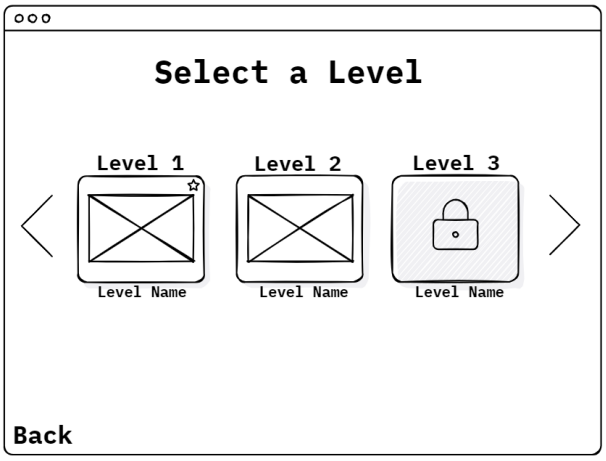


Figura 17: Boceto de la selección de niveles.

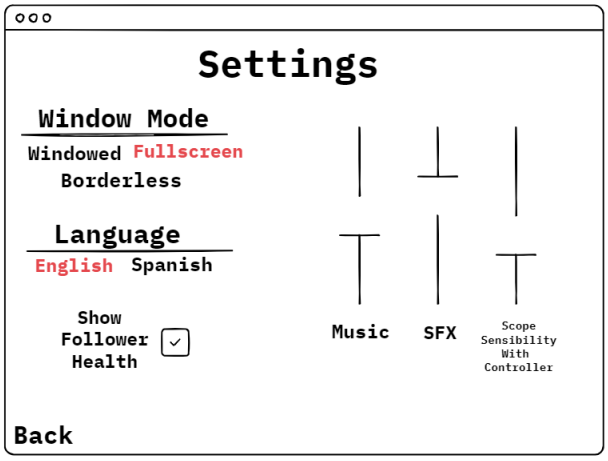


Figura 18: Boceto del menú de ajustes.

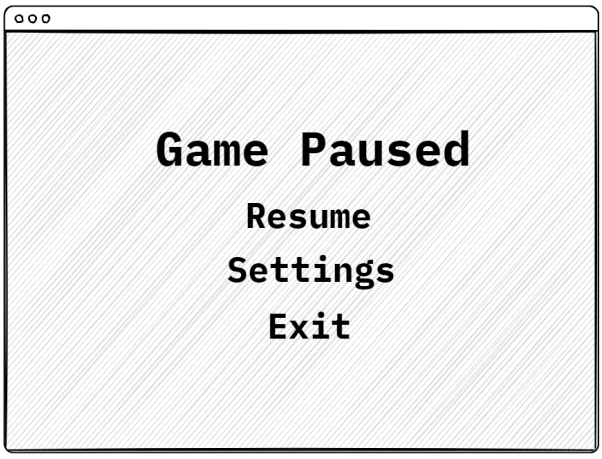


Figura 19: Boceto del menú de pausa.

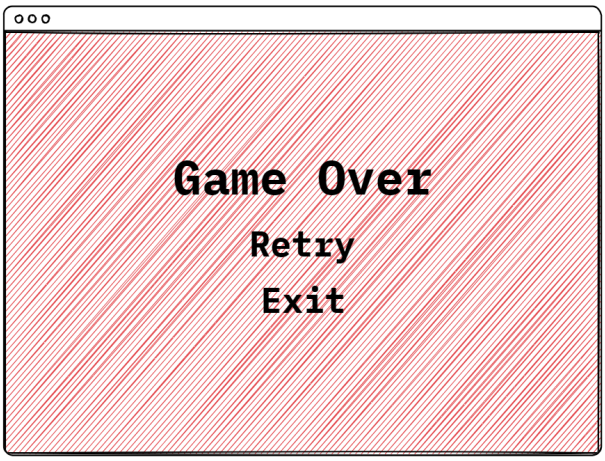


Figura 20: Boceto del menú de *Game Over*.

Figura 21: Boceto del menú de nivel completado.

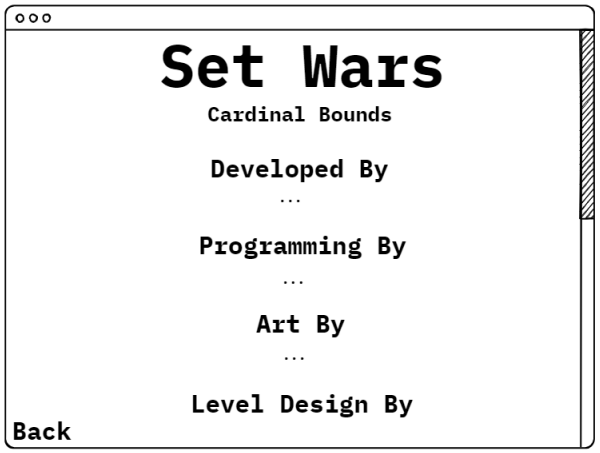


Figura 22: Boceto de los Créditos.

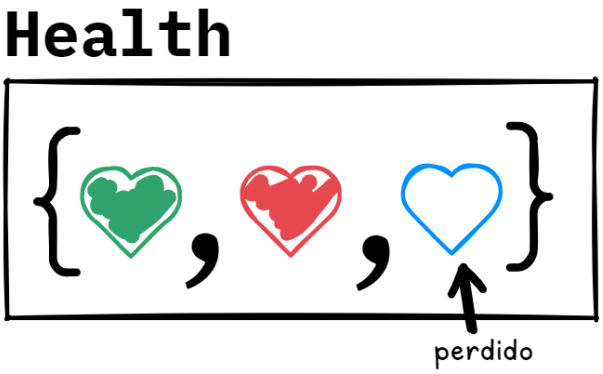


Figura 23: Boceto de la salud del jugador en el HUD.

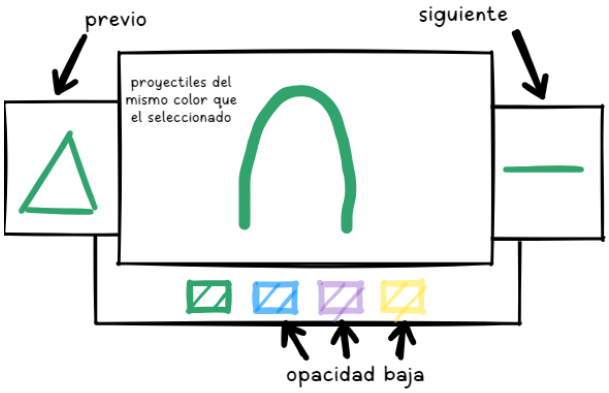


Figura 24: Boceto del selector de proyectiles en el HUD.

Figura 25: Boceto de la interfaz del diálogo.

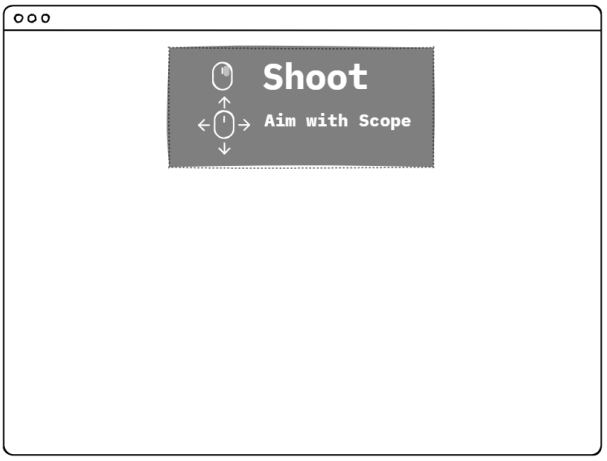


Figura 26: Boceto de la interfaz de leyenda de controles.

3.6. Bocetos de los personajes

Durante esta etapa también se realizaron bocetos de los personajes principales: el jugador y un enemigo. El objetivo de estos dibujos no era definir un estilo artístico final, sino establecer las bases visuales que distinguieran claramente sus roles dentro del juego. El jugador fue representado con un diseño simple y fácilmente identificable, que transmitiera su papel activo en la partida. El enemigo, por su parte, fue esbozado con rasgos diferenciadores que remarcaran su oposición al jugador, reforzando así la dinámica de confrontación central del juego. Al igual que con las interfaces, estos bocetos se centraron en la estructura y la legibilidad, sin aplicar todavía los colores definitivos que se verán en la versión final.

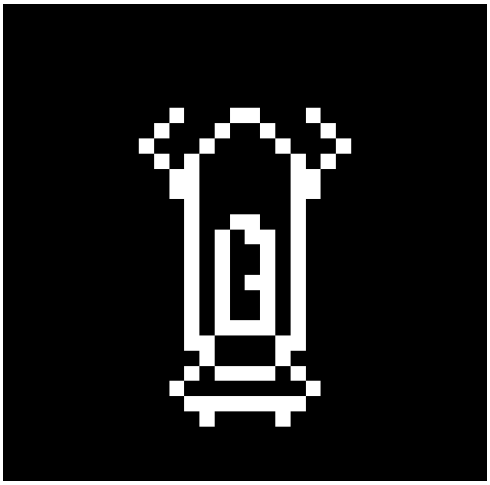


Figura 27: Boceto del enemigo básico.

Figura 28: Boceto del jugador.

3.7. Estilo audiovisual y experiencia de inmersión

El diseño del juego se basa en un estilo *pixel art 2D*, combinando la esencia del **arcade espacial** con una aproximación más **abstracta** y **geométrica**. Esta mezcla busca transmitir simultáneamente la sensación de inmensidad del espacio y la elegancia de un mundo matemático, creando una estética que es a la vez simple y sofisticada.

En cuanto a la paleta de colores, se ha utilizado un fondo **negro** que representa el espacio y la nada, contrastando con los elementos del escenario, que se presentan en **blanco**: la nave, los enemigos y los límites del mapa. El plano cartesiano y las partículas que se desplazan aleatoriamente en el fondo del espacio emplean **gris** para diferenciarse de los objetos principales y aportar sensación de profundidad.

El **naranja** se utiliza para indicar los propulsores del jugador y los enemigos, así como la interacción con la interfaz, destacando acciones como selecciones o pulsaciones.

Este esquema de colores no responde tan solo a un minimalismo matemático, sino a criterios de claridad y accesibilidad visual. Los elementos de salud y los proyectiles mantienen colores distintivos **cromáticos** para que puedan ser identificados fácilmente, mientras que los corazones, al representar componentes de un conjunto, reciben un destaque cromático propio dentro del entorno neutral. La ausencia de flashes en la pantalla refuerza esta **accesibilidad**, facilitando la experiencia a usuarios con sensibilidad visual.

La música del juego se ha seleccionado cuidadosamente para reforzar la atmósfera general y la sensación de inmersión. Cada nivel, así como el menú inicial, cuenta con su propia pista musical, diseñada para acompañar y enriquecer la experiencia del jugador. Los estilos elegidos combinan *electrónica* [20], *synthwave* [21] y *drone* [22], de manera que se enfatiza la abstracción, la sensación de espacio y la inmersión en un mundo de estética geométrica y espacial. La música del menú principal, por ejemplo, recurre a ambientes más calmados y etéreos, introduciendo al jugador en la experiencia antes de comenzar la acción. Mientras que la música de los niveles busca ser más animada, inmersiva y acorde al enfoque de cada nivel.

Los efectos de sonido se han seleccionado siguiendo criterios similares: algunos evocan la tradición arcade, como los disparos y explosiones de las naves, mientras que otros refuerzan la interacción con la interfaz, proporcionando retroalimentación auditiva inmediata al pulsar botones o seleccionar opciones. También se incluyen sonidos específicos asociados a eventos

importantes, como la pérdida o ganancia de corazones, tanto del jugador como de los enemigos, ofreciendo información adicional de manera intuitiva.

En conjunto, todos los sonidos mantienen una estética retro-futurista y electrónica, coherente con la dirección artística del juego, pero se han adaptado para que cada efecto cumpla su función de manera clara y reconocible dentro de la experiencia de juego.

3.8. Arquitectura general del proyecto

La arquitectura del proyecto se organiza en torno a una **escena principal** denominada *Game*, que actúa como nodo raíz y como *singleton central* del sistema. Este nodo cumple la función de coordinador general, ya que almacena en tiempo real el estado actual del juego (*Main menu, Playing, Paused, Game over, Loading, Settings o Level completed*), y es el encargado de gestionar la transición entre ellos. Además, centraliza el acceso a otros módulos que, aunque podrían haberse definido como *singletons independientes*, se han implementado como nodos hijos para simplificar su manejo.

El nodo *Game* se encarga también de invocar las funciones necesarias para instanciar o eliminar tanto las interfaces de usuario como los niveles, actuando como punto de entrada al resto de subsistemas.

Los módulos que incluye *Game* son:

- **GameInputs**: gestiona la entrada del jugador, identificando si se utiliza mando o teclado/ratón, controlando la visibilidad y posición del cursor, y bloqueando la interacción durante las fases de carga.
- **GameLanguage**: administra el idioma activo del juego y notifica cambios cuando se produce una modificación, siendo el módulo de referencia para la traducción de *translation_id*.
- **GameDisplay**: controla el estado de la ventana (modo ventana, pantalla completa o pantalla completa sin bordes) y mantiene la resolución actual.
- **GameData**: actúa como gestor de persistencia, encargándose de la creación, lectura y actualización de los archivos de guardado relacionados con las opciones y el progreso del jugador.

- **GameSettings**: interfaz central para la gestión de opciones, utilizada por la interfaz de ajustes. Se comunica con *GameData* para almacenar configuraciones y aplica los cambios necesarios en el entorno de juego.

- **GameAudio**: administra los buses[6] de audio y sus volúmenes, además de controlar la música global y los sonidos de interfaz que no dependen de la posición espacial.

- **GameLevels**: contiene la lista de niveles disponibles, empleada posteriormente por el menú de selección de niveles.

- **GameUI**: gestiona la instanciación, eliminación y acceso a las diferentes interfaces gráficas del juego.

- **Background**: nodo encargado del fondo visual que permanece visible tanto en los menús como durante la partida.

- **Gameplay**: módulo responsable de la instanciación y destrucción de los niveles. Contiene a su vez la clase *Level*, que organiza y administra elementos clave de cada partida, como la cámara principal del nivel, el jugador y su *spawner*, la mirilla y otros elementos adicionales propios del nivel. Asimismo, define la configuración inicial del jugador (salud y armamento), gestiona su aparición en el escenario y controla los eventos de *Game Over* o *Nivel completado*.

Además de los módulos principales, el proyecto cuenta con un conjunto de elementos denominados **Utils**. Se trata de funciones, clases y recursos *static* que no dependen de una instancia específica de los módulos, sino que proporcionan utilidades generales accesibles desde cualquier parte del código. Estas utilidades son métodos de uso recurrente que facilitan la implementación y evitan la duplicación de código.

[6]Un bus de audio es un canal virtual que agrupa varias fuentes de sonido para su procesamiento y mezcla, son útiles para ajustar el volumen de todo un conjunto de sonidos sin tener que ir uno por uno.

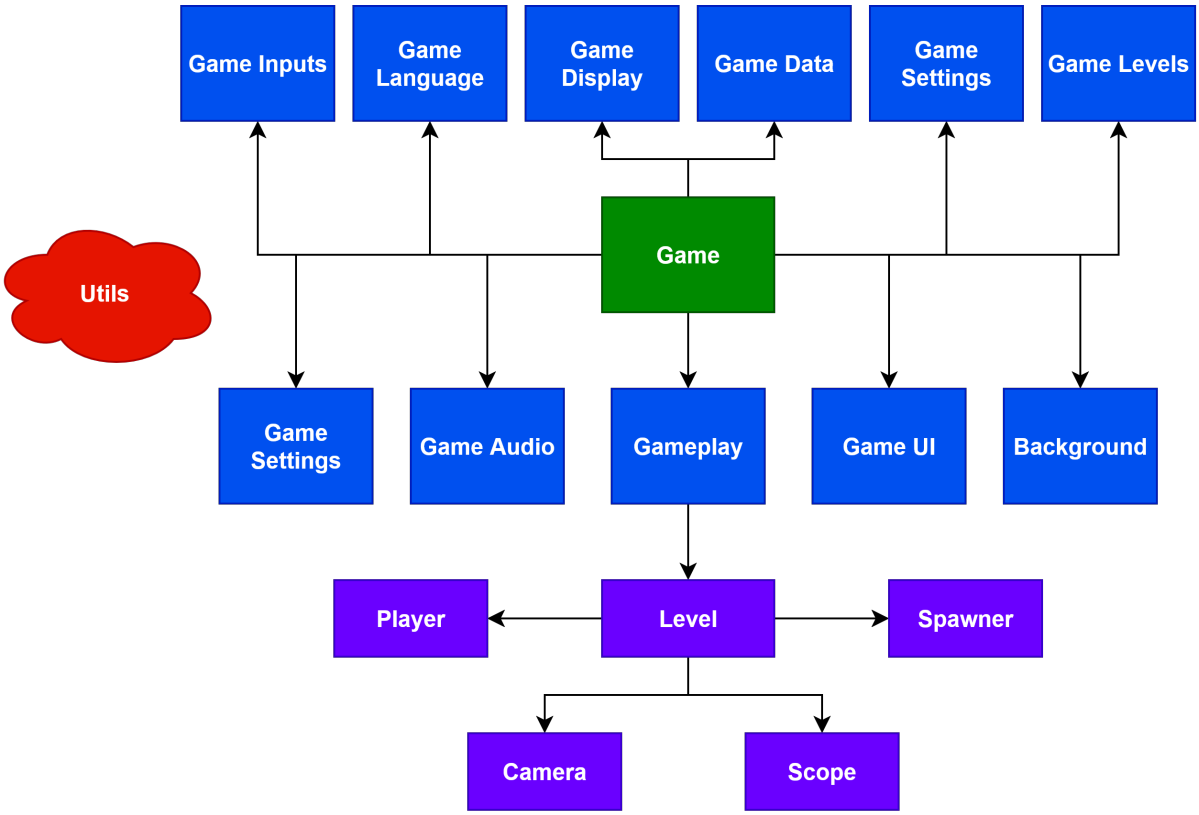


Figura 29: Diagrama de la arquitectura general del proyecto.

4

Implementación y Desarrollo

El presente apartado recoge el proceso de creación, detallando cómo las ideas, requisitos y diseños previos se han transformado en un videojuego funcional. Aquí se explica cómo se ha estructurado el proyecto en Godot, las decisiones técnicas tomadas y los retos encontrados durante el desarrollo.

La sección está organizada siguiendo la propia arquitectura del juego: primero se describe la estructura de carpetas y la organización general, para después profundizar en los distintos módulos y sistemas que componen el núcleo del proyecto. Se abordan tanto los aspectos más visibles para el jugador (interfaz, niveles, entidades) como las utilidades y sistemas internos (gestores, *triggers*, sistema de guardado, etc.), mostrando cómo cada pieza contribuye al conjunto.

4.1. Estructura de las carpetas

La organización interna del proyecto es fundamental para mantener el orden y facilitar tanto el desarrollo como el mantenimiento a largo plazo. Se ha optado por una estructura sencilla pero efectiva, basada en tres carpetas principales:

- **Scenes**: Aquí se encuentran todas las escenas del juego, organizadas jerárquicamente en subcarpetas según su función o nivel. Godot utiliza un sistema de escenas muy flexible, permitiendo instanciar unas dentro de otras para construir la estructura del juego. La escena principal, *Game.tscn*, actúa como raíz y es la que se carga al iniciar el juego.

- **Scripts**: En esta carpeta se almacenan todos los *scripts* escritos en C#, también organizados en subcarpetas siguiendo la lógica de las escenas. Cada nodo que requiere fun-

cionalidad específica tiene asociado su propio script, lo que facilita la reutilización y el mantenimiento del código.

- **Assets**: Esta carpeta contiene todos los recursos necesarios para el juego, organizados en varias subcarpetas:
  - Pistas de audio (música y efectos de sonido).
  - Fuentes de texto.
  - Archivos *.gdshader*, que son shaders programados específicamente para lograr efectos visuales concretos en Godot.
  - *Sprites* de todos los elementos visuales, diseñados personalmente usando Krita.
  - El archivo *.csv* de traducciones, que permite gestionar los diferentes idiomas del juego.
  - Gradientes, materiales y polígonos reutilizables, almacenados para evitar duplicidades y facilitar su uso en distintos nodos. Son archivos nativos de Godot.
- **Build**: Esta carpeta contiene las distintas versiones exportadas (*builds*) del juego listas para ser ejecutadas en diferentes plataformas. Cada subcarpeta corresponde a una plataforma de destino, como Windows o Linux, facilitando la distribución y las pruebas multiplataforma.

Esta estructura no solo ayuda a mantener el proyecto ordenado, sino que también agiliza el trabajo diario y la localización de cualquier recurso o script durante el desarrollo.

4.2. Configuración del proyecto

La configuración global del proyecto en Godot es importante para definir el comportamiento, la apariencia y la compatibilidad del juego desde el primer momento. A continuación se destacan los principales ajustes realizados en las *Project Settings*:

- **Versión del juego**: Durante el desarrollo, la versión fue cambiando según el avance y las pruebas. La versión final publicada es la 1.0.0, reflejando la primera entrega estable del proyecto.

- **Icono del juego**: Se ha personalizado el icono del ejecutable y del proyecto, utilizando una imagen original diseñada para el juego. Siendo el corazón rojo del jugador entre los paréntesis de un conjunto.

Figura 30: Icono del juego (Windows).

- **Boot splash**: Al iniciar el juego, se muestra una pantalla de carga (*boot splash*) que utiliza el icono del juego como imagen central, aportando identidad visual desde el primer momento.

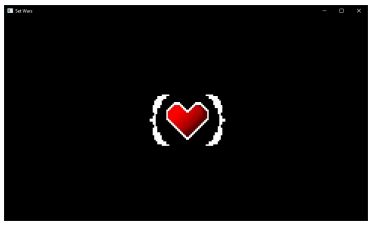


Figura 31: Pantalla de carga del juego al iniciarlo (*boot splash*).

- **Color de fondo**: El fondo por defecto del juego es negro, configurado mediante la opción *bg color* en las opciones de *Display/Window*.
- **Resolución**: El tamaño del *viewport* está fijado en 960x540 píxeles, una resolución panorámica que equilibra la visibilidad y el rendimiento, y que se adapta bien tanto a pantallas grandes como pequeñas. Es una buena escala para mantener la proporción y evitar distorsiones en los gráficos ya que el juego es en pixel art.
- **Grupos**: Godot permite asignar etiquetas (*tags*) llamadas grupos a los nodos. Un mismo nodo puede pertenecer a varios grupos, lo que facilita su identificación y manipulación

desde el código.

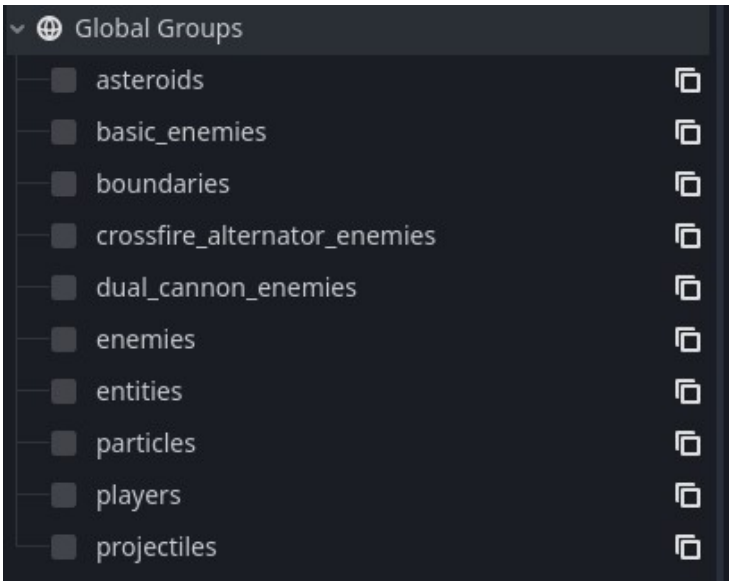


Figura 32: Grupos (*tags*) creados para el juego.

- **Render Layers names**: Se han definido nombres personalizados para las capas de renderizado, lo que permite organizar y controlar el orden visual de los nodos en pantalla. Esto es especialmente útil para asegurarse de que elementos como el HUD, los personajes o los fondos se dibujen en el orden correcto.

Figura 33: Capas de renderizado nombradas.

- **Physics Layers names**: De igual manera, se han configurado nombres para las capas de físicas, facilitando la gestión de colisiones y la interacción entre áreas y cuerpos físicos dentro del juego.

Figura 34: Capas de físicas nombradas.

- **Input Map**: El *input map* recoge todas las acciones del juego y sus asignaciones tanto para teclado/ratón como para mando. Esta configuración permite que el jugador utilice el control que prefiera y facilita la accesibilidad. En el GDD (Apéndice B, apartado 3.1) se detalla una tabla completa con la asignación de controles.

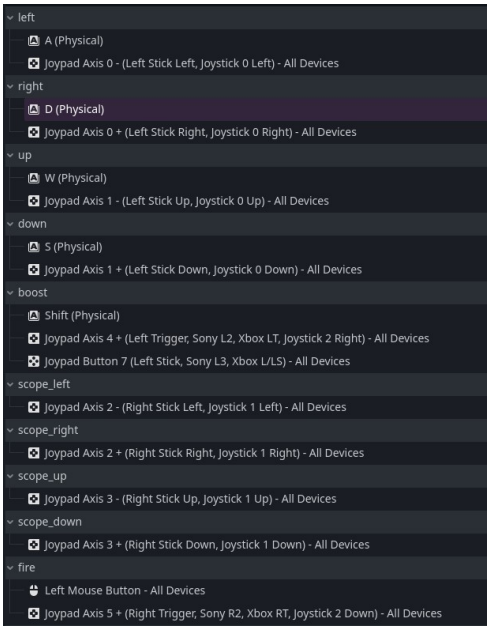


Figura 35: Algunas entradas y sus asignaciones del (*input map*) del juego.

Estos ajustes iniciales son fundamentales para garantizar tener el videojuego con todos los aspectos necesarios para su correcto funcionamiento.

4.3. *Game:* La clase raíz

Tal y como se describe en la arquitectura general del proyecto (sección 3.8), la escena principal *Game.tscn* implementa el nodo raíz del juego, instancia de la clase *Game.cs*. Esta clase centraliza la gestión de los módulos principales y coordina el flujo global del juego.

A diferencia de la descripción conceptual previa, aquí se detalla cómo se estructura realmente en Godot:

- El nodo **Game** es un *singleton* accesible desde cualquier script, lo que permite gestionar estados globales y recursos compartidos.
- Cada módulo (inputs, idioma, display, datos, ajustes, audio, niveles, UI, gameplay, background) se implementa como nodo hijo directo de *Game*, con su propio script asociado.
- El estado global del juego se gestiona mediante un *enum*[7] llamado *GameState*, que permite controlar las transiciones entre menús, partida, pausa, etc.
- La comunicación entre módulos se realiza mediante *signals* y métodos públicos, evitando dependencias circulares y facilitando la mantenibilidad.
- La instanciación y destrucción de interfaces y niveles se realiza desde métodos en *Game*, que se manejan en los módulos correspondientes.

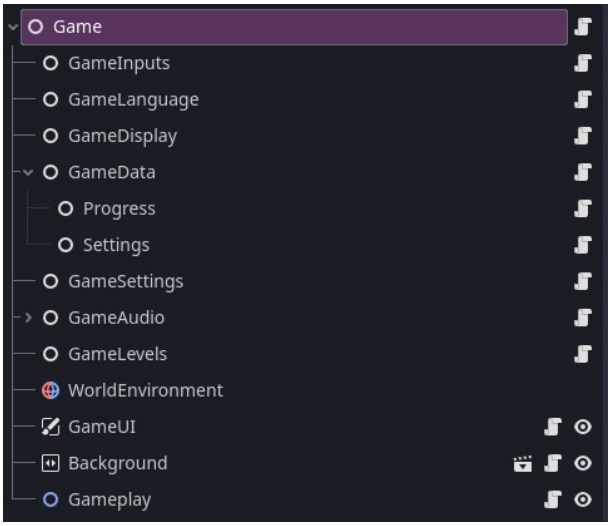


Figura 36: Árbol de nodos de la escena principal del juego en Godot.

La mayoría de nodos en la escena principal del juego se pueden observar que son nodos vacíos (*Node*), representados por un círculo gris, estos no tienen representación visual en la

[7]Un *enum* (o enumerable) es un tipo de dato que permite definir un conjunto de valores constantes. Sirven para representar un grupo de opciones relacionadas de manera más legible y mantenible.

escena y no pueden tener coordenadas en el espacio. Estos nodos actúan como contenedores lógicos para organizar los distintos módulos y sistemas del juego.

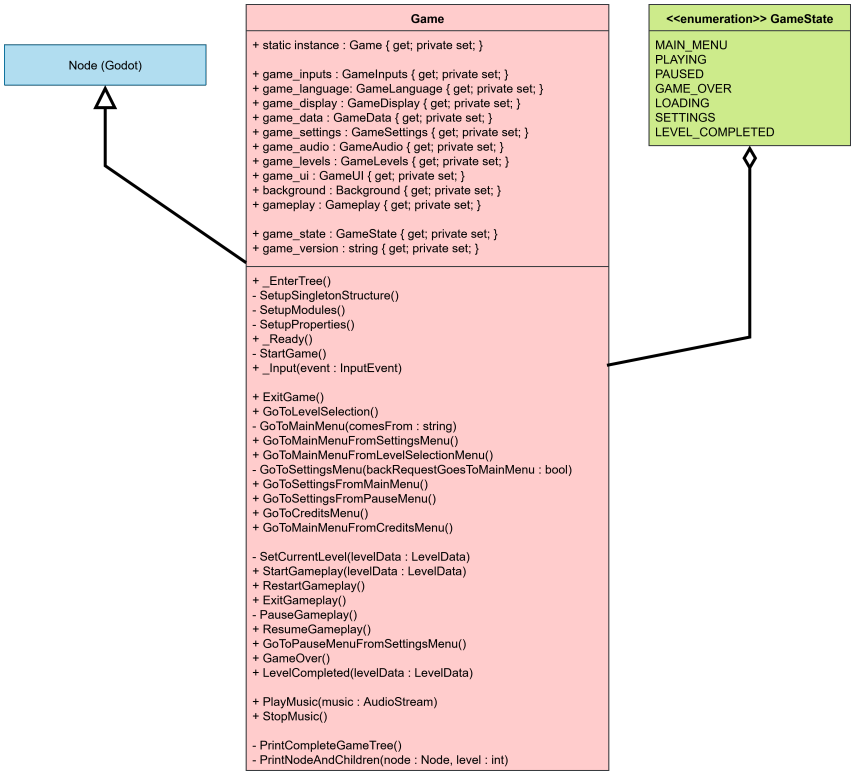


Figura 37: Representación UML de la clase *Game*.

A continuación se describen los principales módulos integrados en la clase *Game*:

4.3.1. *Game Inputs*

Este módulo gestiona lo relacionado con las entradas del jugador, tanto de teclado/ratón como de mando. Mediante el uso de la función nativa *_Input()* es capaz de comprobar qué tipo de entrada se está utilizando en cada momento y actualizar el actual tipo de entrada del juego. Se ha utilizado un *enum* llamado *InputType* para representar los posibles tipos de entrada: *KEYBOARD* para teclado y ratón, *JOYPAD* para mando.

Mediante el uso de una *signal* se notifica a cualquier nodo subscrito cuando el tipo de entrada cambia, permitiendo que la interfaz y otros elementos se adapten dinámicamente.

Por otro lado, proporciona métodos para establecer el estado actual del ratón (visible o invisible) y para bloquear o desbloquear el cursor en el centro de la pantalla, funcionalidades

útiles durante la partida. También guarda siempre la última posición conocida del ratón, para poder utilizarla cuando sea necesario.

Además marca como manejada cualquier entrada para que no se procese en ningún sitio cuando el juego está en un estado donde no se deben aceptar (en este caso, *Loading*).

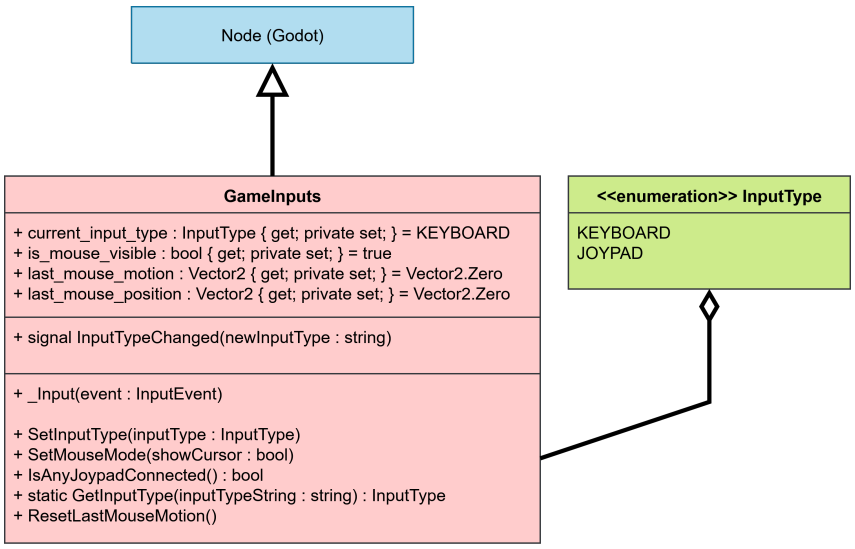


Figura 38: Representación UML de la clase *GameInputs*.

4.3.2. *Game Language*

Módulo encargado de gestionar el idioma actual del juego y las traducciones. Utiliza el sistema de traducciones nativo de Godot, que permite cargar archivos *.csv* con las cadenas traducidas a diferentes idiomas. En este caso, se han definido los idiomas español e inglés, pero el sistema es fácilmente ampliable para añadir más idiomas en el futuro.

Además, existe una clase *estática* nativa de Godot llamada *TranslationServer* que facilita toda esta gestión. En esta clase se actualiza cuál es el idioma actual en tiempo de ejecución y se proporcionan funciones que cargan las traducciones según el idioma activo y la *translation_id* que se quiera traducir.

La creación de este módulo es una capa más encima de la clase nativa, que añade funcionalidades específicas para el juego. Como tener el idioma actual guardado como un *enum* llamado *Language*, una *signal* que notifica a cualquier nodo subscrito cuando el idioma cambia (al igual que en *GameInputs*), y un método para traducir una cadena dada su *translation_id* pero con un sistema de manejo de excepciones en caso de que la *translation_id* no exista o no

esté traducida en el idioma actual.

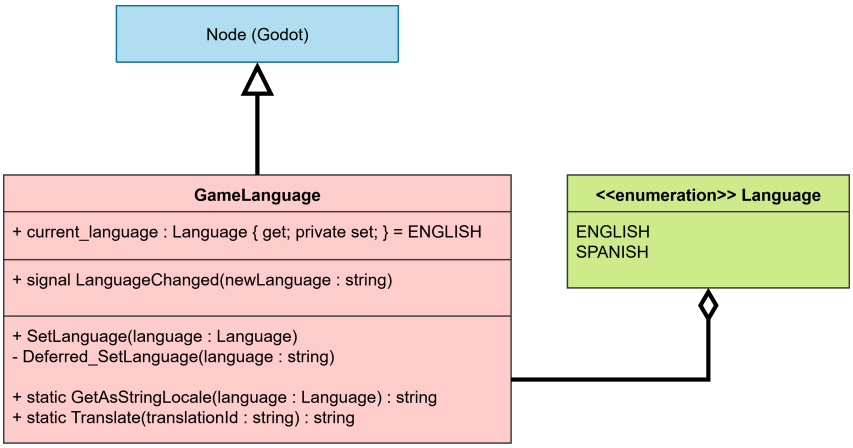


Figura 39: Representación UML de la clase *GameLanguage*.

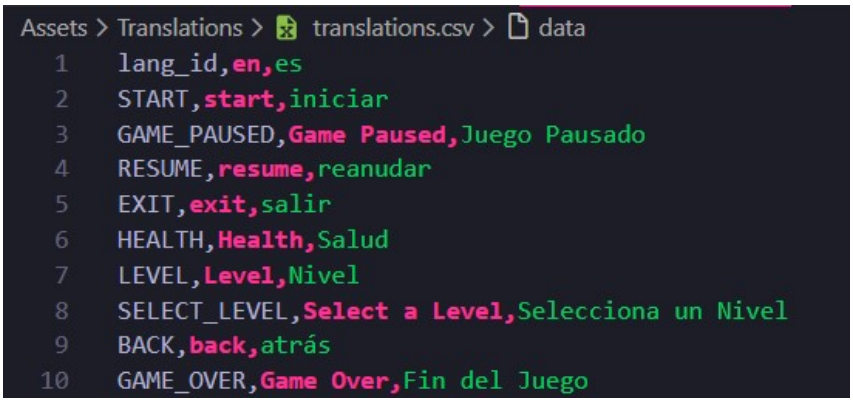


Figura 40: Inicio del archivo *.csv* de traducciones del juego.

4.3.3. *Game Display*

Maneja todo lo relacionado con la pantalla y la resolución del juego. Proporciona métodos alternar entre el modo de la ventana (pantalla completa, pantalla completa sin bordes o ventana). De forma análoga a los módulos anteriores, tiene una *signal* que notifica a cualquier nodo subscrito cuando el modo de pantalla cambia y un *enum* llamado *WindowMode* que representa los posibles modos de pantalla.

También permite conocer la resolución actual del juego y tiene métodos para cambiar la resolución a una dada, pero esto último no se utiliza en el juego ya que la resolución está fija a 960x540 píxeles debido a limitaciones de tiempo. También existe una *signal* que notifica a cualquier nodo subscrito cuando la resolución cambia, aunque actualmente no ocurra nunca.

Estas funcionalidades son útiles para el menú de ajustes, donde el jugador puede cambiar el modo de pantalla.

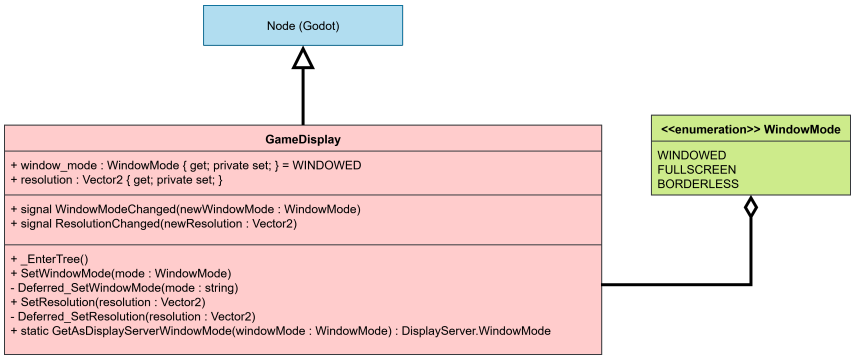


Figura 41: Representación UML de la clase *GameDisplay*.

4.3.4. *Game Data*

Este módulo sirve como punto de acceso a los dos gestores de datos guardados del juego: el gestor de datos de configuración y el gestor de datos de progreso, explicados en la sección 4.21.

Para ello, *Game Data* define varias constantes accesibles y utiliza la clase nativa *DirAccess* de Godot, que permite gestionar directorios y archivos de forma multiplataforma. Antes de guardar cualquier dato, el módulo comprueba si existe la carpeta de usuario correspondiente (ubicada en *user://data/*); si no existe, la crea automáticamente. Esta ruta es gestionada por Godot y se adapta al sistema operativo, garantizando que los datos estén almacenados en un lugar seguro, accesible para el juego y oculto en cierta medida para el usuario.

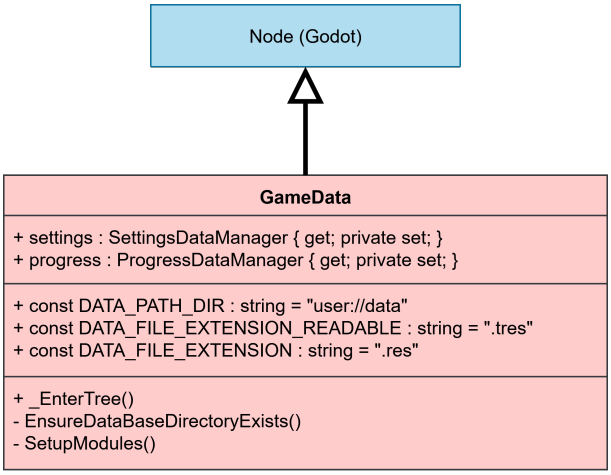


Figura 42: Representación UML de la clase *GameData*.

4.3.5. *Game Settings*

Se encarga de aplicar los cambios requeridos en los ajustes del juego, tanto los cargados al iniciar el juego (que los aplica todos) como los modificados por el jugador en el menú de ajustes. Para ello, utiliza el gestor de datos de configuración (explicado en la sección 4.21), se conecta a su *signal* de cambios y actualiza los valores en consecuencia según lo que se haya modificado.

Además, proporciona una *signal* que notifica que se han hecho cambios en los ajustes, permitiendo que otros módulos o nodos respondan a estas modificaciones si es necesario.

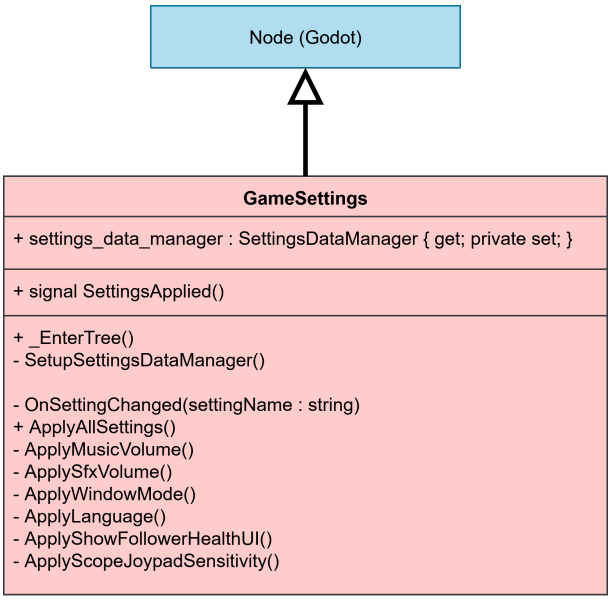


Figura 43: Representación UML de la clase *GameSettings*.

4.3.6. *Game Audio*

Gestiona lo relacionado con los buses de audio del juego, sus volúmenes (tanto lectura como escritura). Hace uso de la clase nativa *AudioServer* de Godot, que permite manipular los buses de audio en tiempo de ejecución.

En Godot, los buses de audio se configuran en el editor, donde se pueden crear, renombrar y organizar en una jerarquía. Existe el bus maestro (*Master*), que es el bus raíz, y debajo de él se pueden crear buses hijos para diferentes categorías de sonido, si un bus padre cambia su sonido también la afectará en cadena a sus hijos. Cada bus puede tener sus propios efectos de audio aplicados, como ecualizadores, compresores o reverberaciones.

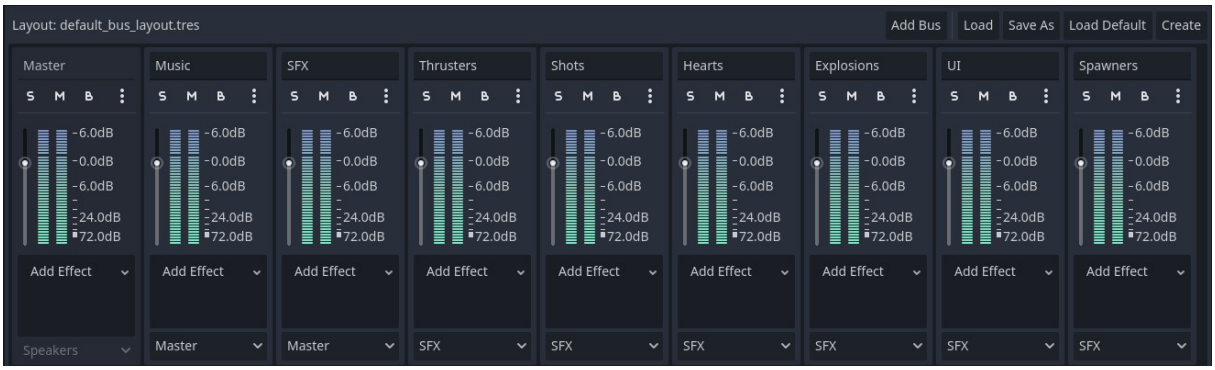


Figura 44: Configuración de los buses de audio en Godot.

El módulo primero obtiene y guarda todos los buses en un diccionario, para después proporcionar métodos que permiten establecer y obtener el volumen de cada bus, pudiéndose manejar en decibelios (dB) o en una escala lineal de 0 a 1. También ofrece métodos para silenciar o activar un bus, y para comprobar si un bus está silenciado.

Para indicar que bus se quiere modificar o consultar, se debe indicar con un *string*, los cuales están definidos como constantes públicas en una clase estática aparte llamada *AudioBuses*.

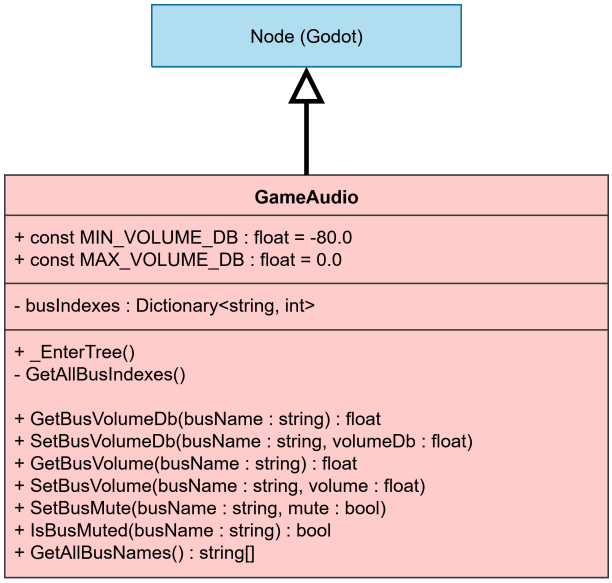


Figura 45: Representación UML de la clase *GameAudio*.

```
public static class AudioBuses
{
    0 references
    public const string MASTER = "Master";
    4 references
    public const string MUSIC = "Music";
    4 references
    public const string SFX = "SFX";
    1 reference
    public const string THRUSTERS = "Thrusters";
    1 reference
    public const string SHOTS = "Shots";
    2 references
    public const string HEARTS = "Hearts";
    1 reference
    public const string EXPLOSIONS = "Explosions";
    3 references
    public const string UI = "UI";
    2 references
    public const string SPAWNERS = "Spawners";
}
```

Figura 46: Código de *AudioBuses*.

4.3.7. *Game Levels*

Este módulo contiene una lista con los datos de todos los niveles del juego, representados por la clase *LevelData*. Esta lista luego es utilizada por el Menú de selección de nivel para mostrar los niveles disponibles y sus detalles.

Los datos que almacena cada instancia de *LevelData* son: la *id*, el nombre (*translation_id*), la *PackedScene* del nivel, el icono que se muestra en el menú de selección de nivel, los niveles que desbloquea al ser completado y si el nivel necesita o no ser desbloqueado para poder jugarlo.

Además, *LevelData* es una clase que no hereda de un nodo normal de Godot, sino que hereda de *Resource*, el cuál es la clase base para todos los recursos que pueden ser guardados en archivos y reutilizados en diferentes partes del proyecto, usando el formato nativo de Godot *.tres*. Esto permite crear instancias de *LevelData* para cada nivel y asignarlas desde el editor en vez de hacerlo mediante código.

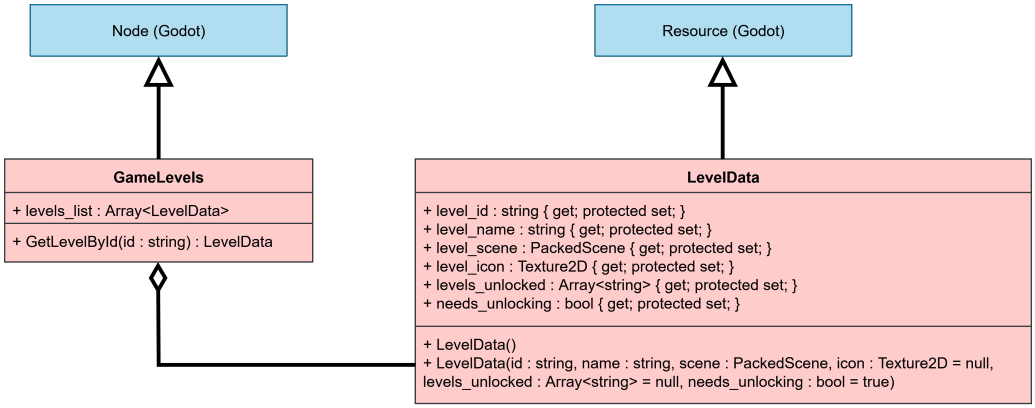


Figura 47: Representación UML de la clase *GameLevels* y *LevelData*.

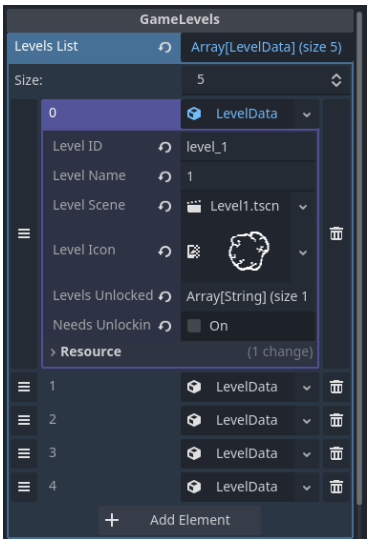


Figura 48: Vista del editor de Godot para asignar los datos de cada nivel en *GameLevels*.

4.3.8. *Game UI*

Instancia y elimina las diferentes interfaces de usuario del juego, tanto el HUD como los menús y la animación de transición, haciéndolas hijos suyos. Además, mantiene una referencia para cada interfaz actualmente activa, permitiendo acceder a ellas desde cualquier parte del código si es necesario (si una interfaz no está activa, la referencia es *null*).

La clase *Game* llama a este módulo para instanciar o eliminar las interfaces en los momentos adecuados, como al iniciar una partida, pausar o reanudar el juego, o volver al menú principal.

Esta clase no hereda de *Node* como los demás módulos, sino que hereda de *CanvasLayer*, que es un tipo especial de nodo en Godot diseñado para manejar elementos de interfaz de usuario y otros elementos gráficos que deben permanecer fijos en la pantalla, independientemente de la posición o el movimiento de la cámara en el juego.

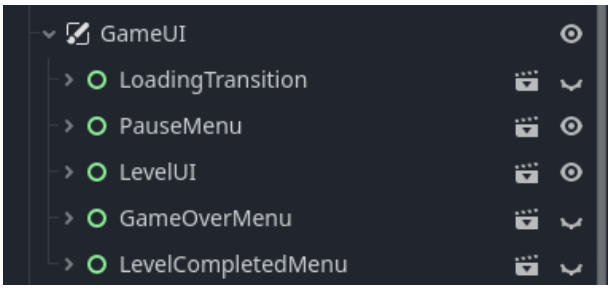


Figura 49: Ejemplo del árbol de nodos de *GameUI* durante la ejecución del juego.

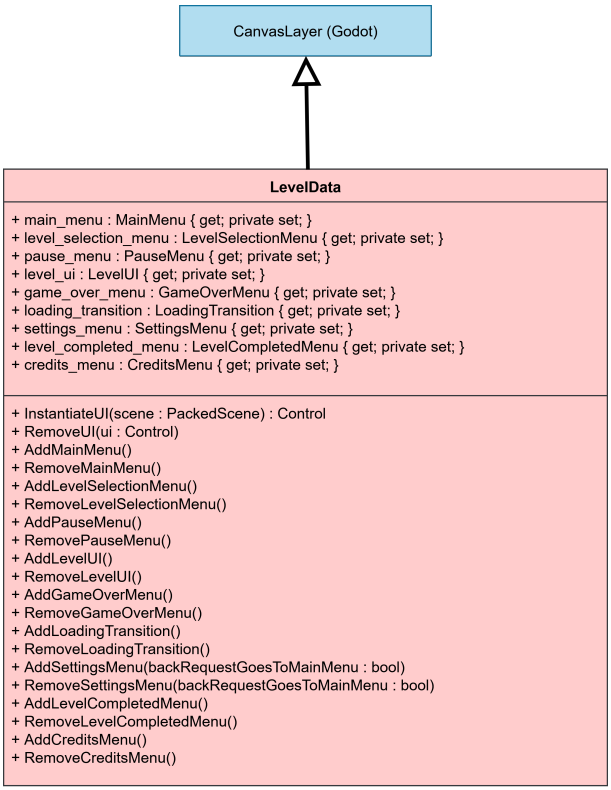


Figura 50: Representación UML de la clase *GameUI*.

4.3.9. *Gameplay*

Se encarga de instanciar y eliminar el nivel actual que se esté jugando, haciéndolo hijo suyo. Mantiene una referencia al nivel actual, permitiendo acceder a él desde cualquier parte del código si es necesario (si no hay ningún nivel cargado, la referencia es *null*). Teniendo correspondientes *signals* que notifican cuando se instancia o elimina un nivel.

También se encarga de emitir al nivel actual si hay cambios en dos de los ajustes del juego: la sensibilidad de la mirilla (*Scope*) usando el mando y si la interfaz de salud seguidora es visible

o no (estos dos ajustes se explican en la sección 4.21.2). Con también correspondintes *signals* que notifican cuando estos ajustes se notifican.

También se encarga de emitir a *Game* cuando el nivel actual ha sido completado, para que este pueda gestionarlo.

Esta clase hereda de *Node2D*, ya que a partir de aquí el nivel instanciado es un nodo 2D y necesita tener coordenadas en el espacio.

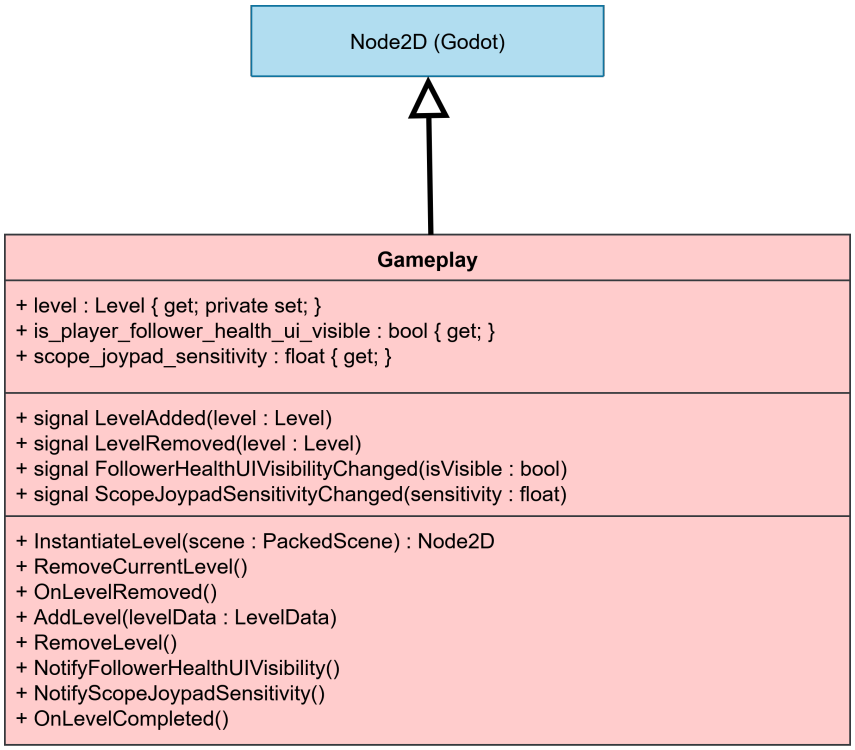


Figura 51: Representación UML de la clase *Gameplay*.

4.3.10. *Background*

El fondo del juego está compuesto por un nodo de tipo *ParallaxBackground*[8] que se mueve automáticamente en función de la posición de la cámara.

Este nodo tiene como hijo un *ParallaxLayer*, que a su vez contiene un *TextureRect* encargado de mostrar la imagen del fondo. Dicha imagen se repite de manera infinita tanto en horizontal como en vertical, cubriendo toda la pantalla y transmitiendo la sensación de un plano cartesiano infinito.

[8]Un *parallax* es un efecto visual que crea la ilusión de profundidad al mover diferentes capas de fondo a diferentes velocidades. Es una técnica muy común a la hora de desarrollar juegos con escenarios bidimensionales.

Además, se incluye una subescena con diferentes partículas animadas que se generan aleatoriamente, variando en tamaño, velocidad y diseño para simular estrellas y formas geométricas en movimiento. Esto aporta mayor dinamismo y vida al fondo.

La clase *Background* hereda de *ParallaxBackground* y su principal función es restablecer la posición inicial (0, 0) del fondo al salir de un nivel, de modo que al iniciar uno nuevo el fondo no esté desplazado, ni tampoco cuando lo esté en el menú principal (ya que el fondo se comparte en todo el juego). Es la clase *Game* la que llama a este método cuando se elimina un nivel.

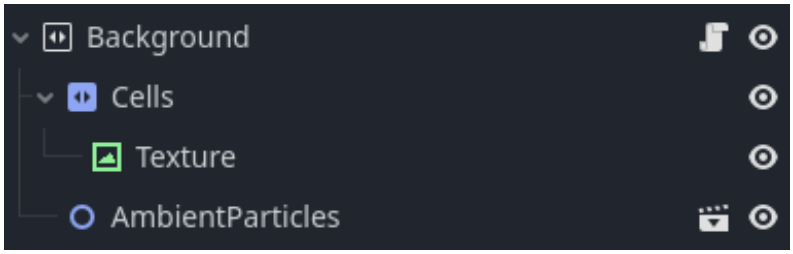


Figura 52: Árbol de nodos de la escena de *Background* en Godot.

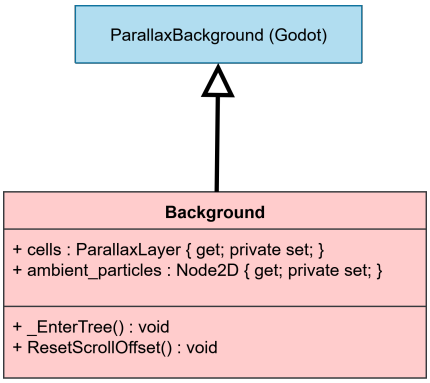


Figura 53: Representación UML de la clase *Background*.

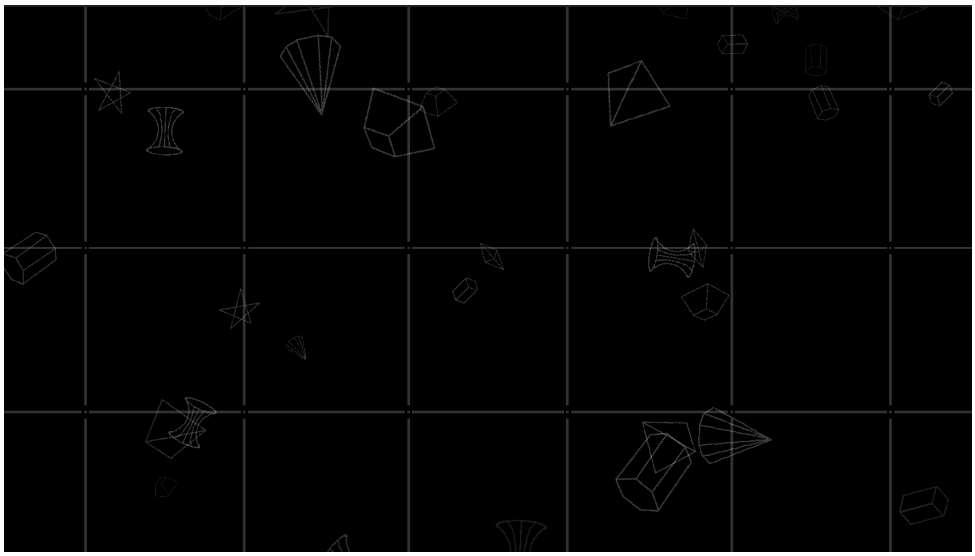


Figura 54: Escena de *Background*.

4.4. Utilidades y extras

Una de las subcarpetas dentro de *Scripts* es *Utils*, que contiene *scripts* con utilidades y funcionalidades adicionales que no encajan directamente en los módulos principales, pero que resultan útiles en diversas partes del juego. La mayoría de estas utilidades son clases estáticas, listas de constantes o métodos estáticos, lo que permite acceder a sus métodos y propiedades sin necesidad de instanciarlas, facilitando así su uso en cualquier parte del código. A continuación, se describen las utilidades que se consideran más relevantes:

4.4.1. Utilidades para el audio

La clase estática *AudioBuses*, explicada en la sección 4.3.6, se incluye en esta subcarpeta.

También se encuentra *AudioType*, un *enum* que define los diferentes tipos de audio en el juego: música y varios tipos de efectos de sonido (UI, explosiones, disparos, entre otros). Este *enum* es utilizado por los reproductores de audio para categorizar y gestionar los sonidos de manera adecuada, como se detalla en la sección 4.6.

Además, se incluye *AudioUtils*, que proporciona un método de extensión[9] para *AudioType*, permitiendo obtener el bus de audio correspondiente a cada tipo mediante un *mapping*[10].

[9]Un método de extensión en C# permite agregar métodos a tipos existentes sin modificar su definición original; es posible extender clases, estructuras y tipos de datos primitivos.

[10]Un *mapping* es un sistema de asignación que relaciona un conjunto de valores con otro. El propósito en

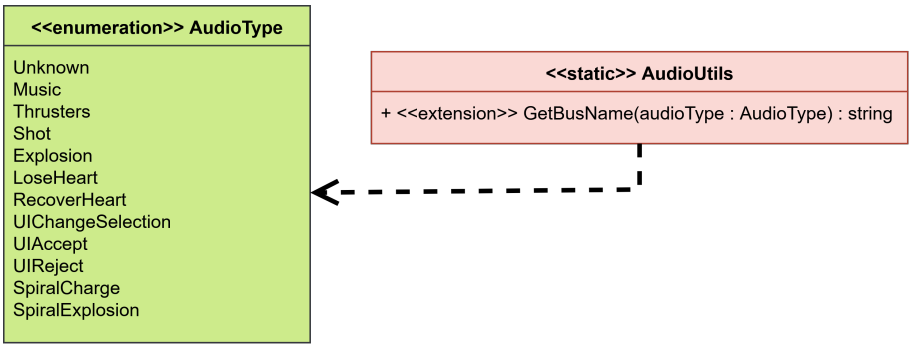


Figura 55: Representación UML de la clase *AudioType* con su método de extensión en *AudioUtils*.

4.4.2. Utilidades para los colores

Los corazones de los conjuntos de salud (explicados en la sección 4.8.1) y los colores de los proyectiles (explicados en la sección 4.10) están representados por un *enum* llamado *GameColor*, que define los colores disponibles en el juego para no tener que manejar valores *RGB* o *Hex* directamente.

Además, la clase estática *GameColorUtils* proporciona un método de extensión para *GameColor*, permitiendo obtener el color real (de tipo *Color*, nativo de Godot) correspondiente a cada valor del *enum* mediante un *mapping*. También se incluye un método análogo para *Color* que devuelve el *GameColor* correspondiente.

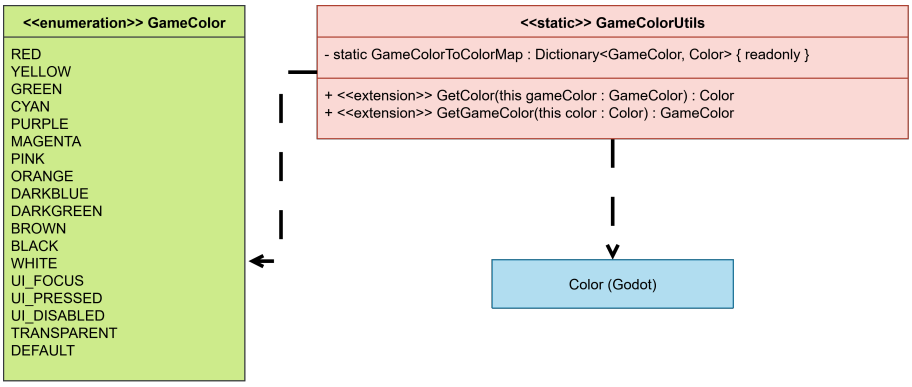


Figura 56: Representación UML de la clase *GameColor* y los métodos de extensión de *GameColorUtils*.

ciertos casos es el de facilitar una conversión de datos equivalentes o que están relacionados.

4.4.3. Utilidades para los grupos

Otro *enum* llamado *GameGroup* define los diferentes grupos (*tags*) utilizados en el juego, explicados en la sección 4.2. De forma análoga, la clase estática *GameGroupUtils* proporciona un método de extensión para *GameGroup*, permitiendo obtener el nombre del grupo correspondiente (en *string*) a cada valor del *enum* mediante un *mapping*.

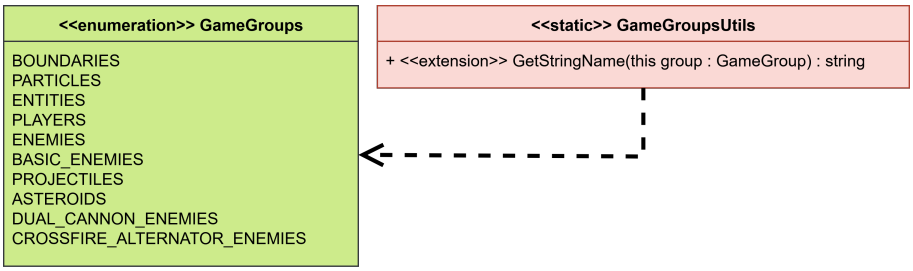


Figura 57: Representación UML de la clase *GameGroup* y su método de extensión en *GameGroupUtils*.

4.4.4. Utilidades para las capas

Similar a los grupos, el *enum GamePhysicsLayer* define las diferentes capas de físicas utilizadas en el juego, explicadas en la sección 4.2.

La clase estática *GamePhysicsLayerUtils* proporciona un método de extensión para *GamePhysicsLayer*, permitiendo obtener el *bitmask*[11] de la capa correspondiente (en *uint*) a cada valor del *enum* mediante un *mapping*.

También se incluye un método de extensión para un *array* de *GamePhysicsLayer*, que devuelve el *bitmask* combinado de todas las capas en el array.

[11] Un *bitmask* es una secuencia de bits que se utiliza para representar un conjunto de valores booleanos. En este caso, cada capa de físicas se representa como un bit en un número entero, lo que permite realizar operaciones de comparación y combinación de capas de manera eficiente.

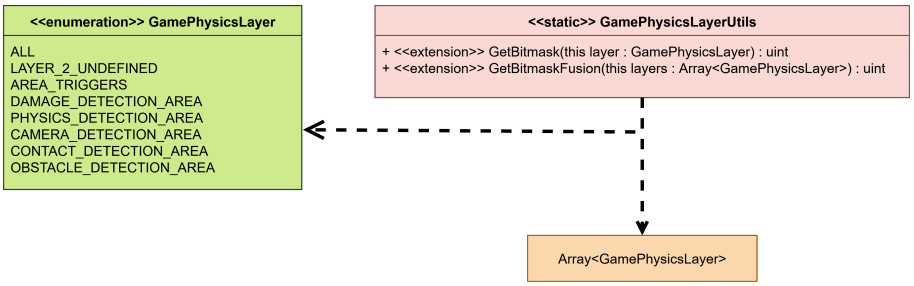


Figura 58: Representación UML de la clase *GamePhysicsLayer* y los métodos de extensión en *GamePhysicsLayerUtils*.

4.4.5. Utilidades para los nodos y procesamiento

La clase estática *ProcessUtils* resuelve una limitación de Godot respecto a Unity: la función *SetActive()*, que no existe en Godot. En Unity, esta función permite activar o desactivar un nodo (*gameObject*) y todos sus hijos de forma recursiva, gestionando su procesamiento y visibilidad. En Godot, esto debe hacerse manualmente, llamando a varios métodos y propiedades que, además, no siempre son recursivos. Por ello, se ha creado esta utilidad para simplificar el proceso.

Se ha implementado un método de extensión para la clase nativa *Node*, lo que permite utilizarlo directamente sobre cualquier nodo y aplicar la acción de forma recursiva a todos sus hijos.

Esta función activa o desactiva la visibilidad, la detección de colisiones, el procesamiento (*_Process()*, *_PhysicsProcess()*, *_Input()*, etc.) y el *ProcessMode*[12] lo alterna entre *Inherit* y *Disabled*.

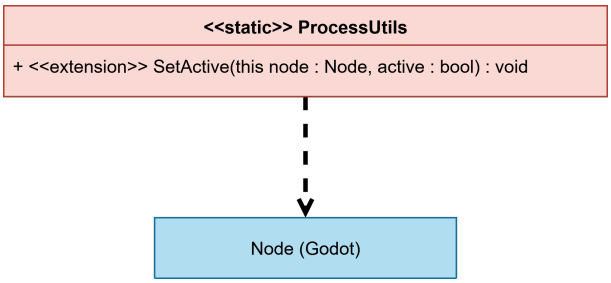


Figura 59: Representación UML de la clase *ProcessUtils*.

[12]El *ProcessMode* determina cómo se procesan los nodos en Godot. Al ser posible pausar todo el árbol de nodos del juego (explicado en 4.19.2), el *ProcessMode* de cada nodo indica como debe actuar según el estado del árbol.

4.4.6. *Logger*

Se ha implementado una clase con métodos estáticos llamada *Logger* que facilita la impresión de mensajes en la consola de depuración de Godot. Esta clase permite categorizar los mensajes en diferentes niveles: *Info*, *Warning*, *Error* y *Debug*.

Además del mensaje en sí, los métodos requieren de pasar como parámetro el *Type* de la clase desde la que se está llamando, lo que permite identificar fácilmente el origen del mensaje en la consola.

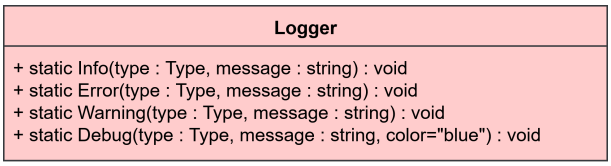


Figura 60: Representación UML de la clase *Logger*.

4.4.7. Rutas del proyecto y traducciones

Las clases estáticas *ProjectRoutes*, *ProjectScenes*, *ProjectMusic*, *ProjectMaterials* y *TranslationKeys* proporcionan listas de constantes públicas que contienen las rutas de los recursos más importantes del proyecto, algunas *translation_id* utilizadas en las traducciones y los nombres de las rutas más relevantes de las carpetas del proyecto. Esto facilita el acceso a estos recursos desde cualquier parte del código sin tener que escribir las rutas manualmente, reduciendo así la posibilidad de errores tipográficos y mejorando la mantenibilidad del código.

También podemos destacar que las escenas, musica y materiales necesitan un proceso de carga para ser usados en el código ya que son archivos que se cargan a partir de su ruta. Por ello, estas referencias ahorran en tener que realizar esta carga cada vez que se necesiten, sino que ya están cargadas y listas para usar.

4.5. *Shaders* del proyecto

En la carpeta *Shaders* dentro de *Assets*, se encuentran un par de *shaders* para lograr efectos específicos en los *sprites* e interfaz de usuario.

Como ya explicó en la sección 2.4.2, los *shaders* en Godot se escriben en un lenguaje nativo llamado *Godot Shader Language*. A continuación, se describen los dos *shaders* principales

utilizados en el proyecto:

4.5.1. *Shader* de desenfoque

Este *shader* se utiliza para crear un efecto de desenfoque en el fondo que aparece detrás de los menús, como el de pausa y el de ajustes. El shader toma la textura de la pantalla y aplica un desenfoque ajustable mediante el parámetro *blur_amount*, permitiendo controlar la intensidad del efecto. Además, incluye parámetros para oscurecer el fondo (*darkness*) y ajustar el brillo y contraste (*brightness_contrast*), lo que facilita adaptar el resultado visual a diferentes necesidades de la interfaz.

Estos parámetros pueden modificarse fácilmente desde el editor de Godot, permitiendo personalizar el efecto según el contexto. El desenfoque ayuda a centrar la atención en los elementos de la interfaz, mientras que el oscurecimiento evitan que el fondo distraiga al jugador.

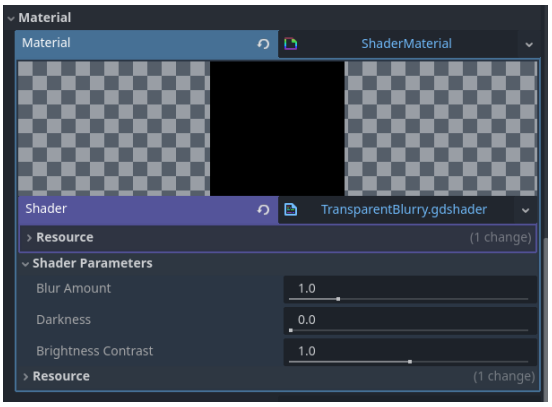


Figura 61: *Shader* de desenfoque aplicado como material desde el editor de Godot.

Figura 62: Demostración en el juego del *Shader* de desenfoque.

4.5.2. *Shader* de resplandor (*Glow*)

Este *shader* se utiliza para crear un efecto de resplandor alrededor de los *sprites*, como el jugador y los enemigos. El efecto se logra acumulando la transparencia (*alpha*) de los píxeles cercanos al borde del *sprite* y aplicando un color de resplandor configurable. Los parámetros permiten ajustar el color del resplandor (*glow_color*), la intensidad (*glow_power*), la distancia de desplazamiento (*glow_shift*) y el radio de extensión (*glow_radius*).

El *shader* detecta los bordes del *sprite* y aplica el color de resplandor en esas zonas, mezclándolo con el color base del *sprite*. Además, si el *sprite* es casi blanco, el resplandor se tiñe con el color especificado. Todos los parámetros pueden modificarse desde el editor de Godot, permitiendo personalizar el efecto para cada entidad.

Este *shader* no se utiliza sin más aplicándolo desde el editor, sino que se utiliza un sistema de animación (explicado mejor en la sección 4.14.2) para modificar sus parámetros en tiempo real, logrando efectos dinámicos como pulsaciones brillantes con cambios de color.

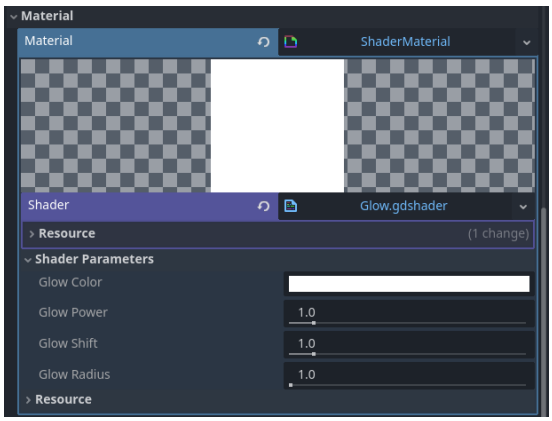


Figura 63: *Shader* de resplandor aplicado como material desde el editor de Godot.

Figura 64: Demostración en el juego del *Shader* de resplandor.

4.6. Reproductores de audio

En Godot existen tres tipos de nodos para reproducir audio: *AudioStreamPlayer*, *AudioStreamPlayer2D* y *AudioStreamPlayer3D*. La diferencia principal entre ellos es que el primero reproduce el sonido de manera global, sin importar la posición del nodo en el espacio, mientras que los otros dos reproducen el sonido en función de la posición del nodo en el espacio 2D o 3D, respectivamente. Esto significa que el volumen y la panorámica del sonido pueden variar según la distancia y la dirección del oyente (la cámara en este caso).

En este proyecto se emplean los dos primeros tipos de nodos para reproducir audio. Dado que *AudioStreamPlayer* y *AudioStreamPlayer2D* no comparten la misma jerarquía en el árbol de nodos de Godot, se han implementado dos clases específicas: *AudioPlayer* y *AudioPlayer2D*. Ambas clases ofrecen la misma funcionalidad y métodos, pero cada una hereda del nodo correspondiente.

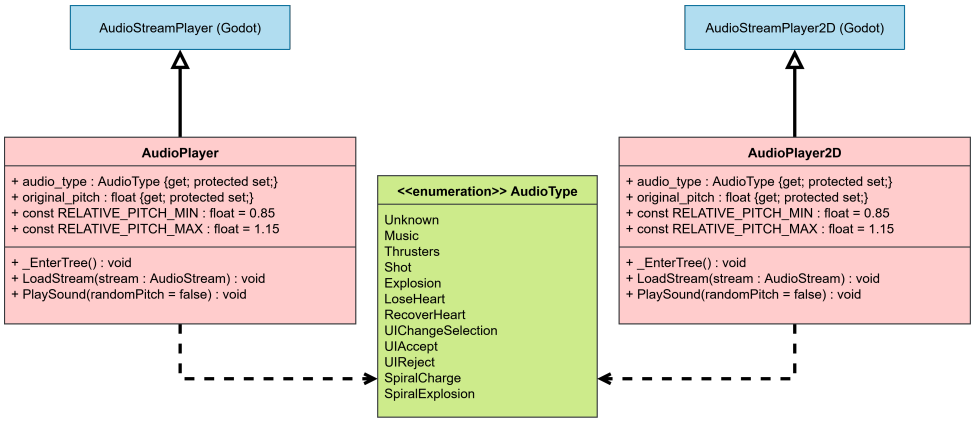


Figura 65: Representación UML de la clase *AudioPlayer* y *AudioPlayer2D*.

Estas clases asocian el reproductor a un tipo de audio (*AudioType*, explicado en la sección 4.4.1), lo que configura automáticamente a qué bus de audio debe dirigirse el sonido.

Incluye métodos para cargar la pista de audio que debe reproducir y para iniciar el sonido, con la opción de variar el tono (*pitch*) de forma aleatoria en un cierto rango para aportar variedad a los efectos sonoros repetitivos, esto es una técnica común para evitar que los sonidos se vuelvan monótonos y fatiguen al oyente.

4.6.1. Gestor de audio

El gestor de audio (*AudioManager*) es una clase que se encarga de manejar una lista de reproductores de audio (*AudioPlayer* y otra de *AudioPlayer2D*) para manejarlos según lo que se necesite en cada momento. Se debe instanciar como hijos suyos los reproductores que se vayan a utilizar en el gestor específico, ya que no los crea automáticamente. Un detalle importante es que no debería haber dos reproductores del mismo tipo (*AudioType*) y clase (*AudioPlayer* o *AudioPlayer2D*) en el gestor, ya que no está diseñado para manejar múltiples reproductores del mismo tipo porque no se requiere en ningún caso del juego.

Una vez los tiene localizados, proporciona métodos para: reproducir o parar uno de los reproductores, parar todos los reproductores, comprobar si un reproductor está reproduciendo sonido, cambiar el archivo de audio que debe reproducir un reproductor y comprobar si un reproductor existe en el gestor. Los métodos que requieren especificar el reproductor necesitan saber el *AudioType* asociado al reproductor y si es de tipo *AudioPlayer* o *AudioPlayer2D*, si no se puede identificar cuál es el reproductor al que se quiere acceder saltará un aviso mediante

el *Logger*.

Esta clase es utilizada en muchas partes del juego: para reproducir los sonidos que hacen los enemigos y el jugador, ciertos sonidos de entorno, los sonidos de las interfaces, la música, etc.

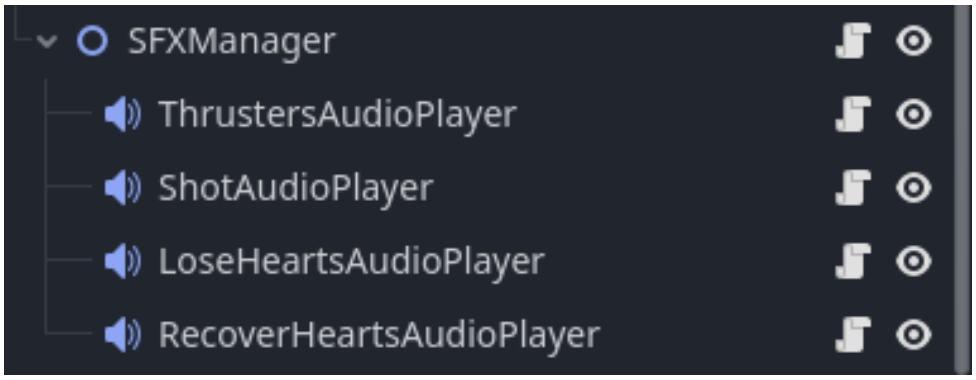


Figura 66: Ejemplo de un *AudioManager* con varios reproductores en Godot.

4.6.2. Gestores de audio globales

Hay ciertos sonidos del juego que actúan de una manera más global, como son la música y los sonidos de las interfaces. Para ello se han creado dos gestores de audio que extienden de *AudioManager* y que actúan como *singleton*, uno para la música (*GlobalMusicManager*) y otro para los efectos de sonidos que sean necesarios (*GlobalSFXManager*). Utilizando este patrón de diseño, su acceso es mucho más sencillo en cualquier parte del código y se ahorra repetir tener nodos que reproducen los mismos sonidos.

Ambos gestores están instanciados como hijos del nodo *GameAudio*, pero esto es solo para tenerlos organizados en el árbol de nodos.

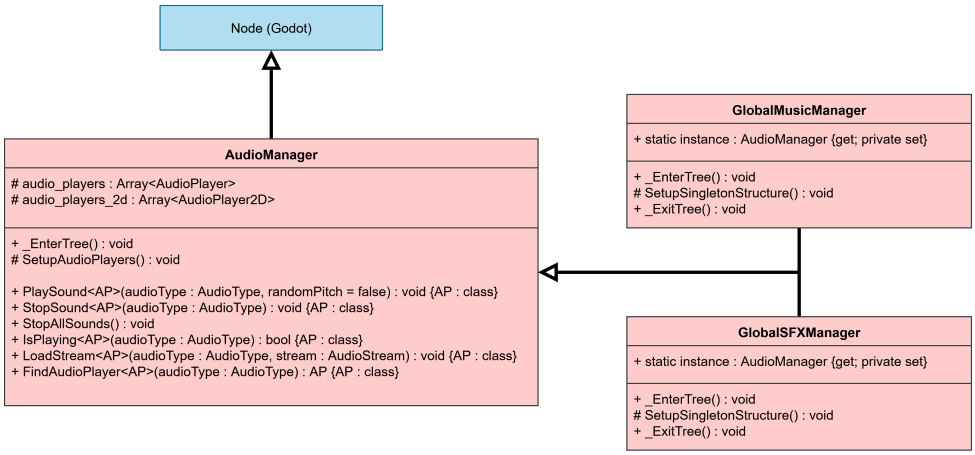


Figura 67: Diagrama de clase de *AudioManager* y sus derivados.

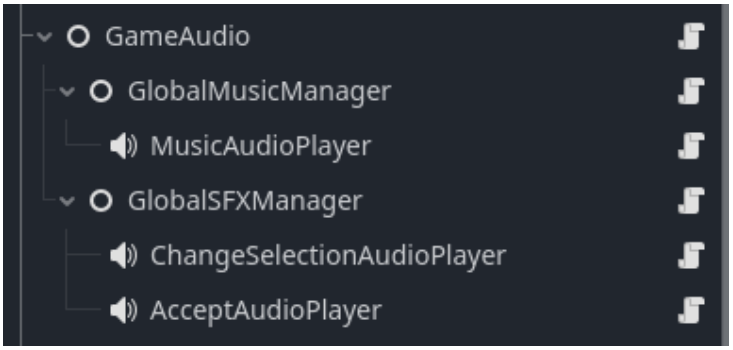


Figura 68: Árbol de nodos de los gestores de audio globales en Godot.

4.7. Entidades

En videojuegos, una entidad es cualquier objeto que tiene una presencia en el mundo del juego y puede interactuar con otros objetos o con el jugador. Las entidades pueden ser personajes (incluido el jugador), enemigos, objetos, *items*[13], entre otros.

En este juego, las entidades son aquellos nodos que heredan de la clase *Entity*, la cuál se refiere a todos aquellos seres que poseen salud y pueden morir, es decir, el jugador y los enemigos. Se ha optado por no incluir a los proyectiles y otros elementos como entidades, ya que no poseen estas cualidades, lo que simplifica la gestión.

La clase *Entity* es una de las más complejas y extensas del proyecto, ya que engloba muchas funcionalidades y sistemas necesarios para que las entidades puedan comportarse co-

[13]Un *item* es un objeto que puede ser recogido o utilizado por el jugador, como armas, armaduras o pociones.

rrectamente en el juego. La clase es abstracta porque no se instancia directamente, sino que se hereda para crear clases específicas para el jugador y los diferentes tipos de enemigos, explicados en la sección 4.16.

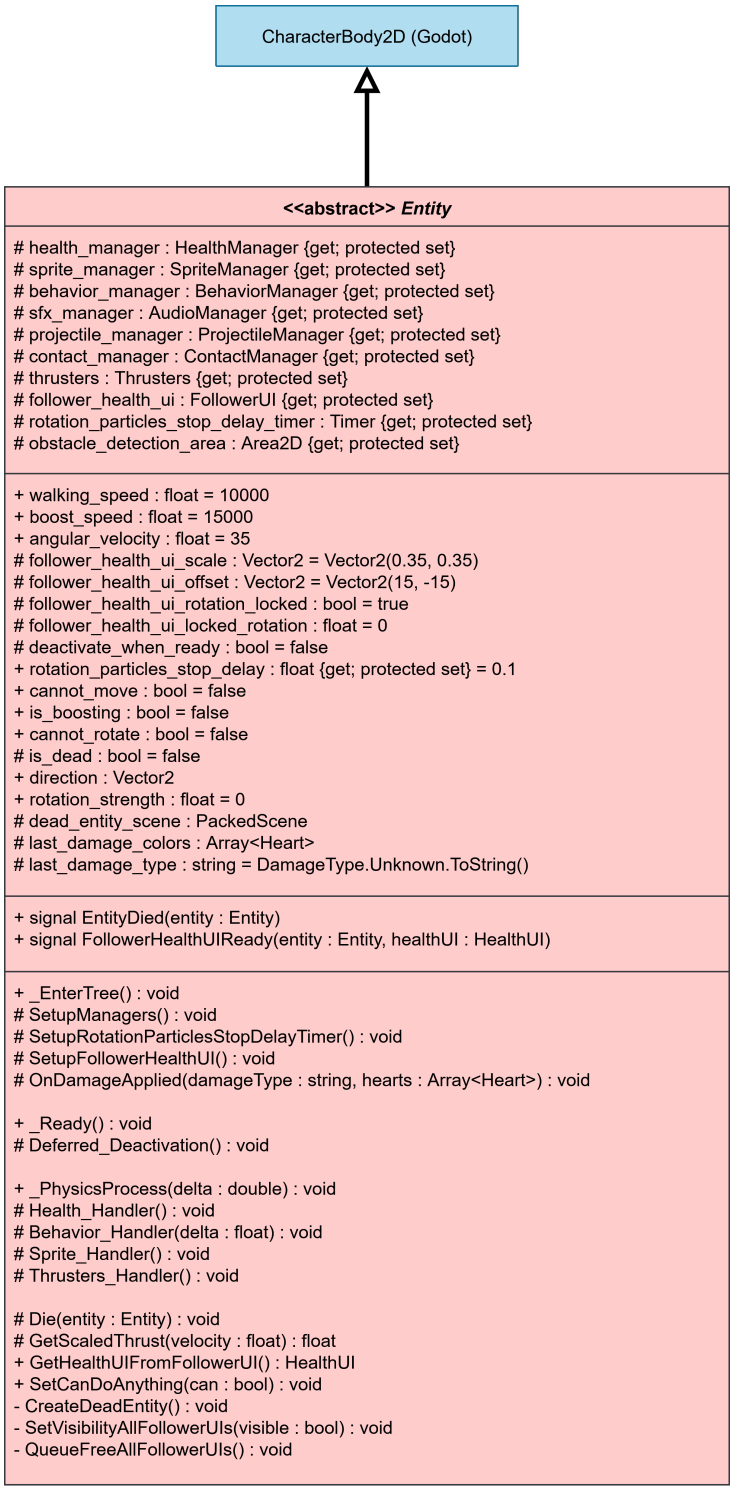


Figura 69: Representación UML de la clase *Entity*.

Extiende de *CharacterBody2D*, que es un tipo de nodo en Godot diseñado para personajes que se mueven y colisionan en un entorno 2D, proporcionando funcionalidades básicas para el movimiento y la detección de colisiones. Debido a esto, es necesario que tenga un nodo hijo de tipo *CollisionShape2D* que defina el área y forma de colisión de la entidad. Para mayor filtrado, este área pertenece a la capa *PhysicsDetectionArea* entre las *Physics Layers* del proyecto (explicadas en la sección 4.2).

Las siguientes secciones describen los diferentes sistemas y gestores que componen la clase *Entity* y así entender mejor su completo funcionamiento.

4.8. Gestor de salud

El gestor de salud (*HealthManager*) es una clase de gran importancia que es utilizada por *Entity* para manejar todo lo relacionado con la salud de esta. También es abstracta y se hereda para crear dos clases específicas: *PlayerHealthManager* y *EnemyHealthManager*, que añaden funcionalidades particulares para el jugador y los enemigos, respectivamente.

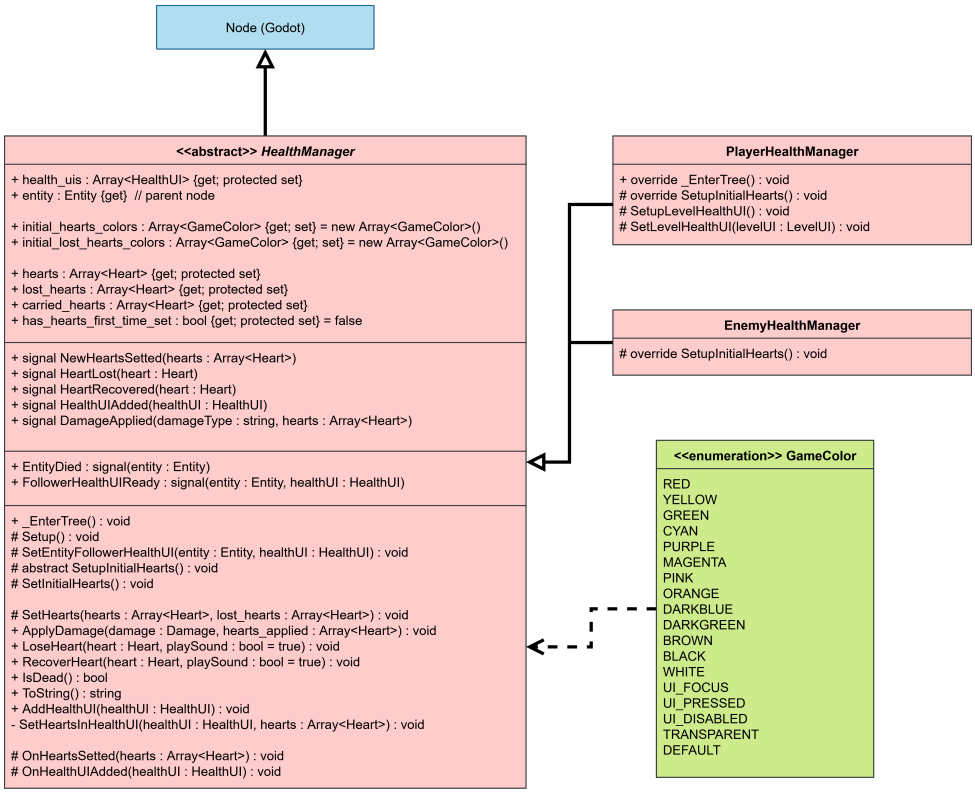


Figura 70: Representación UML de la clase *HealthManager*.

La clase se encarga de gestionar la salud de la entidad mediante un sistema de tres listas de corazones: los corazones totales que puede tener, los corazones que posee actualmente y los corazones que ha perdido. Cada corazón está representado por un color (definido por el *enum GameColor*, explicado en la sección 4.4.2) y nunca puede haber dos corazones del mismo color en una misma lista.

Permite configurar los colores iniciales de los corazones y los corazones perdidos al inicio.

Proporciona métodos para aplicar daño, perder y recuperar corazones, actualizando la interfaz de salud en tiempo real, indicando reproducir ciertos efectos de sonido y emitiendo *signals* para notificar estos cambios al resto del sistema.

Incluye validaciones para asegurar que los colores de los corazones sean únicos y que los corazones perdidos o recuperados pertenezcan al conjunto total de corazones de la entidad.

Además, gestiona múltiples interfaces de salud (llamadas *HealthUI*, explicada en la sección 4.19.4), permitiendo que los cambios en los corazones se reflejen en todas ellas.

El método abstracto *SetupInitialHearts()* se implementa en las clases derivadas para definir cómo se configuran los corazones iniciales según el tipo de entidad. En el caso del jugador, los corazones se establecen en función de la configuración específica del nivel (explicado en la sección 4.20.1), mientras que en los enemigos se asignan directamente desde el editor de Godot, editando las variables *initial_hearts_colors* e *initial_lost_hearts_colors*. De este modo, cada entidad puede tener una configuración de salud personalizada y adaptada a sus características.

4.8.1. Corazones y elementos de un conjunto

Los corazones que representan la salud de las entidades se gestionan mediante la clase *Heart*, que hereda de *SetElement*. Cada corazón está asociado a un color específico (*GameColor*), lo que permite distinguir varias instancias de corazones.

La clase *Heart* proporciona métodos estáticos para crear listas de corazones a partir de una lista de colores, comprobar que los colores sean únicos, buscar corazones por color y validar que los corazones perdidos pertenecen al conjunto total de corazones de la entidad. Estas funciones garantizan la integridad del sistema de salud y evitan errores como duplicidad de colores o asignaciones incorrectas.

Además, se incluyen utilidades para obtener el índice de un corazón en una lista, verificar si un corazón está presente y extraer la lista de colores de los corazones actuales. De este modo,

el sistema permite una gestión flexible y segura de los corazones, facilitando la personalización y el control de la salud de cada entidad.

La clase base *SetElement* sirve como elemento genérico para conjuntos, permitiendo ampliar el sistema en el futuro si se requieren otros tipos de elementos además de los corazones.

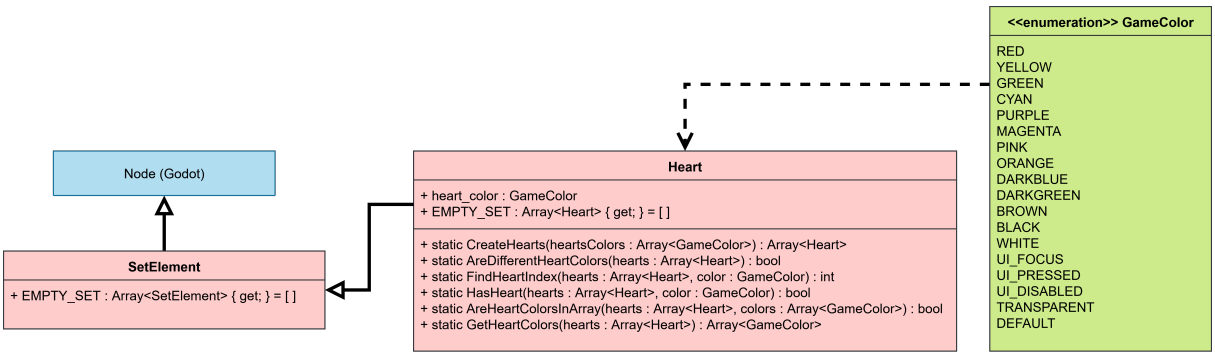


Figura 71: Representación UML de la clase *Heart* y su clase base *SetElement*.

4.9. Entidad muerta

Entity comprueba constantemente si ha muerto, revisando si su *HealthManager* tiene todos los corazones perdidos. Cuando esto ocurre, se llama al método *Die()*, que gestiona el proceso de muerte de la entidad.

Al morir, la entidad desactiva sus movimientos y emite una *signal* para notificar al resto del sistema. Se instancia una escena especial en la misma posición de la entidad, que representa la animación de muerte: el *sprite* de la entidad dividido en varias piezas que se dispersan y rotan, se emiten partículas para simular escombros y se reproduce el sonido de explosión correspondiente. Finalmente, la entidad original se elimina del árbol de nodos y la animación de muerte se encarga de desaparecer cuando termina.

Esta animación de muerte se gestiona mediante la clase abstracta *DeadEntity*, que permite configurar la velocidad de movimiento y rotación de las piezas, el tiempo y la velocidad del efecto de desvanecimiento (*fade*), y la emisión de partículas especiales si la muerte se produce por conjunto vacío (*empty set*, explicado en la sección 4.10.1), las cuales son el símbolo del conjunto vacío (un círculo con una barra diagonal). La entidad muerta además tiene el color del último daño recibido que causó su muerte, afectando tanto a las piezas como a las partículas.

El sistema inicializa las piezas y partículas, asigna direcciones y rotaciones aleatorias, reproduce el sonido, controla la animación de dispersión y el desvanecimiento progresivo de las piezas. Cuando todas las piezas se han vuelto completamente transparentes, la escena se elimina automáticamente.

Existen clases derivadas para cada tipo de entidad (*DeadPlayer*, *DeadEnemy*, *DeadAsteroid*), permitiendo personalizar el efecto visual según el tipo de entidad que ha muerto. Aunque actualmente no hay diferencias y se ha dejado este aspecto para futuras mejoras.

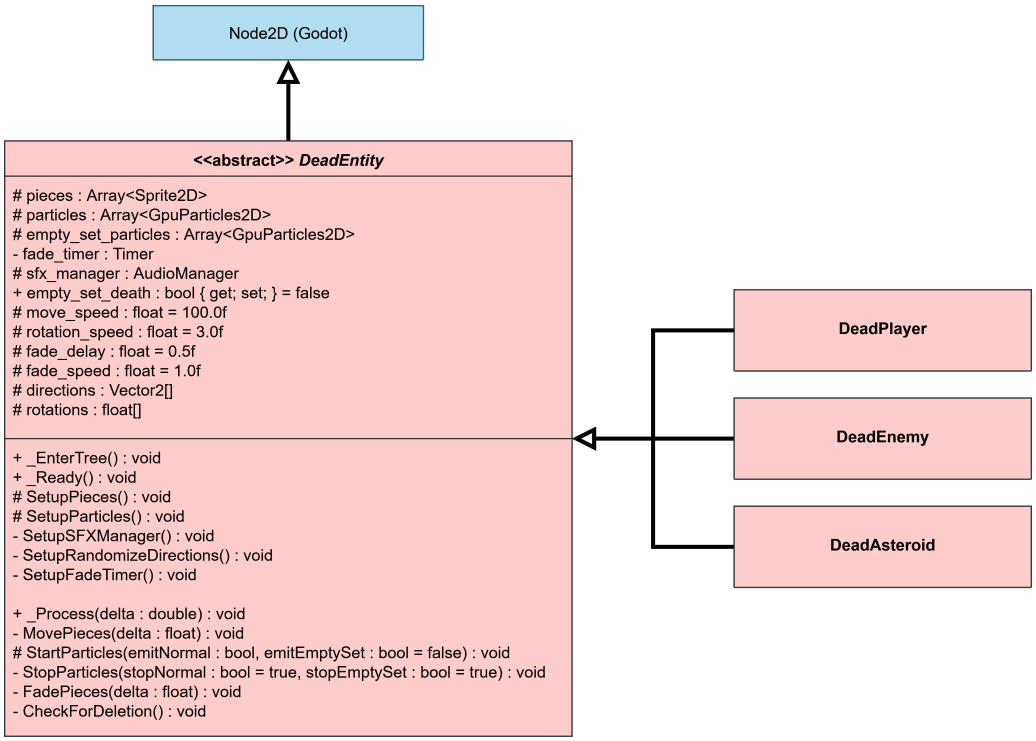


Figura 72: Diagrama de clase de *DeadEntity* y sus derivadas.

Existen en total de nueve escenas diferentes de animaciones de muerte, una para cada entidad (incluyendo los diferentes *sprites* que pueden tener los enemigos).

Cuando una entidad tiene una escala diferente a 1, la animación de muerte se ajusta automáticamente para que las piezas mantengan la misma proporción que la entidad original. También se adapta si la entidad está rotada, de modo que las piezas se dispersen correctamente en función de la orientación inicial.

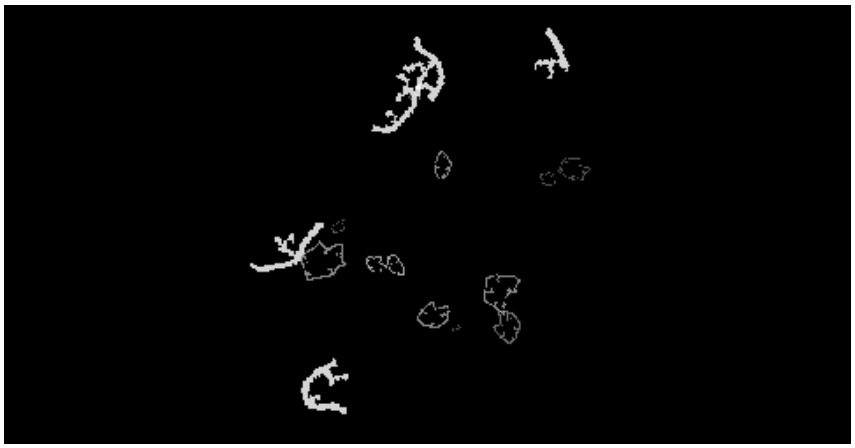


Figura 73: Demostración en el juego de la animación de muerte de un asteroide.

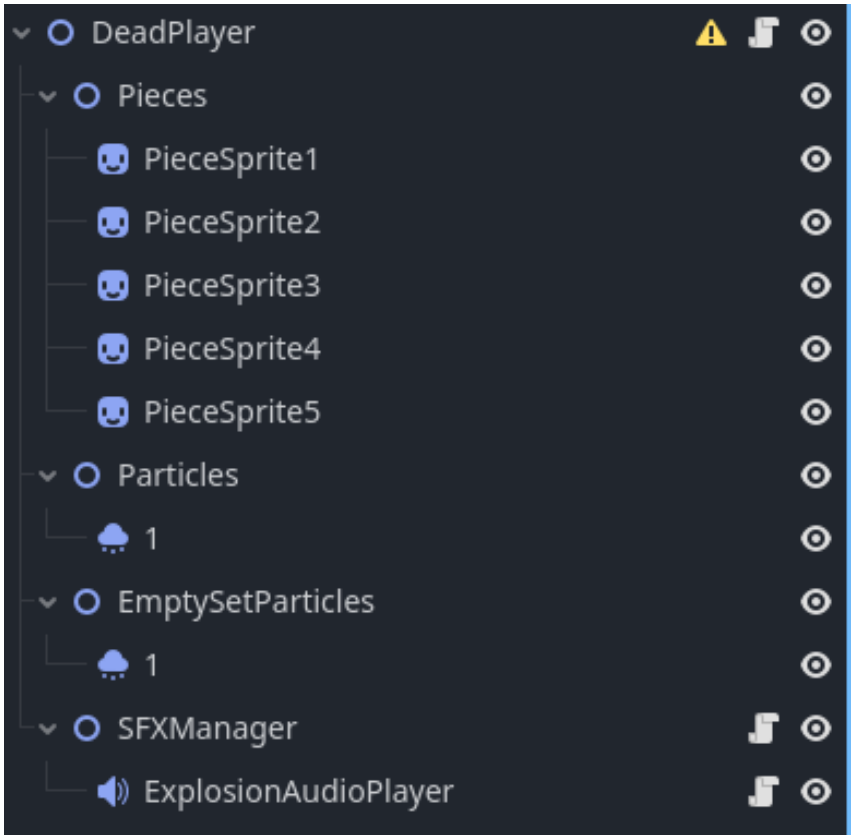


Figura 74: Árbol de nodos de la escena de *DeadPlayer* en Godot.

4.10. Proyectiles

Los proyectiles son elementos clave en el juego, ya que tanto el jugador como los enemigos pueden dispararlos para atacar. La clase abstracta *Projectile* gestiona el comportamiento gene-

ral de todos los proyectiles: movimiento, rotación, colisión, tiempo de vida y efectos visuales asociados.

Cada proyectil puede portar uno o varios corazones, que pueden influir en sus mecánicas de daño, aunque actualmente el uso de más de un corazón no se utiliza en el juego pero funciona.

Los proyectiles siempre avanzan en una dirección fija y a una velocidad constante, hasta que colisionan con una entidad o agotan su tiempo de vida.

Se puede configurar la velocidad, dirección y duración de cada proyectil, así como el propietario que lo ha disparado, evitando que los proyectiles dañen a su propio dueño o los aliados de este.

La clase gestiona dos tipos de partículas: las de estela (*trail particles*), que se emiten mientras el proyectil está en movimiento, y las de impacto (*impact particles*), que se activan al colisionar con una entidad. Ambos efectos se configuran automáticamente al instanciar el proyectil y se eliminan de forma eficiente tras su uso.

La clase incluye validaciones para evitar colisiones repetidas, gestiona la aplicación de daño a la entidad impactada y asegura que los efectos visuales se mantengan sincronizados con el estado del proyectil.

El color de un proyectil y el de sus partículas está determinado por el color del corazón que tienen (actualmente si se usase más de un corazón el proyectil es blanco y no una mezcla de los colores, debido a que esta mecánica no se utiliza como ya se mencionó). Si un proyectil no tiene corazones, se le asigna el color blanco por defecto también, pero esto es intencional para que ciertos proyectiles.

Para determinar el tipo de daño de un proyectil se hace uso de un *enum* llamado *DamageType*, que define los diferentes tipos de daño en el juego, como se explica en la sección 4.10.1.

El método de *CreateProjectile()* (estático) permite crear proyectiles con parámetros personalizados, como el tipo de daño, la dirección, los corazones transportados, el propietario, la velocidad y la duración. Además, el sistema utiliza un *mapping* para asociar cada tipo de daño con la escena del proyectil correspondiente, facilitando la instanciación.

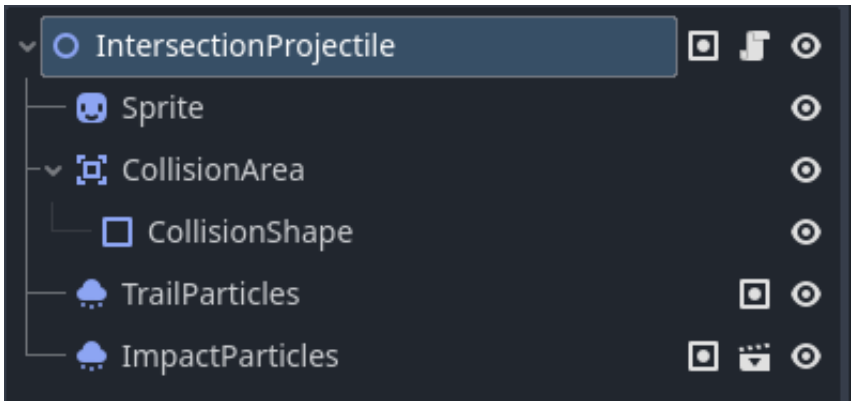


Figura 75: Ejemplo del árbol de nodos de la escena de un *Projectile* en Godot.

4.10.1. Daño

El sistema de daño del juego está basado en operaciones de conjuntos, representadas por el *enum DamageType*. Cada tipo de daño corresponde a una operación matemática distinta: diferencia, unión, intersección, diferencia simétrica y complemento. Esto permite que los proyectiles y otras fuentes de daño interactúen con la salud de las entidades de formas variadas y estratégicas.

La clase abstracta *Damage* define los aspectos comunes para todos los tipos de daño y proporciona métodos estáticos para crear instancias específicas según el tipo requerido. La clase no hereda de ningún nodo de Godot, ya que no es necesario que tenga presencia en el árbol de nodos, solo se utiliza para encapsular la lógica de daño y aplicarla a las entidades.

Cada clase derivada implementa el método *Apply()*, que recibe la entidad objetivo y los corazones aplicados, y ejecuta la lógica correspondiente:

- **DifferenceDamage**: Elimina los corazones de la entidad que coinciden en color con los corazones aplicados.
- **UnionDamage**: Recupera los corazones perdidos de la entidad que coinciden en color con los corazones aplicados.
- **IntersectionDamage**: Elimina los corazones que no coinciden en color con los aplicados y recupera los corazones perdidos que sí coinciden.
- **SymmetricDifferenceDamage**: Elimina los corazones que coinciden en color con los aplicados y recupera los corazones perdidos que también coinciden.

- **ComplementDamage**: Elimina todos los corazones actuales y recupera todos los corazones perdidos de la entidad, independientemente del color. Es decir, alterna toda la salud de la entidad.

Si el daño de *Intersection* aplica un conjunto vacío (es decir, ningún corazón), el daño siempre causa la muerte de la entidad, lo que lo hace especialmente peligroso. Este tipo de muerte se llama *muerte por conjunto vacío* y causa que en los efectos visuales de la *DeadEntity* se emitan partículas especiales para indicar esta forma de morir. Este tipo de daño lo utilizan los asteroides y los límites del mapa.

La clase *Damage* incluye un método estático *CreateDamage()* que utiliza un *mapping* para asociar cada tipo de daño (*DamageType enum*) con su clase correspondiente, facilitando la creación y uso de estos no solo en los proyectiles, sino en cualquier otra parte del juego que requiera aplicar daño de cierta forma.

Una ventaja adicional de utilizar *enums* junto con *mappings* para instanciar clases derivadas es que, en Godot, el editor no permite seleccionar directamente clases implementadas como propiedades de los nodos. Sin embargo, sí permite seleccionar valores de *enum*, lo que facilita la configuración desde el editor y la integración con el sistema de instanciación dinámica en el código.

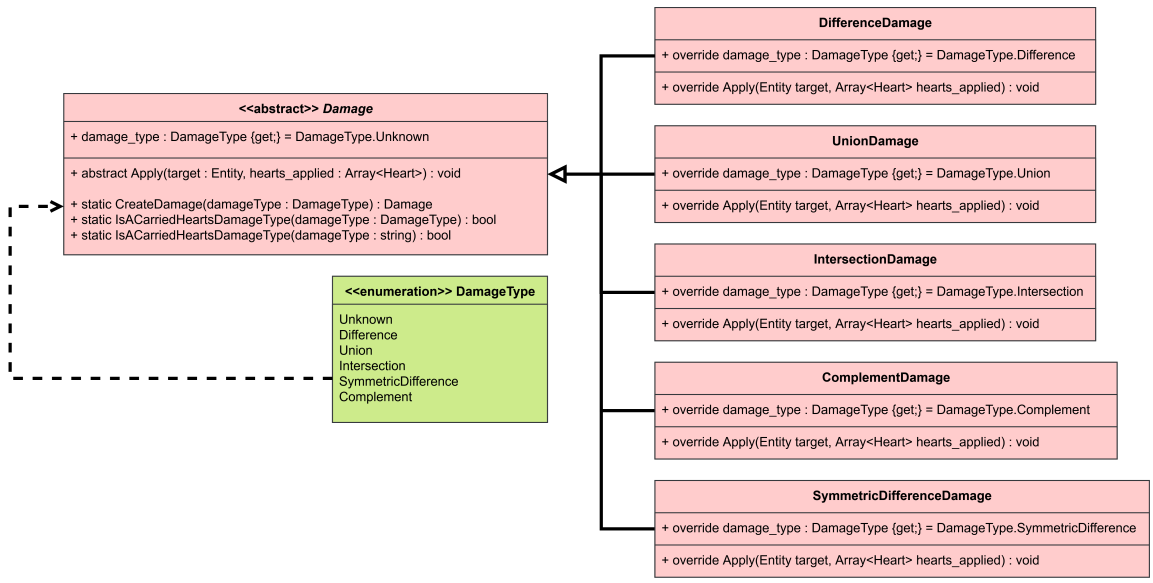


Figura 76: Diagrama de clase de *Damage* y sus derivados.

4.10.2. Tipos de proyectiles

Cada tipo de proyectil en el juego está asociado a una operación de daño concreta, definida por el *enum DamageType*. Para ello, se crean clases derivadas de *Projectile* que sobrescriben la propiedad *damage_type* con el valor correspondiente.

En el caso del proyectil de tipo *Complement*, se sobrescribe además la propiedad *can_carry_hearts* a *false*, ya que este tipo de daño no utiliza corazones transportados.

Cada clase de proyectil se asocia en el editor de Godot con su escena correspondiente, lo que permite instanciar el tipo adecuado según la mecánica de juego. La lógica de aplicación del daño y la gestión de corazones se realiza automáticamente mediante las clases *Damage* y *Projectile*, de modo que al disparar un proyectil, este aplica el efecto correcto sobre la entidad objetivo sin necesidad de código adicional.

Este sistema facilita la extensión y el mantenimiento del juego, permitiendo añadir nuevos tipos de proyectiles o modificar los existentes de forma sencilla y centralizada.

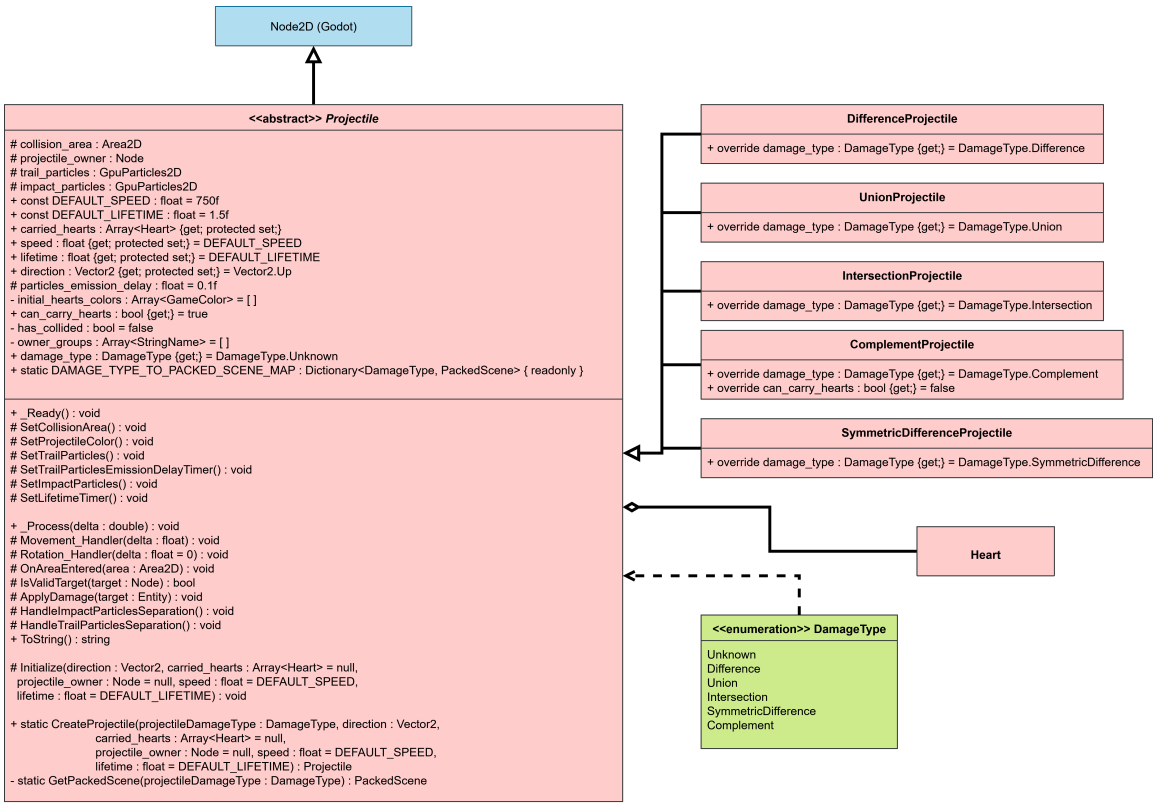


Figura 77: Diagrama de clase de *Projectile* y sus derivados.

Para más información sobre los tipos de proyectiles y el como funciona en detalle cada

uno de los tipos de daño se puede consultar en el GDD (Apéndice B, apartado 9.4.1), es muy recomendable.

4.11. Gestor de proyectiles

El gestor de proyectiles (*ProjectileManager*) es el encargado de controlar el disparo de proyectiles para cada entidad. Esta clase gestiona la cadencia de disparo mediante temporizadores, los puntos de disparo desde los que se instancian los proyectiles y el tipo de proyectil y el color de corazón que porta cada disparo.

Permite configurar la velocidad y duración de los proyectiles desde el editor, así como la estrategia de disparo que se utiliza (*FiringStrategyType*) que determina cómo se distribuyen los disparos entre los diferentes puntos. El gestor mantiene listas de los tipos de proyectiles disponibles y los colores de corazones que pueden portar, permitiendo alternar entre ellos de forma circular y controlada por temporizadores para evitar cambios demasiado rápidos.

Además, al morir la entidad, el gestor desactiva automáticamente la capacidad de disparar y de cambiar proyectil o color.

El gestor se integra con la interfaz de usuario (*ProjectilesUI*, explicada en la sección 4.19.4), actualizando en tiempo real los iconos y colores mostrados según los cambios realizados por la entidad. Aunque solo es utilizado por el jugador, ya que los enemigos no tienen interfaz creada.

Existen dos gestores específicos: *PlayerProjectileManager* para el jugador y *EnemyProjectileManager* para los enemigos. El primero se conecta con la interfaz de usuario *ProjectilesUI* y configura los colores y tipos de proyectiles iniciales según la configuración del nivel (4.20.1), mientras que el segundo solo inicializa los valores directamente según lo configurado desde el editor (de la misma forma que se hace con el gestor de salud).

Este sistema modular permite una gestión flexible y eficiente de los proyectiles, facilitando la personalización de las mecánicas de disparo y la integración con el resto de sistemas del juego.

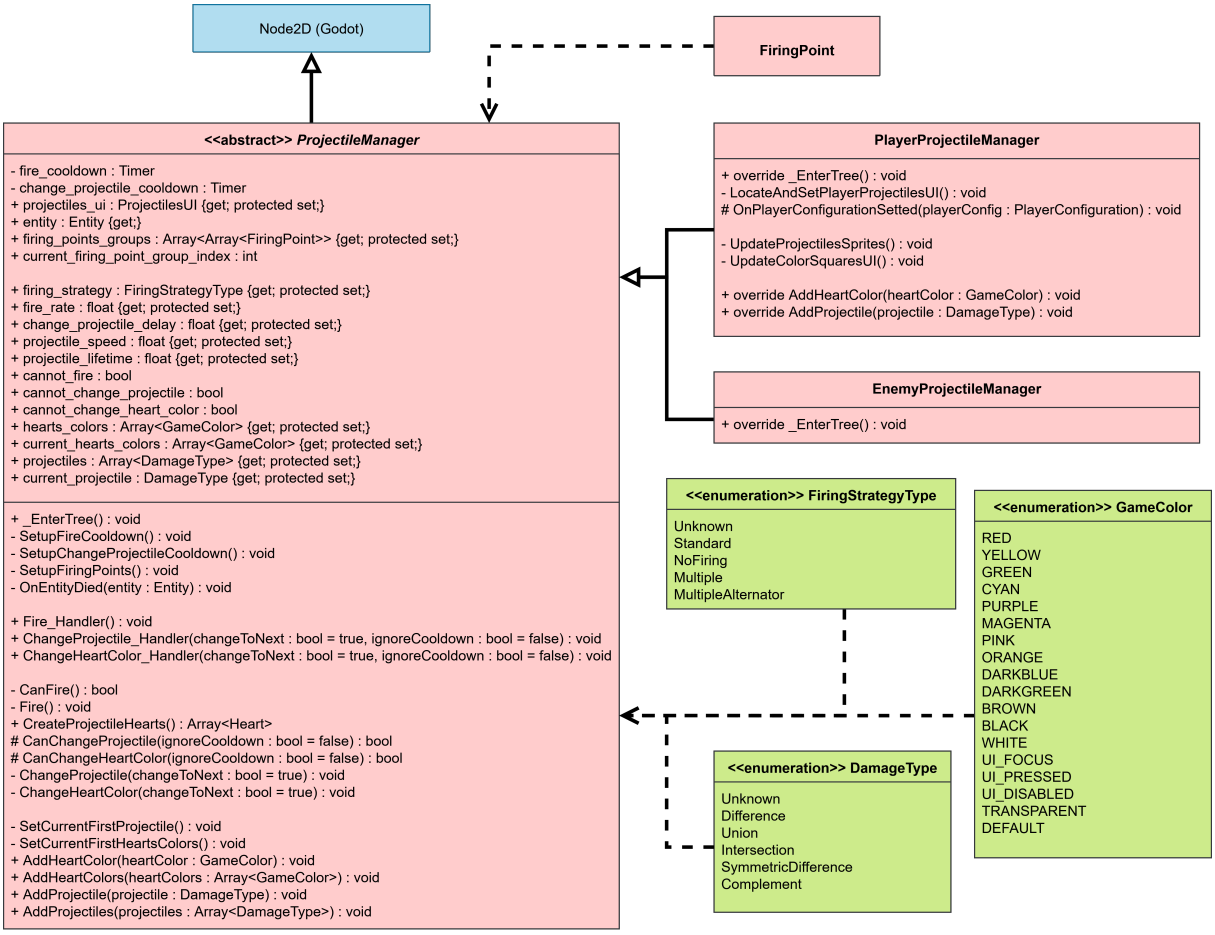


Figura 78: Diagrama de clase de *ProjectileManager* y sus derivados.

Para que una entidad reciba el impacto de un proyectil hace uso de un par de nodos especiales: *Area2D* y *CollisionShape2D*, el segundo siendo hijo del primero. El nodo *Area2D* detecta cuando otro cuerpo (en este caso un proyectil, que tiene otro par de nodos iguales) entra en su área de colisión, y el nodo *CollisionShape2D* define la forma y tamaño de esa área (depende de la entidad). Este área hace uso de las *Physics Layers* (explicadas en la sección 4.2) para definir con qué tipos de cuerpos puede interactuar, en este caso, este área pertenece a la capa *DamageDetectionArea*.

4.11.1. Puntos de disparo

Los puntos de disparo (*FiringPoint*) son nodos hijos del gestor de proyectiles que definen las posiciones y orientaciones desde las cuales se instancian los proyectiles al disparar. Cada punto

de disparo puede tener una rotación específica, permitiendo que los proyectiles se lancen en diferentes direcciones según la configuración del punto. Esto es especialmente útil para crear patrones de disparo variados, como disparos en abanico, en cruz o en línea recta. Además de los puntos, hay que tomar en cuenta que la entidad en sí misma tiene una posición y rotación, que también influyen en la dirección final del disparo.

El gestor de proyectiles mantiene una lista de listas de estos puntos, permitiendo agruparlos y seleccionar diferentes conjuntos según la estrategia de disparo elegida. Al disparar, el gestor instancia los proyectiles en cada punto del conjunto según la configuración actual.

Figura 79: Ejemplo de puntos de disparo en Godot.

Si la lista de listas de puntos de disparo está vacía, lo normal es porque se utilice una estrategia de disparo que no requiere múltiples puntos, como el disparo simple. En este caso, el gestor utiliza la posición y rotación del gestor de proyectiles directamente.

4.11.2. Estrategias de disparo

El sistema de disparo del juego está diseñado de forma modular y extensible, siguiendo una estructura similar al sistema de daño (*Damage*). Cada estrategia de disparo se representa mediante el *enum FiringStrategyType*, que define los distintos modos de disparo disponibles: estándar, sin disparo, múltiple y alternador múltiple.

La clase abstracta *FiringStrategy* define la base común para todas las estrategias y proporciona un método estático para crear instancias específicas según el *enum* indicado. Cada clase

derivada implementa el método *Fire()*, que recibe el gestor de proyectiles (*ProjectileManager*) y ejecuta la lógica de disparo correspondiente:

- **StandardFiringStrategy**: Dispara un único proyectil en la dirección actual del gestor de proyectiles.
- **NoFiringStrategy**: No dispara ningún proyectil, útil para entidades como los asteroides.
- **MultipleFiringStrategy**: Dispara varios proyectiles simultáneamente desde todos los puntos de disparo definidos.
- **MultipleAlternatorFiringStrategy**: Dispara proyectiles desde un grupo de puntos de disparo, alternando entre los diferentes grupos en cada disparo (de ahí el uso de una lista de listas de puntos de disparo).

Cada estrategia se asocia en el editor de Godot mediante el tipo correspondiente, permitiendo que el gestor de proyectiles utilice la lógica adecuada sin necesidad de modificar el código principal. Este sistema facilita la extensión y el mantenimiento del juego, permitiendo añadir nuevas estrategias o modificar las existentes de forma sencilla y centralizada.

La analogía con el sistema de daño permite mantener una arquitectura coherente y flexible, donde tanto el tipo de daño como la estrategia de disparo se gestionan mediante clases especializadas y *mappings*, simplificando la lógica y mejorando la escalabilidad del proyecto.

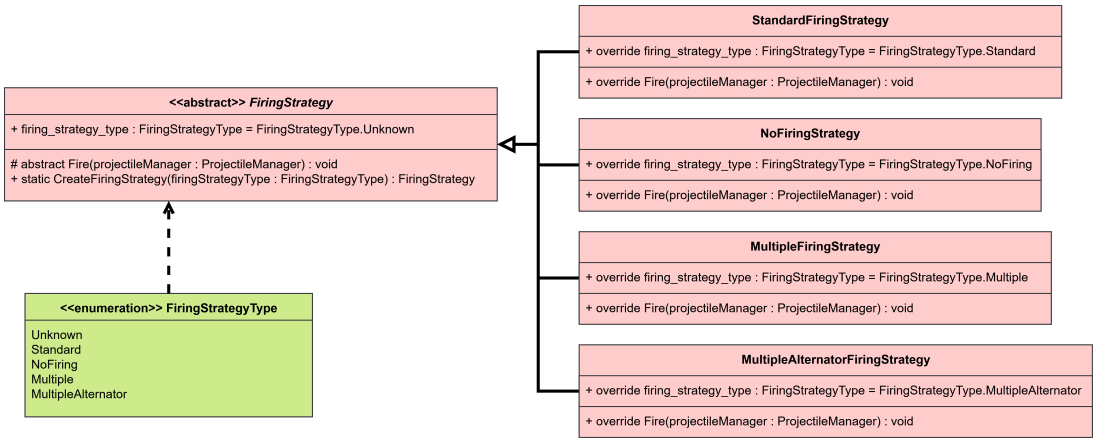


Figura 80: Diagrama de clase de *FiringStrategy* y sus derivados.

4.12. Gestor de contacto

El gestor de contacto (*ContactManager*) es una clase abstracta utilizada para detectar cuándo una entidad entra en contacto con otra, ya sea mediante colisiones físicas o de área. Su función principal es servir como sistema de detección: identifica las entidades con las que se produce el contacto y emite *signals* para que otros sistemas puedan reaccionar según sea necesario.

Este módulo no es obligatorio que las entidades lo tengan para funcionar correctamente como es el caso de que solo lo utilizan los asteroides.

La lógica de respuesta ante el contacto no está definida en el gestor base, sino que se implementa en las clases derivadas según las necesidades de cada entidad. Solo se ha implemetado una en el caso de los asteroides, la clase *AsteroidContactManager* sobrescribe el comportamiento para aplicar daño a la entidad objetivo cuando se detecta un contacto de tipo área, en este caso filtrado a solo detectar al jugador y aplica como daño el tipo *Intersection* con un conjunto vacío, lo que garantiza la muerte inmediata del jugador al colisionar con un asteroide, aunque desde el editor es configurable que hagan el daño que se quiera.

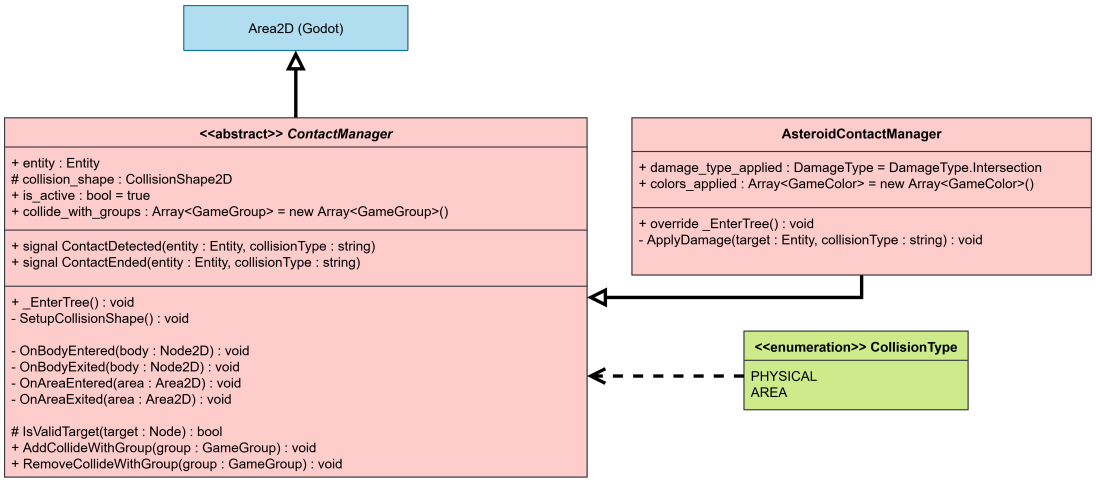


Figura 81: Representación UML de la clase *ContactManager* y sus derivados.

4.13. Gestor de comportamiento

El gestor de comportamiento (*BehaviorManager*) es una clase abstracta que centraliza la gestión de los distintos comportamientos de una entidad en el juego. Su principal función es

coordinar los módulos de movimiento, ataque y otros comportamientos específicos, delegando la lógica a clases especializadas llamadas *Behavior*.

Cada entidad que requiere inteligencia o acciones autónomas (como el jugador, enemigos o asteroides) tiene asociado un gestor de comportamiento, instanciado como nodo hijo en el árbol de la entidad. El gestor detecta automáticamente los módulos de comportamiento presentes (*MovementBehavior*, *AttackBehavior*, etc.) y los inicializa en el método *SetupBehaviors()*, notificando mediante el sistema de *Logger* si alguno no está presente.

El método principal del gestor es *Update(delta)*, que se llama desde el ciclo de procesamiento físico (*PhysicsProcess*) de la entidad. Este método delega la actualización de cada módulo de comportamiento, permitiendo que cada uno gestione su lógica de forma independiente pero sincronizada en cada frame.

Existen gestores específicos para cada tipo de entidad, como *PlayerBehaviorManager*, *EnemyBehaviorManager* y *AsteroidBehaviorManager*, que heredan de la clase base y pueden añadir módulos adicionales según las necesidades. El único gestor que agrega nuevos módulos es *EnemyBehaviorManager*, que incluye un módulo de visión (*VisionBehavior*) que permite detectar al jugador y reaccionar en consecuencia.

Esta arquitectura modular facilita la extensión y el mantenimiento del sistema de comportamientos, permitiendo añadir nuevos tipos de comportamiento o modificar los existentes sin afectar al resto del código. Además, separa claramente la lógica de cada aspecto (movimiento, ataque, visión, etc.), mejorando la legibilidad y la reutilización.

La lógica de comportamiento de los enemigos que se describió en el diagrama de flujo de la sección 3.4.2 se implementa en su totalidad en *EnemyBehaviorManager* y sus tres módulos.

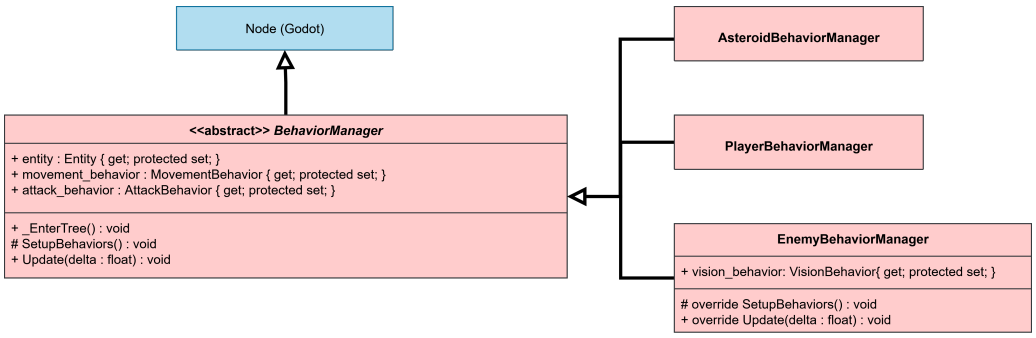


Figura 82: Diagrama de clase de *BehaviorManager* y sus derivados.

4.13.1. Comportamiento

El sistema de comportamiento (*Behavior*) está diseñado para modularizar y separar las distintas acciones que una entidad puede realizar en el juego. Cada comportamiento es una clase derivada de *Behavior*, que representa una acción concreta. Estos módulos se instancian como nodos hijos del gestor de comportamiento (*BehaviorManager*) y se comunican con la entidad a través de referencias directas.

Cada clase de comportamiento implementa el método abstracto *Update(delta)*, que se llama desde el gestor de comportamiento en cada ciclo de procesamiento. Esto permite que cada módulo gestione su lógica de forma independiente y sincronizada, facilitando la extensión y el mantenimiento del sistema.

Existen tres tipos principales de comportamientos:

- **Movimiento (*MovementBehavior*):** Controla el desplazamiento y la rotación de la entidad. Implementa los métodos abstractos *Movement_Handler(delta)* y *Rotation_Handler(delta)*, que definen la lógica específica de movimiento y rotación en las clases derivadas.

- **Ataque (*AttackBehavior*):** Gestiona las acciones de ataque de la entidad, como disparar proyectiles o cambiar el que dispara (maneja el *ProjectileManager* de la entidad). El método abstracto *Attack_Handler(delta)* encapsula la lógica de ataque y se implementa en las clases derivadas según el tipo de entidad.

- **Visión (*VisionBehavior*):** Permite detectar y seguir objetivos en el entorno, como el jugador o enemigos. Mantiene una referencia al objetivo actual y proporciona métodos para obtener su posición y comprobar si es válido. El método abstracto *UpdateTarget(delta)* actualiza el objetivo detectado en cada frame.

Además, es fácil la creación de nuevos tipos de comportamiento sin modificar el resto del sistema, mejorando la escalabilidad y la reutilización del código.

Cada uno de los comportamientos que se han implementado son abstractos porque de ellos se heredan las diversas clases específicas que definen la lógica concreta para cada tipo de entidad, que se explican a continuación.

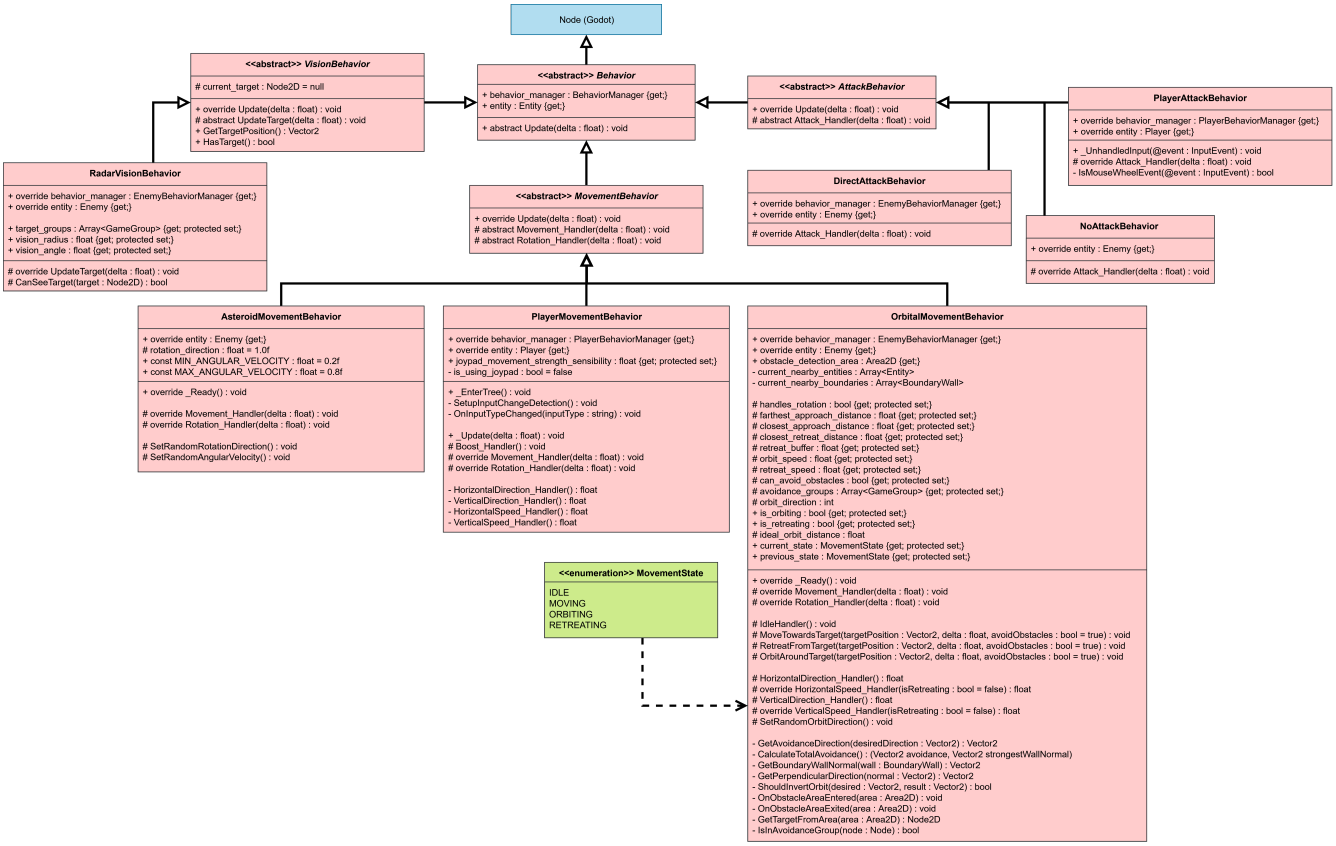


Figura 83: Diagrama de clase de *Behavior* y sus derivados.

4.13.2. Movimiento

Los comportamientos de movimiento (*MovementBehavior*) controlan cómo se desplazan y rotan las entidades en el juego. Cada clase derivada implementa la lógica específica de movimiento y rotación mediante los métodos abstractos *Movement_Handler(delta)* y *Rotation_Handler(delta)*.

Se han implementado tres clases heredadas:

- ***PlayerMovementBehavior***: Gestiona el movimiento del jugador, respondiendo a las entradas del usuario para desplazarse y rotar. Durante el *Process()* del comportamiento se leen las entradas de movimiento para la nave y si se está haciendo que acelere o no (*boost*), la rotación por otro lado se gestiona buscando la posición de la mirilla (*Scope*, explicada en la sección 4.18) y rotando la nave para que apunte hacia esa posición. La lógica de movimiento utiliza interpolación para suavizar las transiciones de velocidad y

dirección, proporcionando una experiencia de control fluida y responsiva.

- ***AsteroidMovementBehavior***: Maneja el movimiento de los asteroides, los cuales no se desplazan pero si rotan constantemente a una velocidad angular fija, que se configura desde el editor o se puede indicar que sea aleatoria (en un rango).

- ***OrbitalMovementBehavior***: Implementa la lógica de movimiento avanzada para los enemigos, permitiendo que se desplacen de forma inteligente en torno a un objetivo, normalmente el jugador. Esta clase utiliza un sistema de estados para alternar entre diferentes modos de movimiento según la distancia al objetivo y la situación del entorno.

  El sistema distingue cuatro estados principales: ***idle*** (reposo), ***moving*** (aproximación), ***retreating*** (retirada) y ***orbiting*** (órbita). La transición entre estos estados se realiza automáticamente en cada ciclo de procesamiento, evaluando la posición del objetivo y la distancia respecto a la entidad. Por ejemplo, si el objetivo está lejos, el enemigo se aproxima; si está demasiado cerca, se retira; y si se encuentra en una distancia intermedia, comienza a orbitar alrededor del objetivo.

  La órbita se realiza calculando un vector tangencial respecto al objetivo, permitiendo que el enemigo rodee al jugador en sentido horario o antihorario, configurable mediante una variable interna que cambia aleatoriamente en ciertos puntos. Además, el sistema ajusta dinámicamente la distancia ideal de órbita, introduciendo variabilidad y evitando movimientos predecibles.

  Una característica avanzada de este comportamiento es la **evasión de obstáculos**, la cual se intentó implementar pero no funciona correctamente, por lo que se descartó usarla y se dejó para una futura mejora (se ha añadido como línea futura, se puede ver en la sección 6.2). La clase detecta entidades (como asteroides) y límites cercanos mediante un área de detección (*Area2D*) y ajusta la dirección de movimiento para evitar colisiones, combinando el vector de movimiento deseado con un vector de evasión calculado. Si la entidad se encuentra bloqueada o la evasión es insuficiente, el sistema puede invertir el sentido de la órbita para buscar una ruta alternativa.

  El sistema está diseñado para ser flexible y configurable desde el editor, permitiendo ajustar parámetros como la velocidad de órbita, la distancia mínima y máxima de apro-

ximación, la velocidad de retirada y los grupos de obstáculos a evitar. Además, utiliza *signals* y métodos públicos para facilitar la integración con otros módulos, como el gestor de comportamiento o el sistema de visión.

La implementación modular y orientada a estados de *OrbitalMovementBehavior* permite crear enemigos con movimientos inteligentes y adaptativos, capaces de perseguir, rodear y esquivar al jugador de forma natural, mejorando la jugabilidad y el desafío del juego.

4.13.3. Ataque

Los comportamientos de ataque (*AttackBehavior*) gestionan las acciones ofensivas de las entidades, como disparar proyectiles o cambiar el tipo o color de proyectil que utilizan. Cada clase derivada implementa la lógica específica de ataque mediante el método abstracto *Attack_Handler(delta)*.

Se han implementado tres clases heredadas:

- ***PlayerAttackBehavior***: Gestiona el ataque del jugador respondiendo a las entradas del usuario. Se hace uso de la función nativa *_UnhandledInput(event)* para detectar cuándo el jugador presiona para realizar las acciones de cambio de tipo y color de proyectil; y se hace uso del método *Process()* para detectar si el jugador está presionando el botón de disparo. También se verifica si se utiliza la rueda del ratón para cambiar el tipo del proyectil para hacer uso del *cooldown* [14] que tiene esta acción, pues si se utilizan botones normales no es necesario.

- ***NoAttackBehavior***: Define el comportamiento para entidades que no realizan ataques, como los asteroides. En este caso, el método de ataque no realiza ninguna acción, permitiendo que la entidad permanezca pasiva y sin capacidad ofensiva.

- ***DirectAttackBehavior***: Implementa el ataque directo de los enemigos, que consiste en disparar proyectiles hacia el objetivo siempre que este sea detectado por el módulo de visión. Si el enemigo tiene un objetivo válido, el comportamiento ordena al gestor de proyectiles que realice el disparo. Este sistema permite que los enemigos ataquen de

[14] El *cooldown* es un tiempo de espera que se aplica después de realizar una acción, durante el cual no se puede volver a realizarla.

forma autónoma y reactiva, adaptándose a la presencia del jugador y siendo la cadencia de disparo controlada por el propio gestor de proyectiles la que ajusta la frecuencia de los disparos.

4.13.4. Visión

El comportamiento de visión (*VisionBehavior*) permite a las entidades detectar objetivos en el entorno, como el jugador o enemigos. Este módulo es especialmente útil para enemigos que necesitan localizar y atacar al jugador de forma autónoma, pues no tiene sentido que el jugador tenga visión, ya que es controlado por el usuario.

Se ha implementado una clase heredada:

- ***RadarVisionBehavior***: Dota a los enemigos de la capacidad de detectar objetivos en su entorno mediante un sistema de "radar". Este módulo permite configurar los grupos de nodos que pueden ser detectados (por ejemplo, el jugador), el radio máximo de visión y el ángulo de detección, haciendo posible tanto una visión omnidireccional como limitada a un cono frontal.

  En cada ciclo de procesamiento, el sistema busca en el árbol de nodos todos los posibles objetivos pertenecientes a los grupos configurados. Para cada objetivo, comprueba si está dentro del rango de distancia y del ángulo de visión respecto a la entidad. Si varios objetivos cumplen los requisitos, se selecciona el más cercano como objetivo actual.

  La lógica de detección puede ampliarse fácilmente para incluir comprobaciones adicionales, como *raycasts*[15] para evitar que la visión atraviese obstáculos, aunque esta funcionalidad se ha dejado como mejora futura.

4.14. Gestor de *sprites*

El gestor de *sprites* (*SpriteManager*) es el módulo encargado de controlar la apariencia visual de las entidades en el juego, gestionando tanto la textura como las animaciones y efectos asociados al *sprite* principal de cada entidad. Este sistema permite separar la lógica de visua-

[15]Los *raycasts* son una técnica de detección de colisiones que permite determinar si un rayo (línea) intersecta con algún objeto en el entorno, como si fuera un láser.

lización del resto de funcionalidades, facilitando la personalización y el mantenimiento del aspecto gráfico de los personajes.

Su principal responsabilidad es coordinar el nodo *SpriteAnimator*, que se encarga de mostrar y animar el *sprite* en pantalla. Además, gestiona la selección de la textura del *sprite*, permitiendo elegir entre varias opciones configuradas desde el editor, e incluso seleccionar una textura aleatoria si así se desea.

El gestor se integra con el sistema de *signals* de la entidad, de modo que puede reaccionar automáticamente a eventos como la muerte o la recepción de daño. Por ejemplo, al recibir daño, el gestor utiliza una animación de *shader* para hacer que el *sprite* parpadee o cambie de color temporalmente, proporcionando una retroalimentación visual inmediata al jugador. Al morir la entidad, el *sprite* se oculta para mostrar la animación de muerte correspondiente.

Entre sus métodos principales se incluyen la asignación de la textura del *sprite*, el cambio de color mediante el sistema de colores del juego (*GameColor*), la gestión de la visibilidad y la selección de la textura por índice. Además, el gestor está diseñado para ser fácilmente ampliable, permitiendo añadir nuevas funcionalidades visuales o integrar efectos adicionales según las necesidades del proyecto.

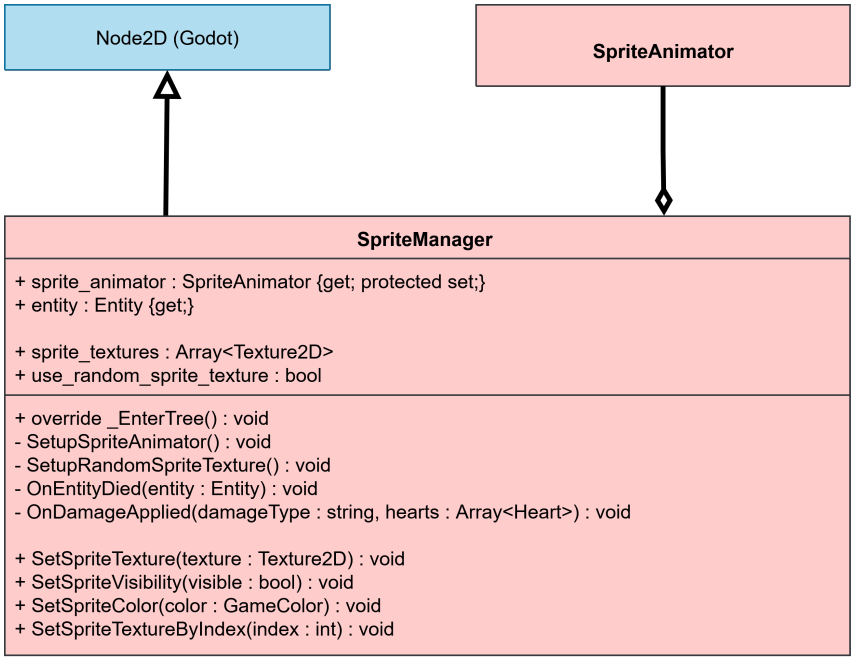


Figura 84: Representación UML de la clase *SpriteManager*.

4.14.1. Animador de *sprites*

El animador de *sprites* (*SpriteAnimator*) es el componente encargado de gestionar las animaciones visuales de cualquier *sprite*, por ejemplo, el de las entidades en su *SpriteManager*, aunque se utiliza en otros contextos.

Hereda de una clase genérica llamada *CanvasAnimator*, diseñada para animar cualquier nodo de tipo *CanvasItem* (como *Sprite2D*, *TextureRect*, *AnimatedSprite2D*, etc.) mediante interpolaciones hechas con corutinas (*tweens* en Godot) y efectos visuales.

El *SpriteAnimator* se especializa en animar *sprites* bidimensionales (*Sprite2D*), permitiendo aplicar efectos como cambios de color, materiales temporales y animaciones avanzadas de parámetros de *shader*. Para ello, utiliza el sistema de *Tween* de Godot, que permite interpolar propiedades del nodo de forma suave y configurable.

Para que un *CanvasAnimator* de cualquier tipo funcione, es necesario que el nodo objetivo sea hijo del animador.

Entre sus métodos principales destacan:

- ***AnimateColorFlash***: Permite animar el color del *sprite* (*modulate*) hacia un color objetivo y luego volver al color original, creando efectos de parpadeo o destello, e incluso se puede utilizar para alternar la transparencia.

- ***AnimateMaterialFlash***: Aplica temporalmente un material al *sprite* y lo restaura después, útil para efectos visuales especiales.

- ***AnimateShaderParameterFlash***: Permite animar múltiples parámetros de un *shader* de forma sincronizada, logrando efectos dinámicos como resplandores y pulsaciones con cambios de color en tiempo real. Este último, hace uso de la clase *ShaderMaterialAnimation* que se explica en la sección 4.14.2.

El sistema está diseñado para ser flexible y reutilizable, permitiendo emitir *signals* al finalizar cada animación para que otros módulos puedan reaccionar si es necesario. Además, el uso de una clase base genérica facilita la extensión a otros tipos de nodos gráficos en el futuro.

Esta arquitectura modular permite separar la lógica de animación visual del resto de la entidad, facilitando la personalización y el mantenimiento de los efectos gráficos en el juego.

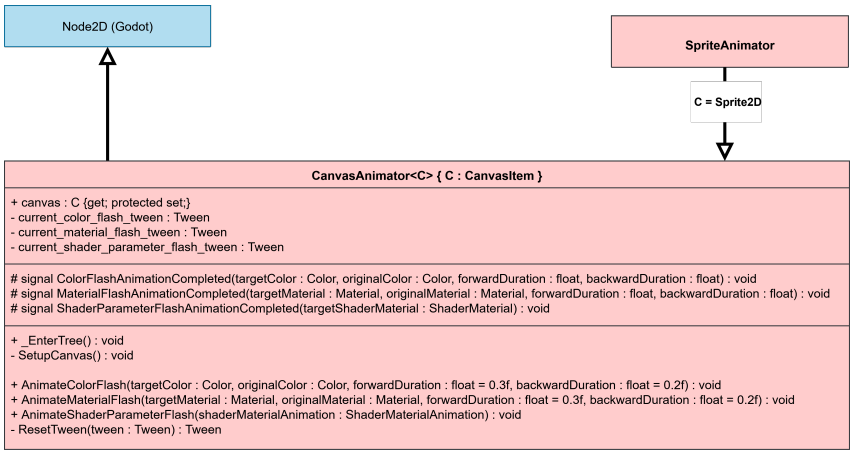


Figura 85: Representación UML de la clase *SpriteAnimator* y su clase base *CanvasAnimator*.

4.14.2. Animaciones para *shaders*

Para lograr efectos visuales avanzados y dinámicos en los *sprites* y elementos de la interfaz, el proyecto utiliza animaciones sobre materiales de *shader*, especialmente se ha implementado con el *shader* de resplandor (*Glow*) descrito en la sección 4.5.2. Estas animaciones permiten modificar en tiempo real los parámetros del *shader*, creando transiciones suaves de color, intensidad y otros atributos visuales.

El sistema está basado en la clase abstracta *ShaderMaterialAnimation*, que define la estructura y el flujo de las animaciones sobre materiales de tipo *ShaderMaterial*. Esta clase gestiona la clonación del material original para evitar modificarlo directamente, aplica los valores iniciales de los parámetros y configura una secuencia de interpolaciones (*tweens*) para animar los parámetros del *shader* tanto en la fase de avance (*forward*) como de retorno (*backward*). Al finalizar la animación, se puede restaurar automáticamente el material original del nodo animado.

Cada parámetro animable se representa mediante la clase genérica *ShaderMaterialAnimationParameter*, que almacena el nombre del parámetro, los valores inicial y final, la duración de la animación en cada sentido y permite configurar si se realiza la animación en uno o ambos sentidos.

La implementación concreta para el shader de resplandor se realiza en la clase *GlowShaderMaterialAnimation*, que permite animar los cuatro parámetros principales del efecto glow: color (*glow_color*), intensidad (*glow_power*), desplazamiento (*glow_shift*) y radio (*glow_radius*).

Cada uno de estos parámetros puede ser interpolado de forma independiente, permitiendo crear efectos como destellos, pulsaciones o cambios de color.

Además, el sistema es fácilmente extensible para animar otros tipos de shaders o parámetros según las necesidades del proyecto.

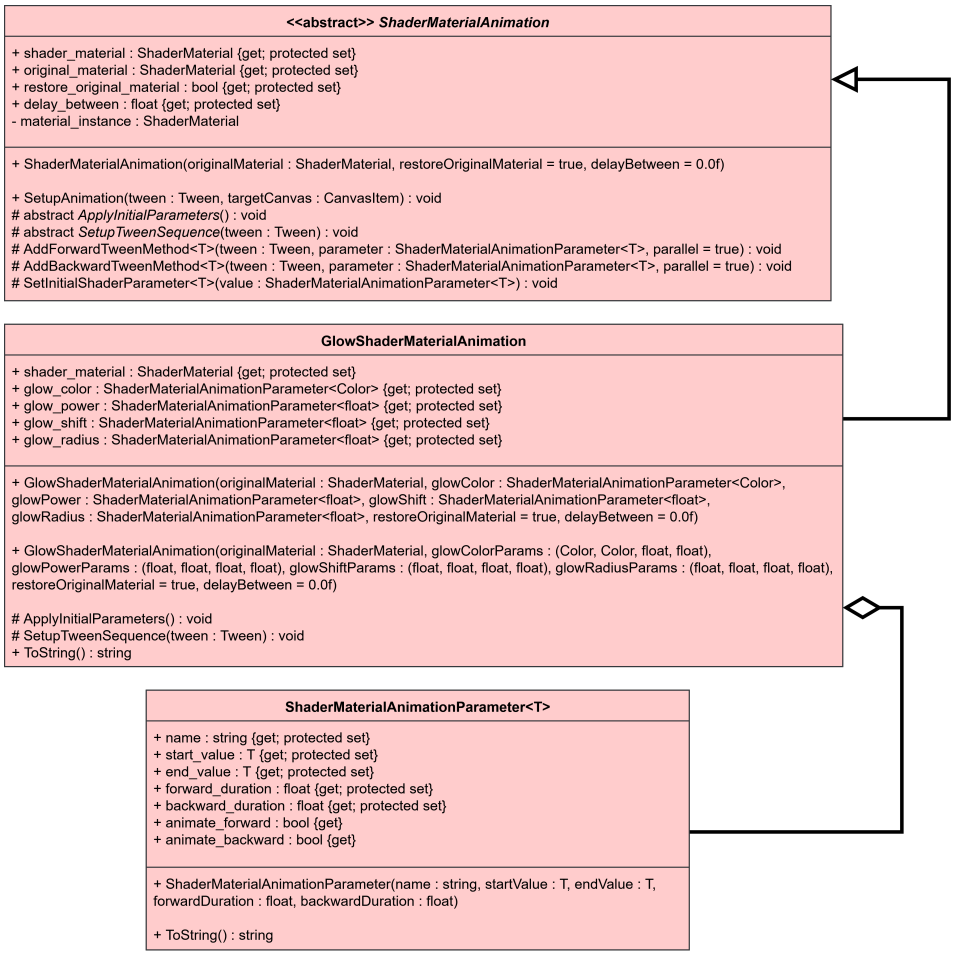


Figura 86: Representación UML de la clase *ShaderMaterialAnimation*.

4.15. Propulsores

El sistema de propulsores (*Thrusters*) es el encargado de gestionar los efectos visuales de partículas que simulan el impulso y la rotación de las entidades. Este módulo utiliza nodos de tipo *GpuParticles2D* para generar las partículas de los propulsores, diferenciando entre propulsores de movimiento y de rotación.

La clase *Thrusters* se instancia como nodo hijo de la entidad y detecta automáticamente los nodos de partículas configurados como sus hijos en la escena. Permite ajustar desde el editor la

cantidad de partículas emitidas y la intensidad del efecto varía según la fuerza de movimiento o rotación aplicada por la entidad, por lo que si rota con más velocidad o se aplica *boost* en el desplazamiento se notará que hay más cantidad de partículas.

El gestor de propulsores está diseñado para ser opcional: no todas las entidades lo utilizan (por ejemplo, los asteroides carecen de propulsores), también se puede configurar en el editor la posición y número de propulsores por cada entidad, lo que permite adaptar el sistema a las necesidades de cada tipo de entidad sin añadir complejidad innecesaria.

Además, en el caso del jugador, el sistema está integrado con el gestor de sonido (*AudioManager*), de modo que al activar los propulsores también se reproduce el efecto sonoro correspondiente, mejorando la retroalimentación audiovisual, aunque es un sonido que suena muy bajo y sin auriculares puede ser difícil de escuchar.

Visualmente, las partículas de los propulsores son cuadradas si son del jugador y triangulares si son de enemigos. Y sus colores son naranja si provienen de los propulsores de movimiento y naranja más claro si son de los propulsores de rotación.

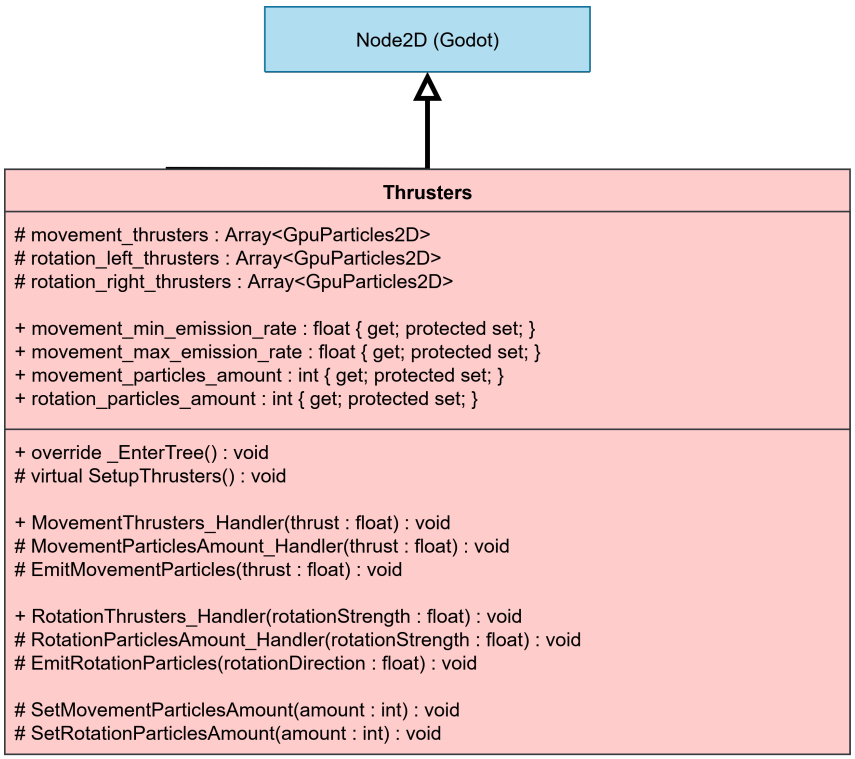


Figura 87: Representación UML de la clase *Thrusters*.

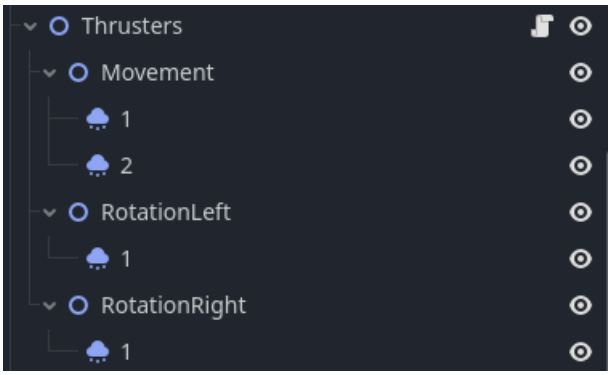


Figura 88: Ejemplo del árbol de nodos de los *Thrusters* del jugador.

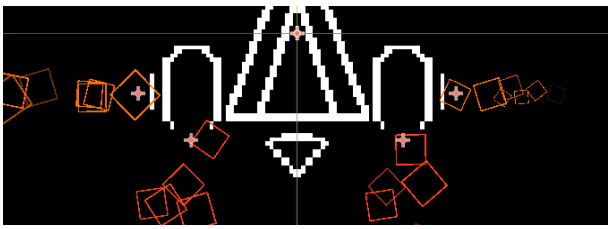


Figura 89: Ejemplo visual de los *Thrusters* del jugador desde el editor.

4.16. Tipos de entidades

Tras entender todos los sistemas y módulos que componen una entidad, es importante ver cómo se aplican estos a los distintos tipos de entidades que existen en el juego. Cada tipo de entidad hereda de la clase base *Entity* y configura sus módulos según sus necesidades específicas, en su mayoría desde el editor o utilizando las variantes heredadas de los módulos explicados anteriormente.

Toda entidad tiene una *PackedScene* de su *DeadEntity* asociada (ya sea por código o desde el editor) y también se añaden a los diferentes *GameGroups* (explicados en la sección 4.2) según su tipo (pueden estar en varios grupos a la vez).

Los aspectos comunes que tienen todas las entidades son: el área de detección de daño (*DamageDetectionArea*) y la *PhysicsCollisionShape* para el *CharacterBody2D* que es la entidad en sí misma (ambos explicados en la sección 4.10.1 y 4.7 respectivamente). Y también el gestor de *sprites* (*SpriteManager*), que controla la apariencia visual, incluyendo animaciones y efectos al recibir daño o su visibilidad.

4.16.1. Jugador

El jugador es la entidad controlada por el usuario, representada por la clase *Player* que hereda de *Entity*. La cual maneja aspectos extras como ciertos aspectos con la mirilla (*Scope*, explicada en la sección 4.18) y la visibilidad de la interfaz de salud seguidora (explicada en la sección 4.19.5). El jugador utiliza los siguientes módulos:

- **Comportamiento:** *PlayerBehaviorManager*, que incluye los módulos de movimiento y ataque específicos para el jugador: *PlayerMovementBehavior* y *PlayerAttackBehavior* respectivamente.

- **Gestor de salud:** *PlayerHealthManager*, que maneja la salud del jugador y emite *signals* para actualizar la interfaz del HUD (*HealthUI*).

- **Gestor de proyectiles:** *PlayerProjectileManager* para manejar la interfaz del HUD (*ProjectileUI*) y está configurado con la estrategia de disparo *StandardFiringStrategy*.

- **Propulsores:** *Thrusters*, que gestionan los efectos visuales de los propulsores de movimiento y rotación. Teniendo un total de un par de cada tipo.

- **Gestor de sonido:** *AudioManager*, que reproduce los efectos sonoros de: disparo, perder y recuperar corazón, y propulsores.

- **Área de detección de triggers:** para detectar cuando el jugador entra en contacto con *triggers* del escenario, explicado en la sección 4.22.

- **Área de detección para la cámara:** para que la cámara interactúe correctamente con el jugador, explicado en la sección 4.17.

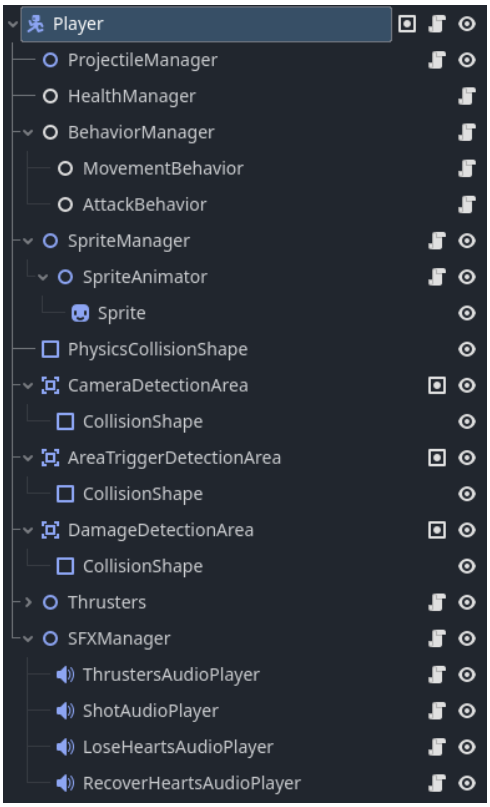


Figura 90: Árbol de nodos del *Player*.

4.16.2. Enemigos

Los enemigos utilizan la clase abstracta *Enemy*, que hereda de *Entity* y sirve como base para los distintos tipos de enemigos en el juego. Cada enemigo configura sus módulos según su comportamiento y características específicas.

Además de los aspectos comunes a todas las entidades, los enemigos tienen en común el mismo gestor de salud (*EnemyHealthManager*), el cual inicializa la salud del enemigo según su configuración desde el editor.

Los diferentes tipos de enemigos implementados son:

- **Asteroides:**

  Los asteroides son entidades que actúan como obstáculos en el juego. Ellos de por sí no pueden desplazarse, solo rotan (aunque en la sección 4.22.1 se explica que pueden desplazarse mediante el *RotationTrigger*). Utilizan la clase *Asteroid* que hereda de *Enemy*, la cual maneja un sistema de aleatoriedad para seleccionar entre los tipos (cuatro) de asteroides disponibles, lo cual afecta a su *sprite*. Utilizan los siguientes módulos:

- **Comportamiento:** *AsteroidBehaviorManager*, que incluye los módulos de movimiento y ataque específicos para el asteroide: *AsteroidMovementBehavior* y *NoAttackBehavior* respectivamente.
- **Gestor de proyectiles:** *EnemyProjectileManager* pero implementando la estrategia de disparo *NoFiringStrategy*, pues no dispara.
- **Gestor de sonido:** *AudioManager*, que reproduce los efectos sonoros de perder y recuperar corazón.

Figura 91: Árbol de nodos del *Asteroid*.

- **Enemigo básico**

  El enemigo básico es la entidad enemiga más simple, que persigue al jugador y dispara proyectiles directamente hacia él, además es de tamaño pequeño y tiene dos variantes. Implementa la clase *BasicEnemy* que hereda de *Enemy*. Utiliza los siguientes módulos:

  - **Comportamiento:** *BasicEnemyBehaviorManager*, que incluye los módulos de movimiento, ataque y visión: *OrbitalMovementBehavior*, *DirectAttackBehavior* y *RadarVisionBehavior* respectivamente.
  - **Gestor de proyectiles:** *EnemyProjectileManager* pero implementando la estrategia de disparo *StandardFiringStrategy*, pues dispara como el jugador.

- **Propulsores:** ambas variantes tienen un par de propulsores de rotación, pero la variante 1 tiene un propulsor de movimiento y la variante 2 tiene un par.
- **Gestor de sonido:** *AudioManager*, que reproduce los efectos sonoros de perder y recuperar corazón, y de disparo.

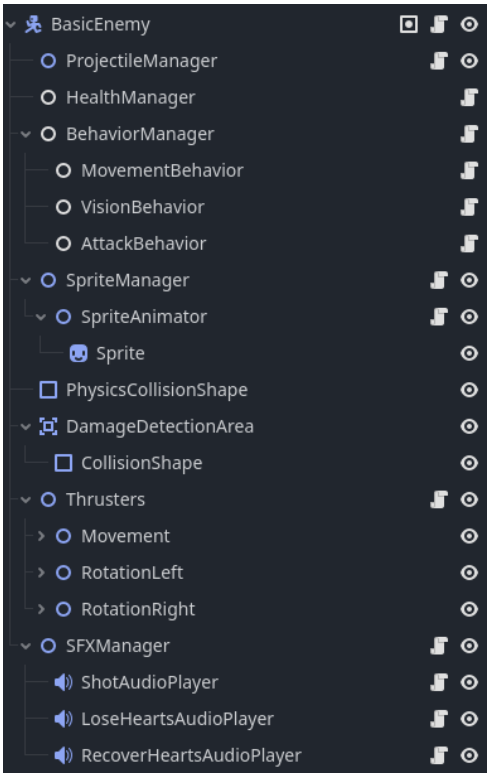


Figura 92: Árbol de nodos del *BasicEnemy*.

- **Enemigo con cañón dual**

  El enemigo con cañón dual persigue al jugador y dispara proyectiles desde dos cañones simultáneamente. Es de tamaño más grande que el enemigo básico. Además es más lento y suele orbitar a mayor distancia. Implementa la clase *DualCannonEnemy* que hereda de *Enemy*. Tiene en común los mismos módulos que el enemigo básico, pero con las siguientes excepciones:

  - **Gestor de proyectiles:** *EnemyProjectileManager* pero implementando la estrategia de disparo *MultipleFiringStrategy*, pues dispara dos proyectiles simultáneamente.

- **Propulsores:** tiene un par de propulsores de rotación y un par de propulsores de movimiento.

- **Enemigo con cañón cruzado alternado**

  El enemigo con cañón cruzado alternado persigue al jugador y dispara proyectiles desde dos cuartetos de cañones que forman cruces, alternando entre cada cuarteto en cada disparo. No es capaz de rotar, por lo que siempre apunta en las mismas direcciones. Es lento y de tamaño grande. Implementa la clase *CrossfireAlternatorEnemy* que hereda de *Enemy*. Tiene en común los mismos módulos que el enemigo básico, pero con las siguientes excepciones:

  - **Gestor de proyectiles:** *EnemyProjectileManager* pero implementando la estrategia de disparo *MultipleAlternatorFiringStrategy*, pues dispara cuatro proyectiles simultáneamente, alternando entre dos grupos de cañones.
  - **Propulsores:** tiene dos pares de propulsores de cada tipo.

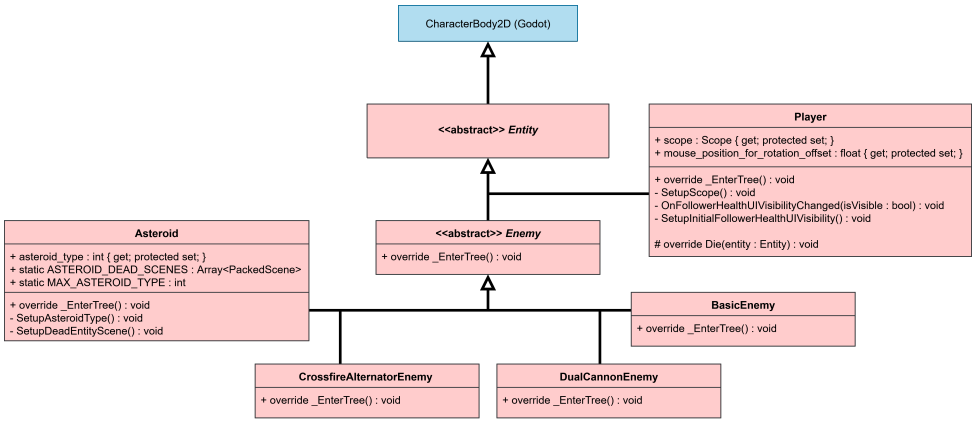


Figura 93: Diagrama de clase de *Entity* y sus derivados.

Para ver más detalles sobre los enemigos se puede ver en el GDD (Apéndice B, sección 9.5).

4.17. Cámara

La cámara del juego está gestionada por la clase *LevelCamera*, que hereda del nodo *Camera2D* y se encarga de seguir al jugador o a cualquier objetivo de forma suave y dinámica.

El sistema implementa dos áreas de detección (*Area2D*): la primera detecta cuando el objetivo la abandona, lo que produce que comience a seguirlo; y la segunda detecta cuando el objetivo entra en ella, lo que detiene el seguimiento. Estas áreas están para que la cámara no esté siempre siguiendo al objetivo, sino que solo lo haga cuando este se aleje demasiado, proporcionando una experiencia visual más cómoda.

Permite alternar entre un modo de cámara estática y un modo de seguimiento, adaptando el comportamiento según se desee.

La transición del movimiento de la cámara se realiza mediante interpolación (*lerp*), lo que garantiza desplazamientos suaves y naturales. La velocidad de seguimiento y el umbral mínimo de movimiento son configurables desde el editor, permitiendo ajustar la respuesta de la cámara a las necesidades de cada nivel o situación.

El sistema permite cambiar el objetivo a seguir en tiempo real y ajustar el desplazamiento (*offset*) respecto a la entidad, facilitando la personalización de la experiencia visual. Además, incluye métodos para convertir posiciones de pantalla a coordenadas del mundo, útiles para la interacción con la interfaz y otros sistemas.

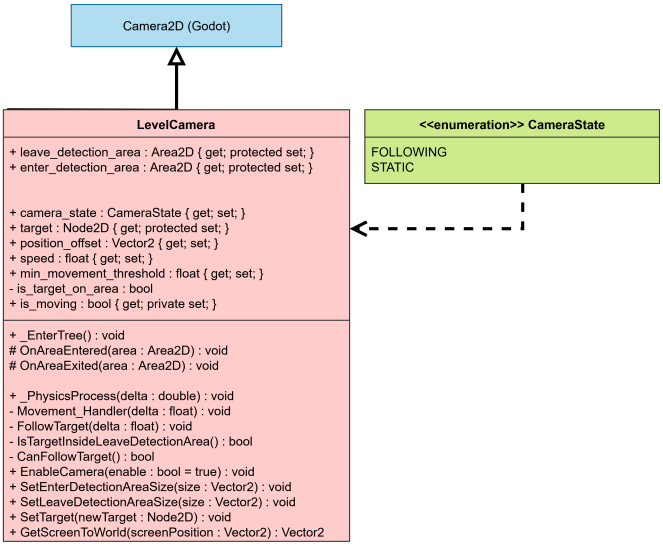


Figura 94: Representación UML de la clase *LevelCamera*.

4.18. Mirilla (*Scope*)

La mirilla (*Scope*) es un elemento visual que indica el punto de referencia hacia el que apunta el jugador y sirve como objetivo para la rotación de la nave. Está gestionada por la

clase *Scope*, que hereda de *Node2D* y se integra con la cámara principal del nivel para asegurar que la mirilla permanezca siempre dentro de los límites visibles de la pantalla.

El sistema permite controlar la posición de la mirilla tanto con el ratón como con el joystick derecho del mando, adaptándose automáticamente al tipo de entrada que esté utilizando el jugador. Cuando se usa el ratón, la mirilla sigue la posición del cursor en el mundo del juego; cuando se utiliza el mando, la mirilla se desplaza en la dirección indicada por el stick, con una sensibilidad configurable desde los ajustes.

La posición de la mirilla se actualiza suavemente mediante interpolación, garantizando una transición fluida y natural. Además, se aplica un margen configurable para evitar que la mirilla se sitúe demasiado cerca del borde de la pantalla, mejorando la experiencia visual y evitando movimientos bruscos.

La clase *Scope* también permite controlar su visibilidad, lo que resulta útil en situaciones como la muerte del jugador o la transición entre menús. Su integración con el sistema de entrada y la cámara asegura que la mirilla responda correctamente a las acciones del usuario y se mantenga siempre en una posición válida dentro del área jugable.

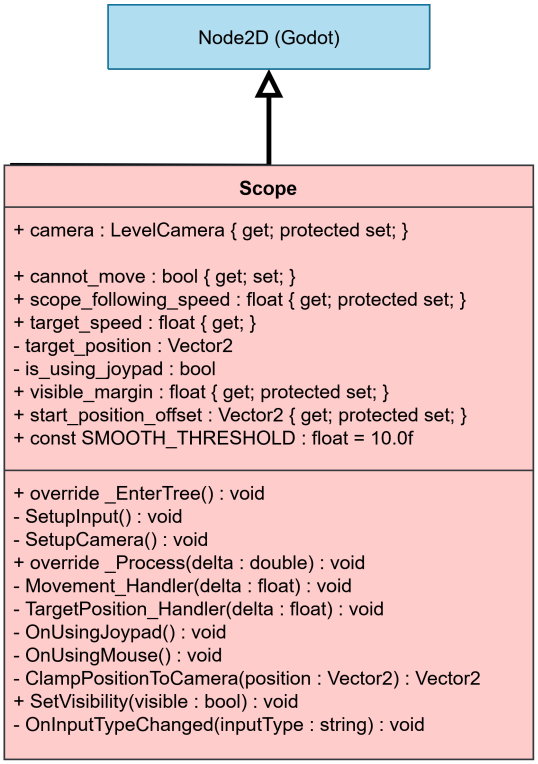


Figura 95: Representación UML de la clase *Scope*.

4.19. Interfaces de usuario

La interfaz de usuario (UI) es el conjunto de elementos visuales que permiten al jugador interactuar con el juego, recibir información y navegar por los distintos menús. En este proyecto, la UI se ha diseñado para ser clara, intuitiva y coherente con la estética general del juego, utilizando los sistemas nativos de Godot para su implementación: los nodos *Control* y sus derivados.

La estructura de la interfaz se organiza en varios módulos y componentes reutilizables, lo que facilita su mantenimiento y ampliación. Los menús principales, el HUD y los componentes separados se gestionan como escenas independientes, permitiendo su instanciación y eliminación según las necesidades de cada momento del juego.

Algunos aspectos importantes durante el desarrollo de las interfaces han sido el manejo de los diferentes tipos de entrada (ratón y mando) incluyendo el sistema de *focus*[16] con el mando, el cambio en el idioma de los textos, y la adaptación a distintas resoluciones de pantalla mediante el uso de contenedores y anclajes adecuados.

A continuación se describen los principales componentes y menús que forman la interfaz de usuario, así como las técnicas y decisiones de diseño empleadas para su desarrollo.

4.19.1. Componentes

Godot ofrece una amplia variedad de nodos que derivan de *Control*, cada uno con funcionalidades específicas para construir interfaces de usuario. A pesar de esto, es necesario ajustar esos componentes para que se adapten a las necesidades del juego y mantengan una coherencia visual, ya sea mediante código o configuraciones en el editor.

Se han creado los componentes en escenas independientes para facilitar su reutilización. A continuación se describen los principales componentes utilizados en la interfaz de usuario:

- **Etiquetas de texto (*Labels*)**

  Las etiquetas de texto se gestionan mediante la clase *BasicLabel*, que hereda de la clase nativa *Label* y añade soporte para traducciones automáticas. Cada etiqueta puede asociarse a un identificador de traducción (*translation_id*), permitiendo que su texto se

[16] El *focus* es el concepto de que un elemento está actualmente seleccionado y que recibirá la entrada del usuario. Cuando se utiliza mando, el *focus* se transfiere entre los elementos de la interfaz según la navegación del usuario.

actualice dinámicamente según el idioma seleccionado en el juego. Al cambiar el idioma, la etiqueta se actualiza automáticamente gracias a la suscripción de la *signal* de cambio de idioma del módulo *GameLanguage*, ya explicado en la sección 4.3.2.

Esto facilita la internacionalización y garantiza que todos los textos de la interfaz se adapten correctamente sin necesidad de modificar manualmente cada etiqueta. Además, la clase permite configurar si una etiqueta necesita traducción o no, lo que aporta flexibilidad para textos que no cambian entre idiomas.

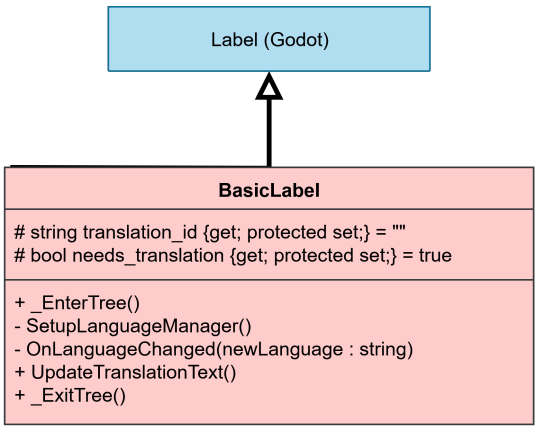


Figura 96: Representación UML de la clase *BasicLabel*.

- **Botones**

  Los botones de la interfaz se gestionan mediante la clase *BasicButton*, que hereda del nodo nativo *Button* y sirve como base para otros tipos de botones más especializados. En Godot, los botones pueden tener diferentes estados visuales (normal, presionado, deshabilitado, etc.) y pueden incluir tanto texto como iconos.

  La clase implementada agrega funcionalidades específicas para el juego. Entre ellas destaca la integración con el gestor global de efectos de sonido (*GlobalSFXManager*, explicado en la sección 4.6.2), de modo que al pulsar, seleccionar o pasar el ratón sobre el botón se reproducen sonidos que refuerzan la retroalimentación audiovisual.

  Existen dos extensiones principales de *BasicButton*:

  - **BasicTextButton**: Añade soporte para traducción automática del texto del botón, asociando cada instancia a un identificador de traducción (*translation_id*). Al

cambiar el idioma del juego, el texto del botón se actualiza automáticamente, garantizando la internacionalización de la interfaz. Esta clase se suscribe al evento de cambio de idioma del módulo *GameLanguage*.

- **BasicCheckButton**: Extiende *BasicTextButton* para implementar botones de tipo alternador (*check button*), que pueden estar activados o desactivados. Además de la traducción automática, esta clase gestiona el color del texto según el estado del botón (marcado o no), "corrigiendo" el comportamiento nativo de Godot para mantener la coherencia visual en todos los estados del botón de la manera que se deseaba.

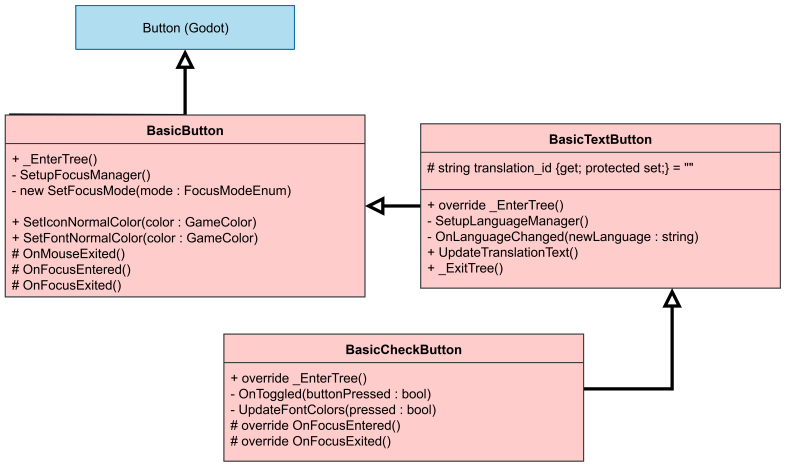


Figura 97: Representación UML de la clase *BasicButton* y sus derivados.

- **Deslizadores (*Sliders*)**

  Los deslizadores se gestionan mediante la clase *BasicVerticalSlider*, que hereda del nodo nativo *VSlider*. Este componente incluye una etiqueta de título y una etiqueta para mostrar el valor numérico actual, ambas gestionadas como nodos hijos y actualizadas automáticamente mediante la clase creada.

  El deslizador soporta traducción automática del título mediante un identificador de traducción (*title_translation_id*), permitiendo que el texto se adapte al idioma seleccionado en el juego. Además, está también integrado con el gestor global de efectos de sonido (*GlobalSFXManager*).

  El componente gestiona el estado de edición y el modo de foco, asegurando que solo sea

interactivo cuando corresponde y liberando el foco si el ratón sale del área del deslizador. El valor mostrado se actualiza en tiempo real y se formatea sin decimales para facilitar la lectura.

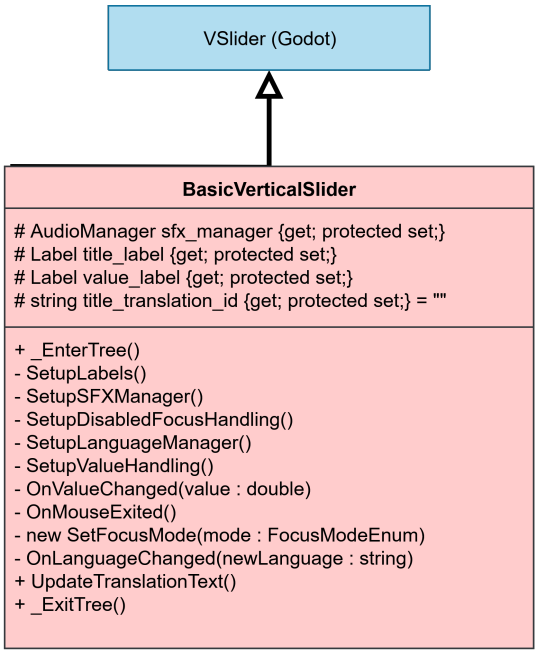


Figura 98: Representación UML de la clase *BasicVerticalSlider*.

- **Icono de *input***

  El componente *InputIcon* se utiliza para mostrar visualmente el tipo de entrada (teclado o mando) asociado a una acción en la interfaz, junto con su texto descriptivo. Hereda de *BoxContainer* y está compuesto por dos iconos (*TextureRect*) y una etiqueta de texto (*Label*), permitiendo mostrar tanto el icono principal como uno alternativo si es necesario.

  El sistema gestiona automáticamente el cambio de icono y texto según el tipo de entrada detectado en el juego, suscribiéndose a las *signals* de cambio de idioma y de tipo de entrada. Así, al cambiar entre teclado y mando, el icono se actualiza dinámicamente para reflejar el control activo, y el texto se traduce según el idioma seleccionado.

  El componente permite configurar desde el editor las texturas de los iconos para cada tipo de entrada, así como sus variantes alternativas (pues una misma acción puede tener más de una entrada con el mismo tipo de control). Esto facilita la personalización visual

y la adaptación a diferentes estilos de controles. Además, la clase gestiona la visibilidad del icono alternativo y asegura que el texto se mantenga sincronizado con el idioma y la acción representada.

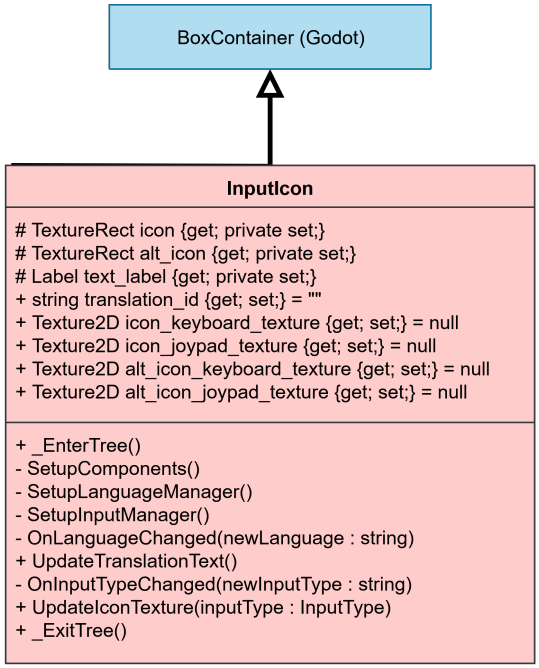


Figura 99: Representación UML de la clase *InputIcon*.

- **Selección**

  El componente de selección se gestiona mediante la clase genérica *BasicSelection*, que hereda de *BoxContainer* y permite crear selecciones de opciones de cualquier tipo (por ejemplo, idioma). Este sistema está formado por una etiqueta de título y una serie de botones organizados en filas, cada uno representando una opción disponible.

  La clase gestiona automáticamente la traducción del título mediante un identificador (*title_translation_id*), actualizándolo al cambiar el idioma del juego (el texto de los botones es manejado por cada uno porque son *BasicTextButton*).

  Los botones de opción se configuran dinámicamente según los valores de selección definidos, y su estado visual se actualiza para resaltar la opción seleccionada.

  Al pulsar un botón, el componente actualiza el índice de selección y emite una *signal* para notificar el cambio, permitiendo que otros sistemas reaccionen en consecuencia. Además, el sistema permite habilitar o deshabilitar todos los botones, gestionar el foco

y liberar el foco de manera centralizada, facilitando la navegación tanto con ratón como con mando.

La clase es abstracta y las clases heredadas deben implementar los valores de selección específicos, adaptando el componente a diferentes contextos dentro del juego. Se han creado dos extensiones principales: *LanguageSelection* (para seleccionar el idioma) y *WindowModeSelection* (para seleccionar el modo de ventana).

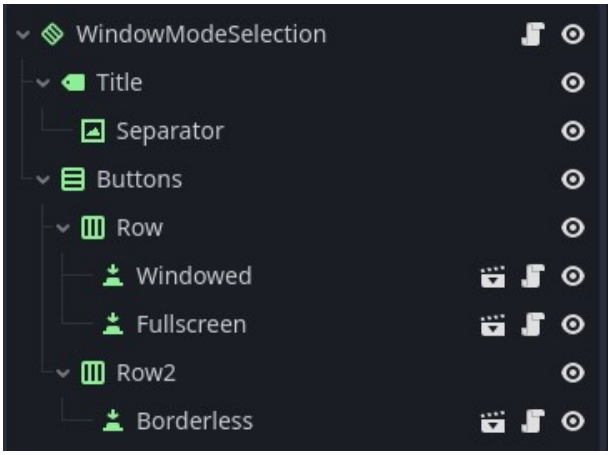


Figura 100: Ejemplo del árbol de nodos de una *BasicSelection*.

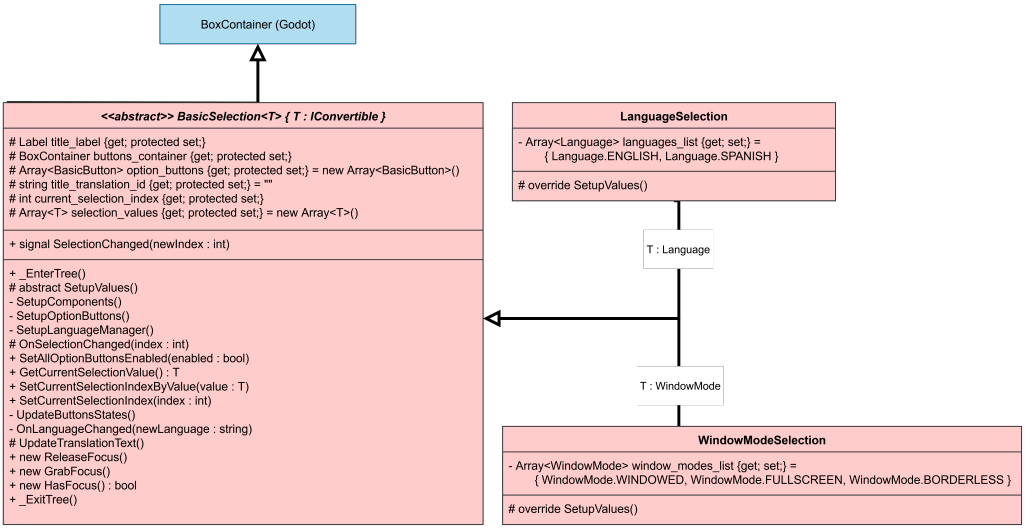


Figura 101: Representación UML de la clase *BasicSelection*.

- **Barra de desplazamiento (*Scroll Bar*)**

  La barra de desplazamiento vertical se gestiona mediante la clase *BasicVerticalScrollBar*,

que hereda del nodo nativo *VScrollBar* y añade funcionalidades específicas para la interfaz del juego. Este componente permite al usuario desplazarse por contenidos extensos, como los créditos, tanto con el ratón como con el mando.

El sistema gestiona el estado de edición, permitiendo habilitar o deshabilitar la interacción del usuario. También integra el gestor global de efectos de sonido (*GlobalSFXManager*).

La barra permite ajustar la velocidad de desplazamiento tanto para la rueda del ratón como para el mando. El componente también gestiona el aspecto visual, cambiando el estilo del *grabber*[17] (es un corazón con y sin relleno) según el estado de foco y entrada.

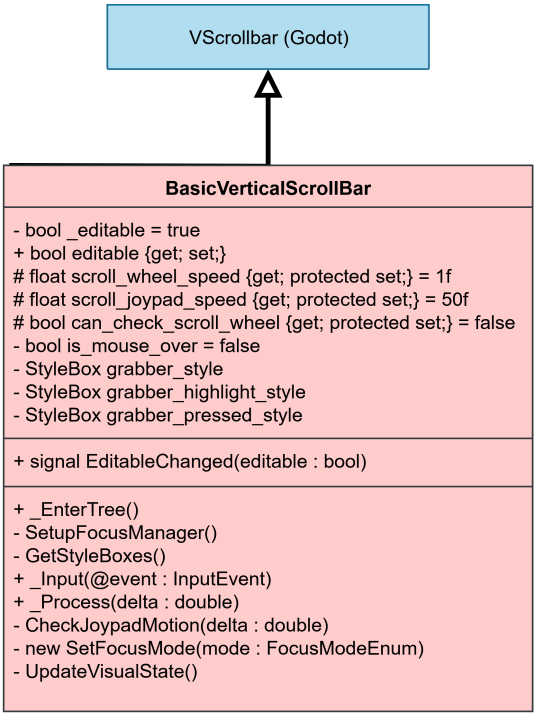


Figura 102: Representación UML de la clase *BasicVerticalScrollBar*.

Para ver más detalles sobre los componentes de interfaces, ver en el GDD (Apéndice B, sección 5.1 y 5.2).

[17] Un *grabber* es la parte de la barra de desplazamiento que el usuario puede arrastrar para mover el contenido.

4.19.2. Menús

Los menús del juego se gestionan mediante la clase abstracta *Menu*, que hereda de *Control* y sirve como base para todos los menús principales: menú principal, selección de nivel, pausa, ajustes, *Game Over*, nivel completado y créditos.

La clase base *Menu* sobrescribe el método nativo *_Notification()*, que en Godot se utiliza para recibir eventos del sistema, como cambios de visibilidad o foco. En concreto, se emplea para detectar cuándo el menú se muestra en pantalla (*NotificationVisibilityChanged*) y, si el jugador está utilizando un mando, tomar automáticamente el foco del primer control interactivo.

Además, la clase gestiona la suscripción y desconexión de *signals* relevantes, como el cambio de idioma (*LanguageChanged*) y de tipo de entrada (*InputTypeChanged*), asegurando que los textos y el foco se actualicen correctamente en tiempo real.

La arquitectura basada en métodos abstractos obliga a que cada menú concreto implemente su propia lógica de inicialización (*SetupUI*), traducción (*UpdateTranslationText*) y gestión de foco (*GrabInitialFocus*, *ReleaseAllFocus*), garantizando flexibilidad y coherencia en toda la interfaz del juego. Además, el sistema permite habilitar o deshabilitar todos los botones del menú de forma centralizada, útil para evitar acciones durante transiciones o animaciones.

Todos los menús son gestionados por la clase *GameUI*, que se encarga de instanciarlos, mostrarlos y ocultarlos según el estado del juego.

A continuación se describen las implementaciones específicas de cada menú:

- **Menú principal**

  El menú principal es la interfaz de entrada al juego y está implementado mediante la clase *MainMenu*. Este menú presenta al jugador los botones principales para iniciar la partida (ir a la selección de niveles), acceder a los ajustes, ver los créditos y salir del juego. Cada botón está conectado a una *signal* específica que permite al sistema central (*Game*) reaccionar y navegar hacia la sección correspondiente.

  Además, el menú principal muestra la versión actual del juego en la parte inferior, facilitando la identificación de la versión instalada (leído del atributo de *Game*).

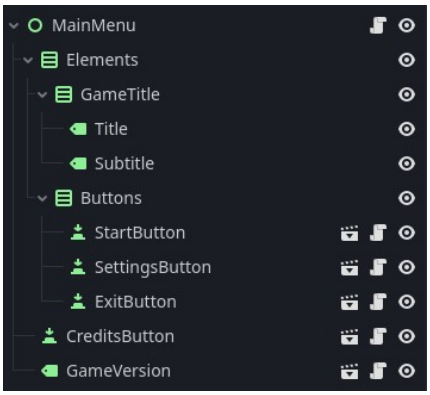


Figura 103: Árbol de nodos del *MainMenu*.

- **Menú de selección de nivel**

  El menú de selección de nivel, que es la clase implementada *LevelSelectionMenu*, permite al jugador elegir el nivel que desea jugar. La interfaz organiza los niveles en páginas, mostrando hasta tres niveles por página y permitiendo navegar entre ellas mediante botones de flechas. Cada nivel se representa con un botón personalizado (*SelectableLevel*) que muestra el icono, el nombre y el estado de desbloqueo y completado.

  Los niveles bloqueados aparecen con una textura especial de un candado y no pueden seleccionarse, mientras que los niveles desbloqueados muestran su icono específico y permiten iniciar la partida. El sistema ajusta el estado de cada nivel según el progreso del jugador (sección 4.21.3), gestionando la activación y desactivación de botones para evitar acciones no válidas.

  El componente *SelectableLevel* encapsula la lógica visual y funcional de cada botón de nivel, mostrando el icono correspondiente, el nombre traducido y el estado de completado. Si el nivel está bloqueado, el botón se desactiva y se ocultan los textos descriptivos. Todo esto tiene que ver con *GameLevels* y *LevelData*, explicados en la sección 4.3.7.

  En conjunto, este menú proporciona una experiencia clara y accesible para la selección de niveles integrando los datos de progreso del jugador.

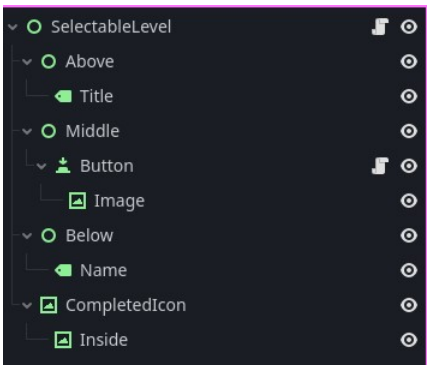


Figura 104: Árbol de nodos del *SelectableLevel*.

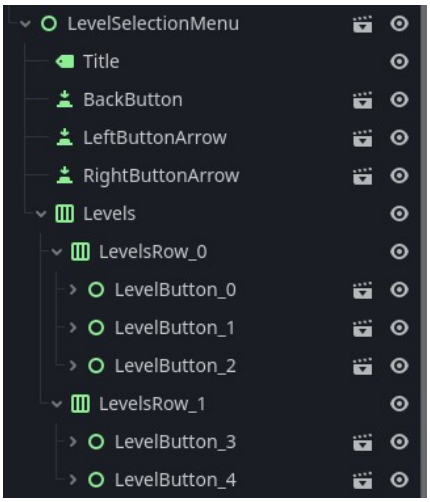


Figura 105: Árbol de nodos del *LevelSelection*.

- **Menú de pausa**

  El menú de pausa usa la implementación *PauseMenu*, permite al jugador detener temporalmente la partida y acceder a opciones como reanudar el juego, abrir el menú de ajustes o salir al menú principal. Este menú se activa cuando el juego entra en estado de pausa (*GameState.PAUSED*), haciendo *Game* visible este menú con tres botones principales: continuar, ajustes y salir.

  El menú de pausa está instanciado desde el comienzo de que se carga un nivel, pero permanece oculto hasta que el jugador decide pausar la partida. Se oculta nuevamente al reanudar el juego y se elimina al salir del nivel. (Todo esto gestionado por *Game* y *GameUI*).

El sistema de pausa en Godot se basa en el concepto de **pausar el árbol de nodos**, lo que significa que todos los nodos cuyo *ProcessMode* está configurado como *Pausable* dejan de procesar sus métodos (*_Process*, *_PhysicsProcess*, etc.) mientras el juego está pausado. Esto garantiza que la lógica del juego, el movimiento de entidades y otros sistemas se detengan completamente, permitiendo al jugador interactuar únicamente con la interfaz de pausa, la cual tiene su *ProcessMode* configurado como *When Paused* para funcionar solo cuando lo demás está en pausa.

Añadir como extra que en la jerarquía de nodos de *Game*:

- El nodo *Game* usa *ProcessMode.Always* para que siempre procese sus métodos y pueda gestionar el estado del juego aunque esté pausado.
- Todos los módulos de *Game* usan *ProcessMode.Inherit*, heredando el modo de *Game*. Es decir, siempre procesan, incluso en pausa. La única excepción es *Background*, *GlobalMusicManager* y *GlobalSFXManager*, que usan *ProcessMode.Pausable* para que sí se pausen.
- El nodo *Level*, hijo de *Gameplay*, usa *ProcessMode.Pausable* para que se pause junto con el resto del juego.

Los botones del menú emiten *signals* específicas al sistema central (*Game*), permitiendo reanudar la partida, abrir el menú de ajustes o salir del nivel según la acción seleccionada.

Este sistema de pausa, junto con el manejo del *ProcessMode*, asegura que la experiencia de usuario sea consistente y que el juego se detenga correctamente, evitando problemas de procesamiento o interacción no deseada durante la pausa.

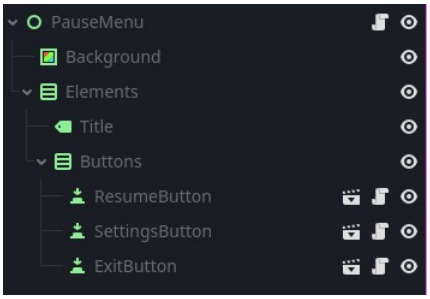


Figura 106: Árbol de nodos del *PauseMenu*.

- **Menú de ajustes**

  El menú de ajustes, implementado en la clase *SettingsMenu*, permite al jugador modificar las principales opciones de configuración del juego de forma visual e intuitiva. La interfaz está compuesta por controles agrupados en dos columnas: a la izquierda se encuentran las opciones de idioma (*LanguageSelection*), modo de ventana (*WindowModeSelection*) y la opción para mostrar la interfaz de salud seguidora (es un *BasicCheckButton*); a la derecha, los deslizadores para ajustar el volumen de la música y los efectos de sonido (*BasicVerticalSlider*), así como la sensibilidad de la mirilla al usar mando.

  Cada control está vinculado directamente con el sistema de datos guardados y aplicación de los ajustes en el juego (*GameData* y *GameSettings*, sección 4.21 y 4.21.2 respectivamente), de modo que los cambios realizados por el jugador se aplican y guardan automáticamente.

  El botón de volver permite salir del menú de ajustes y regresar al menú anterior, emitiendo una *signal* para que el sistema central (*Game*) gestione la transición (pues puede ser desde el menú principal o desde el menú de pausa).

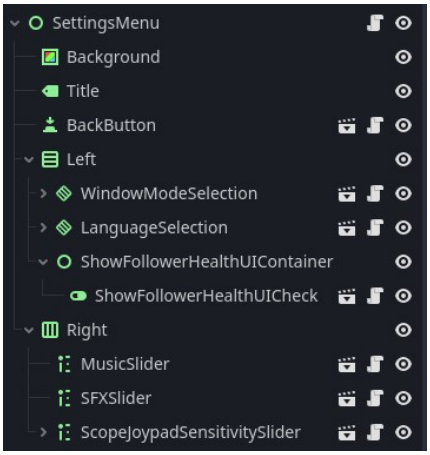


Figura 107: Árbol de nodos del *SettingsMenu*.

- **Menú de *Game Over***

  El menú de *Game Over* implementa la clase *GameOverMenu*, aparece cuando el jugador pierde la partida. Su interfaz muestra el título correspondiente y dos botones principales: reintentar el nivel y salir al menú principal. Cada botón emite una *signal* específica para que el sistema central (*Game*) gestione la acción seleccionada.

El menú se instancia al iniciar el nivel y permanece oculto hasta que el jugador muere. Al mostrarse permite ser utilizado porque el *GameState* está en *GAME_OVER*. Se elimina al salir del nivel.

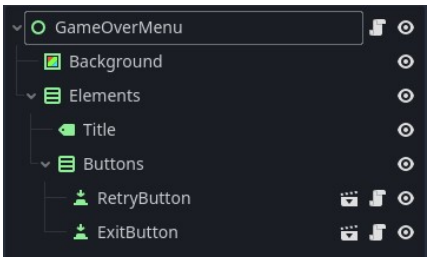


Figura 108: Árbol de nodos del *GameOver*.

- **Menú de nivel completado**

  El menú de nivel completado es la clase implementada *LevelCompletedMenu*, aparece cuando el jugador termina un nivel porque ha entrado en el *portal de fin de nivel*. Su interfaz muestra el título correspondiente y dos botones principales: reiniciar el nivel y salir al menú principal. Cada botón emite una *signal* específica para que el sistema central (*Game*) gestione la acción seleccionada, la cual son las mismas que en el menú de *Game Over* aunque con diferente nombre y contexto.

  El menú se instancia al iniciar el nivel y funciona igual que el de pausa y el de *Game Over*, pero en este caso el menú se muestra cuando el *GameState* está en *LEVEL_COMPLETED*. Se elimina de la misma forma.

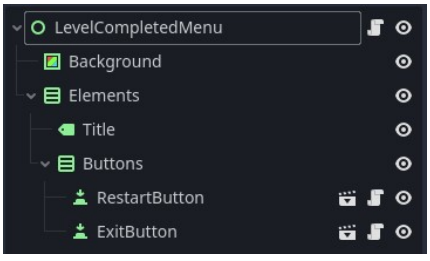


Figura 109: Árbol de nodos del *LevelCompletedMenu*.

- **Menú de créditos**

  El menú de créditos (*CreditsMenu*), muestra la información sobre los autores y colaboradores del juego. La interfaz está compuesta por un contenedor vertical que agrupa las

líneas de créditos (*CreditLine*), cada una con su título y los nombres correspondientes. El menú incluye una barra de desplazamiento vertical (*BasicVerticalScrollBar*) que permite navegar por el contenido verticalmente.

El botón de volver permite regresar al menú principal, emitiendo una *signal* para que el sistema central (*Game*) gestione la transición.

El componente *CreditLine* encapsula la lógica visual de cada línea de créditos, mostrando el título (traducido si es necesario) y los nombres asociados.

Este menú proporciona una presentación clara y accesible de los créditos, integrando la traducción de textos y la navegación adaptada a diferentes tipos de entrada.

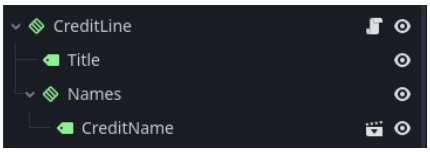


Figura 110: Árbol de nodos del *CreditLine*.

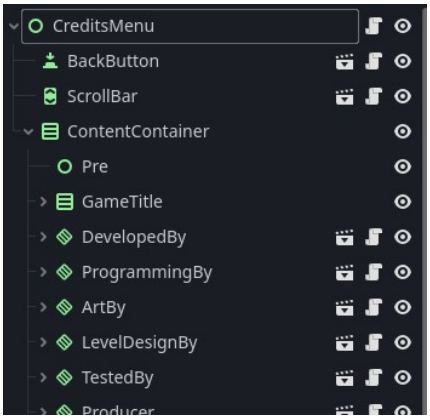


Figura 111: Árbol de nodos del *CreditsMenu*.

Para ver más detalles sobre los menús, ver en el GDD (Apéndice B, sección 5.3).

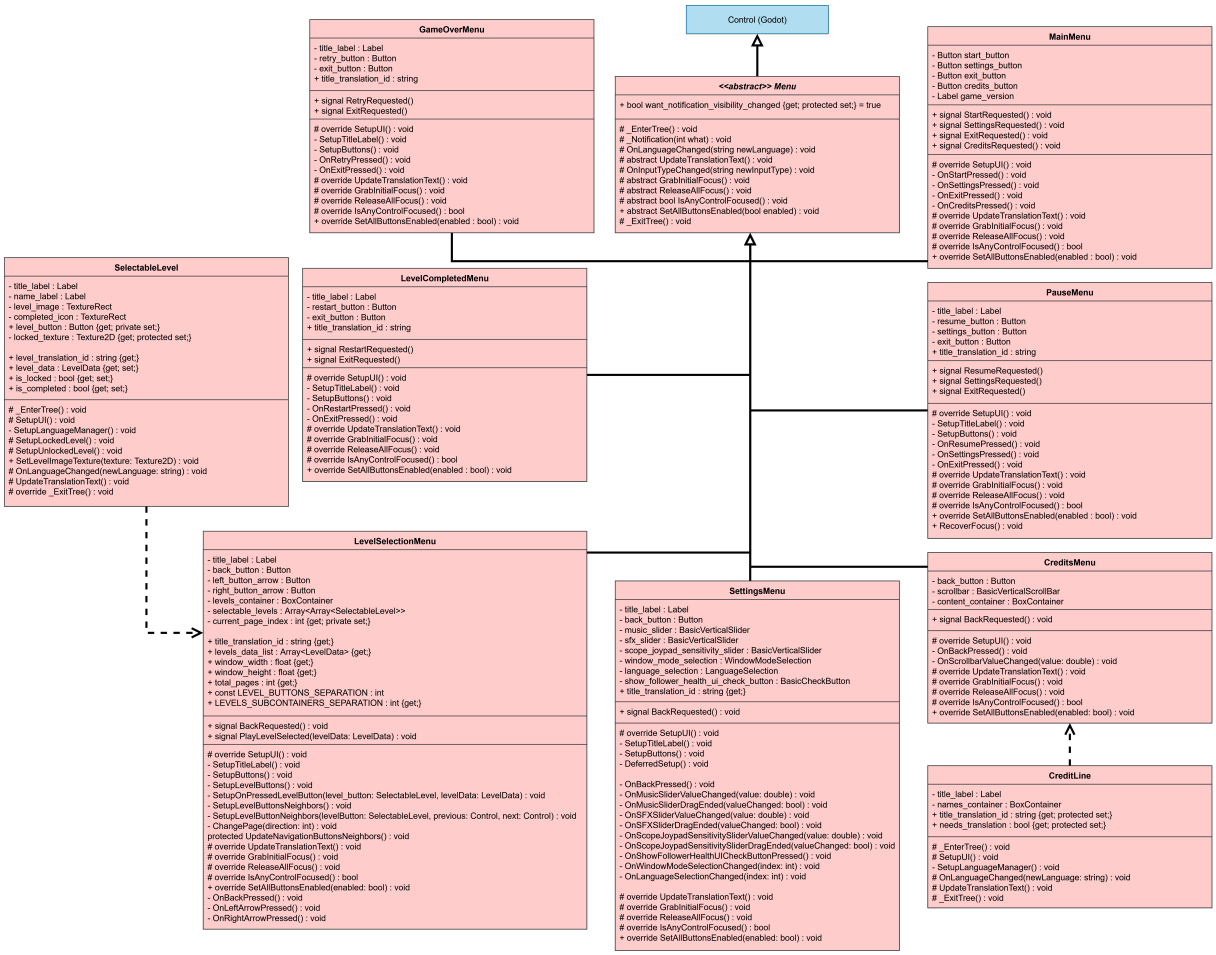


Figura 112: Diagrama de clase de *Menu* y sus derivados.

4.19.3. Transición de carga

La transición de carga (*LoadingTransition*) es una escena especial utilizada para animar el paso entre menús y niveles, proporcionando una experiencia visual fluida durante los tiempos de carga. Su diseño consiste en dos contenedores (*Above* y *Below*) que agrupan varias franjas horizontales (*TextureRect*), las cuales se animan de izquierda a derecha para cubrir y descubrir la pantalla. El sistema es adaptable a diferentes cantidades de franjas, permitiendo ajustar el número y tamaño desde el editor.

El sistema utiliza interpolaciones (*Tween*) para animar el tamaño y la posición de cada franja, creando un efecto de cierre y apertura progresiva. La transición de entrada (*TransitionIn*) muestra las franjas desde el borde izquierdo hasta cubrir toda la pantalla, mientras que la

transición de salida (*TransitionOut*) las desplaza hacia la derecha reduciendo su ancho hasta desaparecer.

La duración y el retardo entre franjas son configurables, permitiendo ajustar la velocidad y el ritmo de la animación. Al finalizar cada transición, se emite una *signal* para notificar al sistema central (*Game*) que puede continuar con la siguiente acción (cargar menú, nivel, etc.).

Este componente mejora la percepción de calidad y continuidad en el juego, evitando cortes bruscos y aportando identidad visual a las transiciones entre estados.

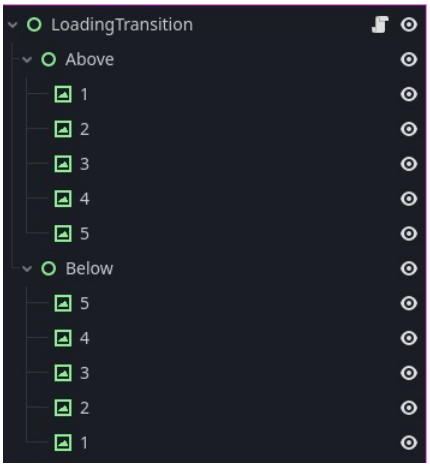


Figura 113: Árbol de nodos del *LoadingTransition*.

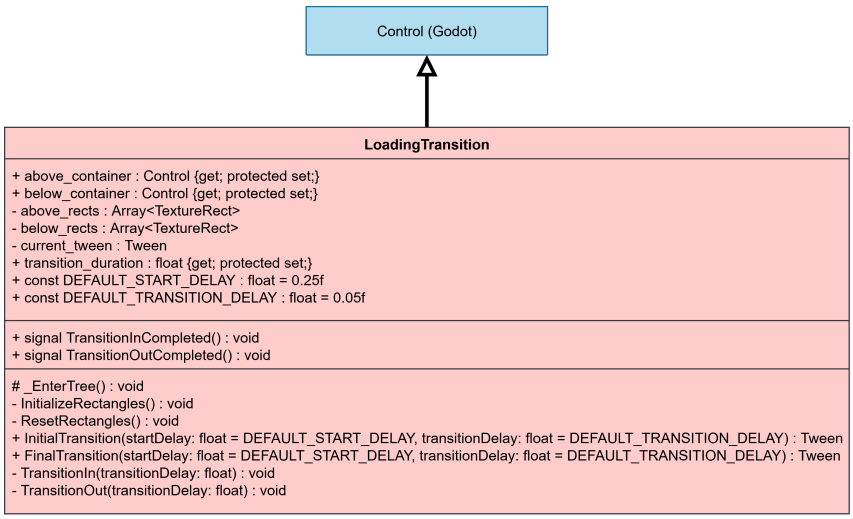


Figura 114: Representación UML de la clase *LoadingTransition*.

Para ver más detalles sobre la transición de carga, ver en el GDD (Apéndice B, sección 5.5).

4.19.4. HUD (*Head-Up Display*)

El HUD del juego está implementado en la clase *LevelUI*, que se instancia como interfaz principal durante la partida. Este componente agrupa y gestiona los elementos clave de la interfaz en tiempo real: la salud (*HealthUI*), la interfaz de proyectiles (*ProjectilesUI*), el sistema de diálogos (*DialogueUI*) y la leyenda de controles (*InputLegendUI*). Además, incluye una etiqueta de título para la salud, que se traduce automáticamente según el idioma seleccionado.

La clase *LevelUI* centraliza la inicialización y actualización de estos subcomponentes, suscribiéndose al sistema de idiomas para actualizar los textos dinámicamente. Cada elemento se obtiene como nodo hijo y se expone mediante propiedades públicas, facilitando el acceso y la interacción desde otros sistemas del juego.

Esta arquitectura modular permite que cada subcomponente gestione su propia lógica y visualización, mientras que *LevelUI* actúa como punto de entrada y coordinación para el HUD.

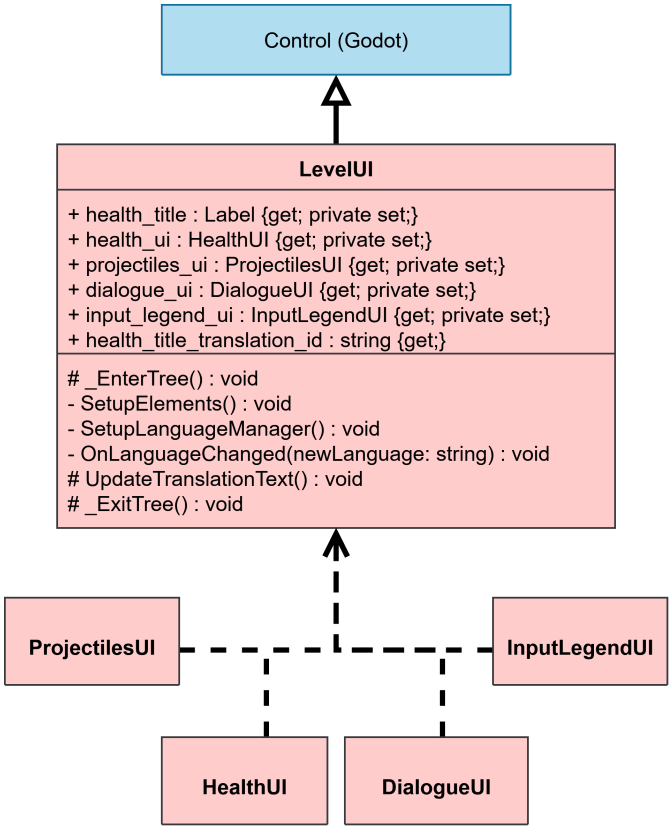


Figura 115: Representación UML de la clase *LevelUI*.

A continuación se describen los subcomponentes principales del HUD:

- **Interfaz de salud**

  La interfaz de salud (*HealthUI*) muestra visualmente el estado de los corazones del jugador en tiempo real. Está compuesta por un contenedor con bordes y fondo, los iconos de corazones y comas, y los corchetes que enmarcan el conjunto de salud. El número de corazones se adapta dinámicamente según la configuración de la entidad.

  Cada corazón es una instancia del componente *HeartUI*, que gestiona su color, visibilidad y efectos de partículas al perder o recuperar salud. El sistema permite actualizar el color de cada corazón individualmente, ocultar o mostrar corazones según el daño recibido, y reproducir partículas para reforzar la retroalimentación visual.

  El diseño modular facilita la actualización de la interfaz: al cambiar el número de corazones, se recrean los iconos y se ajusta la visibilidad de las comas y corchetes. El componente también permite modificar la transparencia del borde y fondo, útil para transiciones o efectos especiales o para hacerlos invisibles en el caso de la interfaz seguidora (sección 4.19.5).

  En resumen, *HealthUI* centraliza la gestión visual de la salud, integrando animaciones, colores y efectos para ofrecer una representación clara y dinámica del estado vital del jugador.

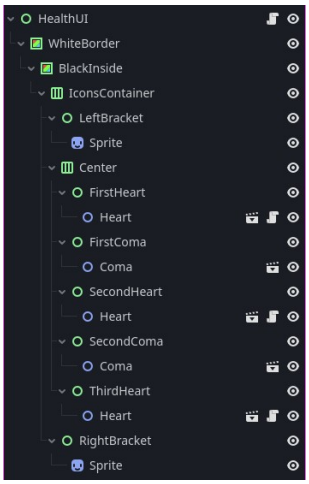


Figura 116: Árbol de nodos de la *HealthUI*.

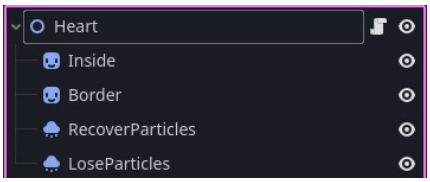


Figura 117: Árbol de nodos del componente *HeartUI*.

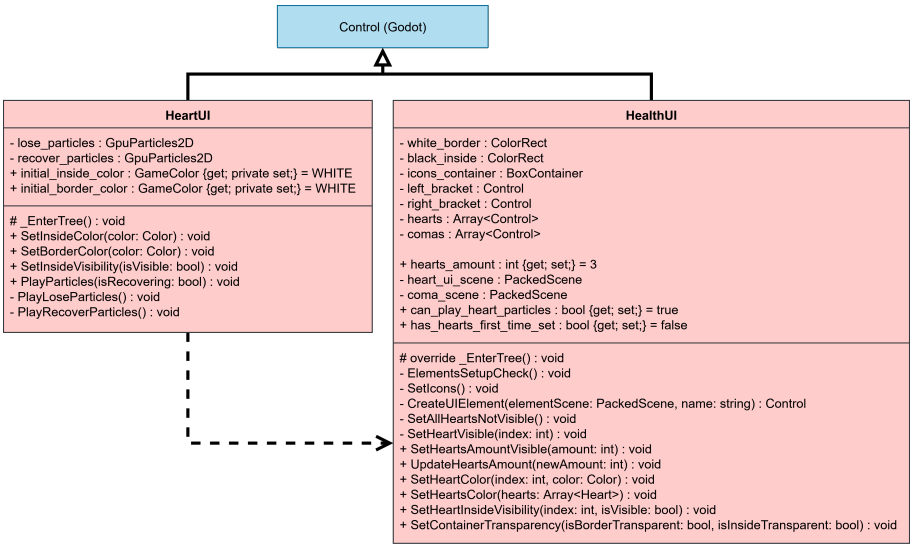


Figura 118: Representación UML de la clase *HealthUI* y *HeartUI*.

- **Interfaz de proyectiles**

  La interfaz de proyectiles (*ProjectilesUI*) muestra al jugador los tipos de proyectiles y colores disponibles para disparar en cada momento. Está compuesta por tres paneles principales: el proyectil anterior, el actual y el siguiente, cada uno con su icono y color correspondiente. El proyectil actual incluye además un efecto de partículas que refuerza la retroalimentación visual.

  Debajo de los proyectiles se encuentra una fila de cuadrados de color (*ColorSquareUI*), cada uno representando un color de corazón que puede portar el proyectil. El cuadrado seleccionado se muestra resaltado, indicando el color activo. El jugador puede cambiar tanto el tipo de proyectil como el color mediante las entradas configuradas, y la interfaz se actualiza dinámicamente para reflejar estos cambios.

  El sistema gestiona la visibilidad, el color y el tipo de proyectil mostrado, asegurando

que los iconos se adapten correctamente según el tipo de daño y el color seleccionado. Si un proyectil ignora el color (por su tipo de daño), se muestra con el color por defecto.

Esta interfaz facilita la comprensión y el control de las mecánicas de disparo, integrando la selección visual y la retroalimentación de partículas para mejorar la experiencia de usuario.

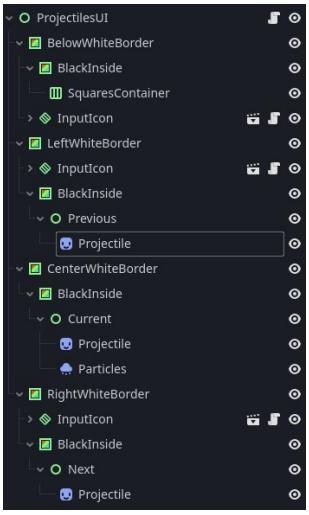


Figura 119: Árbol de nodos de la *ProjectilesUI*.

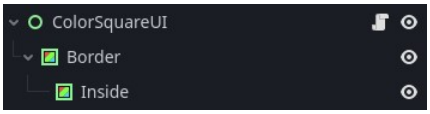


Figura 120: Árbol de nodos del componente *ColorSquareUI*.

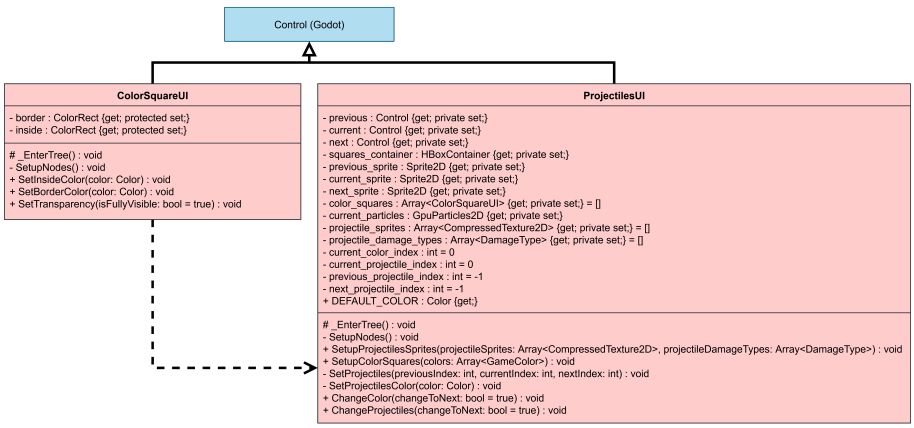


Figura 121: Representación UML de la clase *ProjectilesUI* y *ColorSquareUI*.

- **Interfaz de diálogos**

  La interfaz de diálogos (*DialogueUI*) permite mostrar diálogos en pantalla durante la partida, gestionando tanto el texto como la interacción del jugador. Está compuesta por un fondo con efecto de desvanecimiento, un texto enriquecido (*RichTextLabel*) para mostrar el diálogo y un icono de entrada (*InputIcon*) que indica cómo avanzar el diálogo.

  El sistema utiliza animaciones suaves (*Tween*) para mostrar y ocultar el fondo, el texto y el icono de entrada, mejorando la experiencia visual. Los diálogos se gestionan mediante una lista de objetos *Dialogue*, cada uno representando una línea de texto con su identificador de traducción, opciones de avance y permiso de avance por el jugador.

  El jugador puede avanzar el diálogo mediante una acción configurada (*next_dialogue*), siempre que el diálogo lo permita. Además, cada línea puede requerir un *AdvanceDialogueTrigger* (explicado en la sección 4.22.1) específico para avanzar, permitiendo sincronizar el diálogo con eventos del juego (por ejemplo, llegar a un cierto área para que el diálogo avance).

  El componente *Dialogue* es un recurso global (*GlobalClass*) que encapsula las propiedades de cada línea de diálogo: el identificador de traducción, si permite avanzar con entrada, el ruta del nodo *trigger* para avanzar (si requiere) y si el jugador puede realizar acciones durante el diálogo (moverse, disparar, etc.). Esto facilita la edición y reutilización de diálogos desde el editor de Godot, similar a otros recursos como *LevelData*.

  La interfaz gestiona automáticamente la traducción de los textos al cambiar el idioma y emite *signals* para notificar el inicio, actualización y finalización de cada diálogo, permitiendo que otros sistemas reaccionen en consecuencia.

  En resumen, *DialogueUI* centraliza la presentación y control de los diálogos, integrando animaciones, traducción y lógica de avance para ofrecer una experiencia narrativa fluida y flexible.

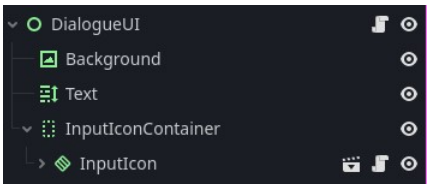


Figura 122: Árbol de nodos del componente *DialogueUI*.

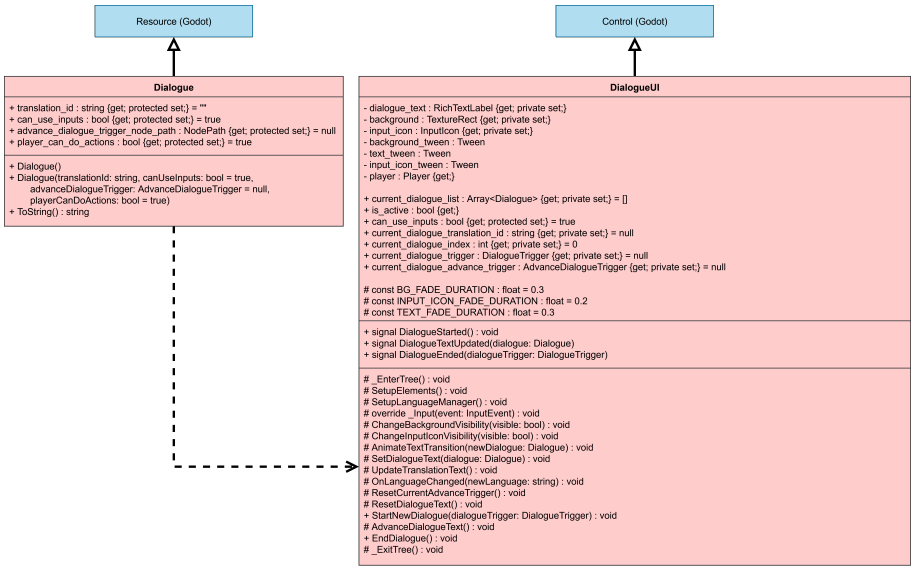


Figura 123: Representación UML de la clase *DialogueUI* y *Dialogue*.

- **Interfaz de leyendas de *input***

  La interfaz de leyendas de *input* (*InputLegendUI*) muestra al jugador las acciones disponibles y los iconos correspondientes según el tipo de entrada (teclado o mando). Este componente se utiliza para informar al jugador sobre los controles según cada contexto, facilitando la accesibilidad y la comprensión de las mecánicas del juego, se utilizan a modo de tutoriales visuales para la primera vez que se introduce una nueva acción.

  Cada leyenda de acción se representa mediante el recurso global *InputLegend*, que almacena el identificador de traducción del nombre de la acción y las texturas de los iconos para teclado, mando y sus variantes alternativas. Esto permite configurar fácilmente las leyendas desde el editor y adaptarlas a diferentes idiomas y controles.

  La interfaz gestiona dinámicamente la lista de acciones mostradas, añadiendo o eliminando elementos. El fondo de la interfaz se anima mediante interpolaciones (*Tween*) para

mostrar u ocultar la leyenda de forma suave, adaptando el tamaño y la posición según el contenido.

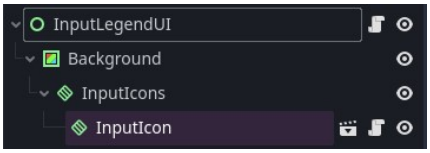


Figura 124: Árbol de nodos de la *InputLegendUI*.

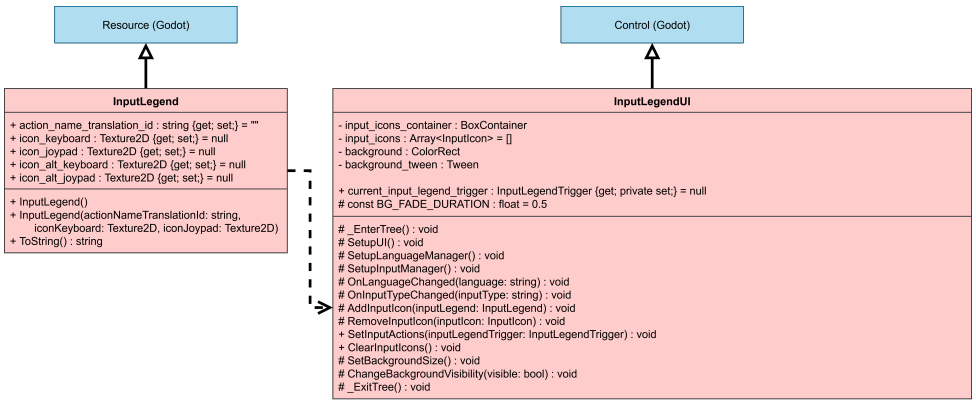


Figura 125: Representación UML de la clase *InputLegendUI* e *InputLegend*.

Para ver más detalles sobre el HUD, ver en el GDD (Apéndice B, sección 7).

4.19.5. Interfaz seguidora

La interfaz seguidora (*FollowerUI*) es un componente especializado que permite mostrar elementos de interfaz (como la salud) que siguen dinámicamente a una entidad en el mundo del juego. Este sistema es útil para mostrar información relevante sobre entidades concretas, como los enemigos, manteniendo la posición del elemento de interfaz sincronizada con la entidad objetivo.

El componente gestiona la posición mediante interpolación suave, evitando movimientos bruscos y permitiendo configurar un *offset* para ajustar la ubicación relativa respecto a la entidad. Además, incluye una zona muerta (*dead zone*) para evitar micro-movimientos cuando la distancia es mínima, mejorando la estabilidad visual.

La interfaz permite bloquear la rotación del elemento de UI o hacer que siga la rotación de la entidad, según la configuración. También gestiona la visibilidad del elemento, ocultándolo

automáticamente si la entidad objetivo no es visible.

El sistema es flexible y configurable desde el editor, permitiendo seleccionar la entidad objetivo, ajustar el *offset*, la velocidad de seguimiento y el comportamiento de rotación. Esto facilita la integración de elementos de interfaz personalizados que acompañan a cualquier entidad en tiempo real.

La interfaz seguidora se utiliza principalmente para mostrar la salud de los enemigos, pero puede adaptarse para otras interfaces si fuera necesario.

Además de los enemigos, la salud se adapta también en el jugador si el usuario lo desea (sección 4.19.2) desde los ajustes, lo que hace útil ver la salud en todo momento sin necesidad de mirar al HUD principal. Para más detalles sobre la interfaz seguidora en el jugador, ver en el GDD (Apéndice B, sección 7.1.1).

La interfaz seguidora de salud se instancia por entidad desde el código de *Entity*, con todas las configuraciones necesarias, y se elimina automáticamente al morir la entidad.

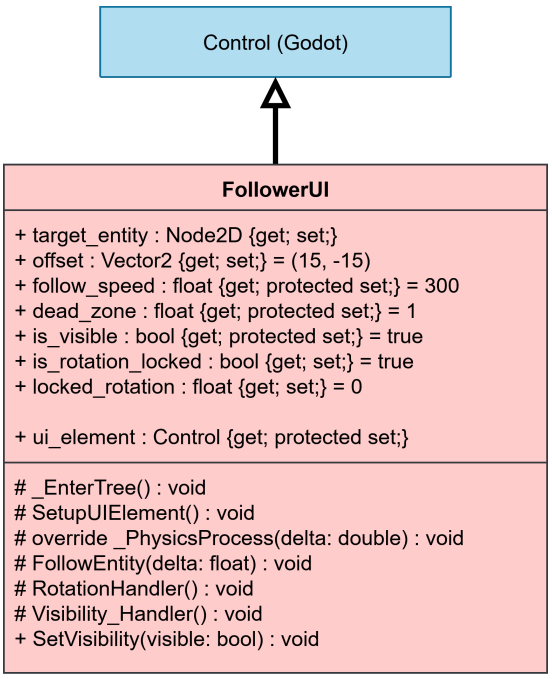


Figura 126: Representación UML de la clase *FollowerUI*.

4.20. Niveles

La clase *Level* representa la escena principal de cada nivel jugable. Hereda de *Node2D* y centraliza la gestión de los elementos clave del nivel: la cámara principal (*LevelCamera*), el

jugador (*Player*), la mirilla (*Scope*), el sistema de aparición del jugador (*PlayerSpawner*), y contenedores auxiliares para partículas e interfaces seguidoras.

Al iniciar el nivel, el sistema configura los nodos principales y aplica la configuración específica del jugador definida en el nodo *LevelConfiguration*. El proceso de aparición del jugador está animado mediante el nodo *PlayerSpawner* (sección 4.23.3), que controla la secuencia de entrada del jugador en el nivel. Una vez completada la animación, el jugador y la mirilla se activan y quedan listos para la partida.

El sistema de *signals* permite notificar eventos importantes, como el inicio de la partida, el fin del nivel por muerte del jugador (*Game Over*) o la compleción del nivel. Al completar el nivel, se desactivan el jugador y la mirilla, y la cámara puede seguir un objetivo especial (como el portal de fin de nivel, sección 4.23.4) o permanecer estática.

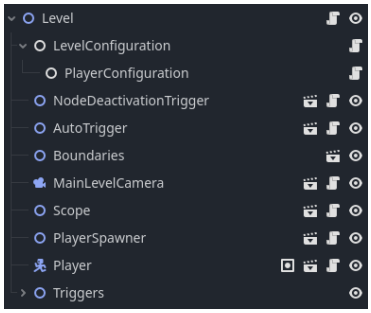


Figura 127: Ejemplo del árbol de nodos de un *Level*.

Para ver más detalles sobre los niveles, como la lista de niveles creados y sus características, ver en el GDD (Apéndice B, sección 10).

4.20.1. Configuración de niveles

La configuración de cada nivel se gestiona mediante el nodo *LevelConfiguration*, que permite definir las condiciones iniciales del jugador a través del nodo *PlayerConfiguration*. Esta configuración incluye los corazones iniciales de salud, los colores de corazones que pueden portar los proyectiles y los tipos de proyectiles disponibles.

Los valores se pueden editar fácilmente desde el editor de Godot, facilitando la creación y ajuste de niveles sin necesidad de modificar el código.

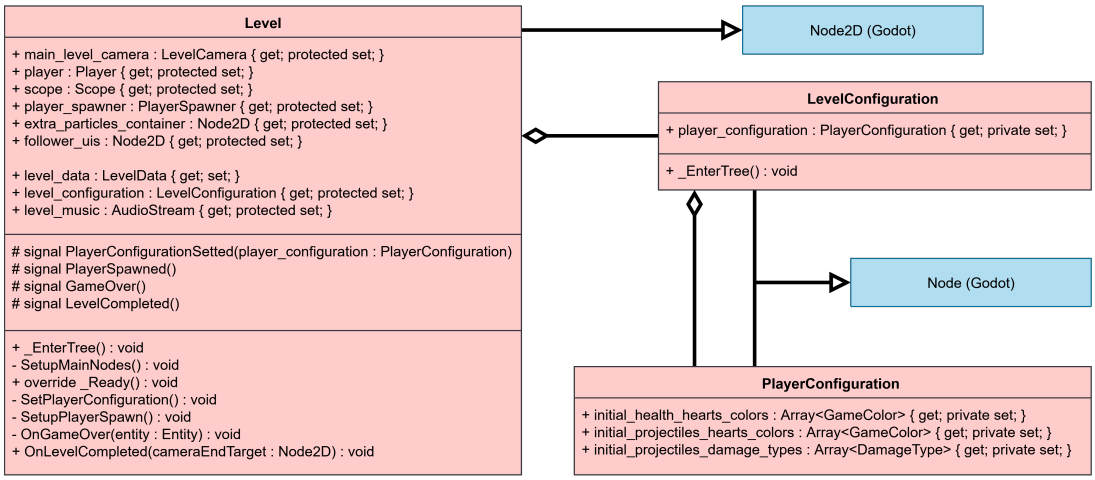


Figura 128: Representación UML de la clase *Level*.

4.21. Gestor de datos guardados

El sistema de guardado del juego está basado en un una clase abstracta (*DataManager*) que permite almacenar y recuperar datos persistentes en archivos nativos de Godot (*.res* o *.tres*). Este sistema es fundamental para conservar tanto la configuración del jugador como su progreso entre sesiones.

El gestor de datos se organiza en dos grandes bloques: los datos de configuración (*SettingsData*) y los datos de progreso (*ProgressData*), cada uno con su propio gestor especializado (*SettingsDataManager* y *ProgressDataManager*). Ambos gestores heredan de la clase abstracta *DataManager*, que centraliza la lógica de carga, guardado y creación de archivos de datos. Además, la clase *GameData* actúa como punto de acceso único para ambos gestores, facilitando la interacción con los datos desde cualquier parte del juego.

El archivo de datos se almacena en una ruta específica dentro de la carpeta de usuario (*user://data/*), esto ya fue explicado en la sección 4.3.4.

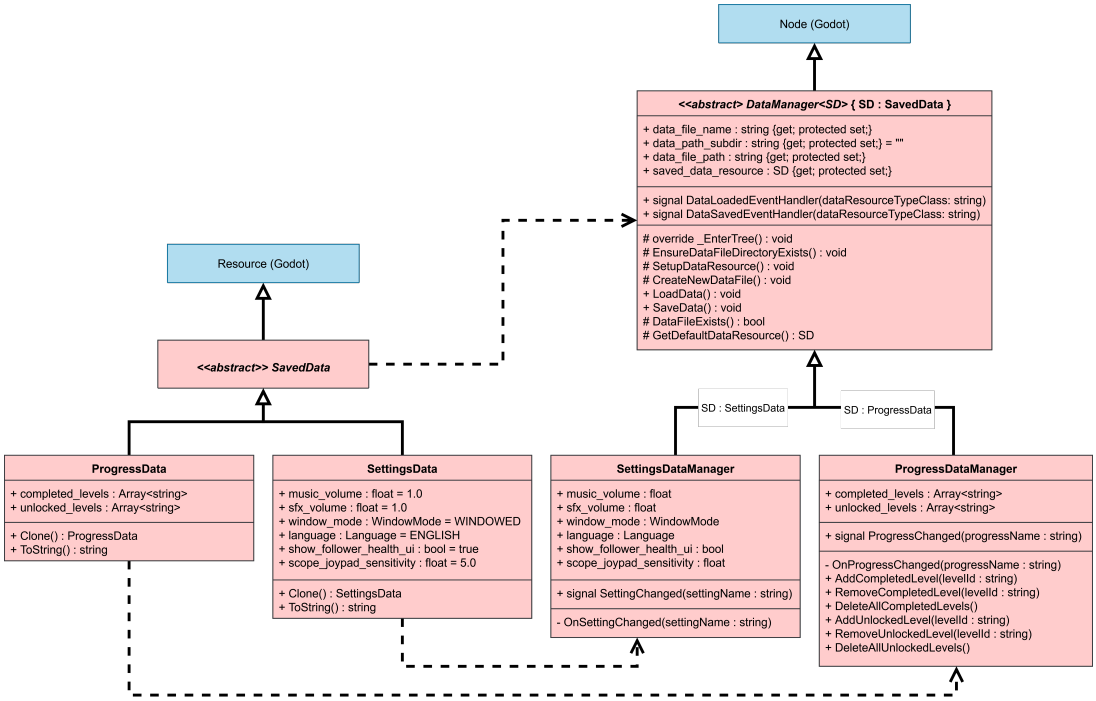


Figura 129: Representación UML de la clase *DataManager* y *SavedData* y los derivados de ambos.

4.21.1. Archivo de datos guardados

Cada archivo de datos se crea y gestiona mediante el gestor correspondiente. Al iniciar el juego, el sistema comprueba si el archivo existe; si no, crea uno nuevo con valores por defecto. La subruta y el nombre del archivo se configuran según cada gestor desde el editor.

El gestor se encarga de crear la carpeta de datos si no existe, utilizando métodos nativos de Godot para asegurar la correcta estructura de directorios. Los archivos se guardan en formato *.res* para uso normal y en formato *.tres* en modo depuración, facilitando la lectura y edición manual si es necesario.

La clase base *DataManager* proporciona métodos para cargar, guardar y comprobar la existencia del archivo, así como *signals* para notificar cuando los datos han sido cargados o guardados. Para ver más detalles sobre el sistema de datos guardados, ver en el GDD (Apéndice B, sección 11).

A continuación se describen los dos tipos de datos gestionados, es decir, las dos implementaciones concretas de este sistema:

4.21.2. Datos de configuración

Los datos de configuración (*SettingsData*) almacenan las preferencias del jugador: volumen de música y efectos, modo de ventana, idioma, visibilidad de la interfaz seguidora de salud del jugador y sensibilidad de la mirilla con mando. Estos datos se guardan y recuperan automáticamente mediante el gestor *SettingsDataManager*, que expone propiedades públicas para cada ajuste y emite *signals* cuando se modifica algún valor, lo que permite al *GameSettings* (sección 4.3.5) aplicar los cambios en tiempo real.

Cada vez que el jugador cambia una opción en el menú de ajustes, el gestor actualiza el valor correspondiente y guarda el archivo de datos, asegurando que los cambios persistan entre sesiones.

4.21.3. Datos de progreso

Los datos de progreso (*ProgressData*) registran los niveles completados y desbloqueados por el jugador. El gestor *ProgressDataManager* proporciona métodos para añadir, eliminar y consultar niveles en ambas listas, así como para borrar todo el progreso si es necesario.

Cada vez que el jugador completa o desbloquea un nivel, el gestor actualiza el archivo de datos y emite una *signal* para notificar el cambio. Esto permite que otros sistemas (como el menú de selección de nivel) se actualicen según el progreso guardado.

El sistema utiliza listas de identificadores de nivel (*strings*) como *LevelData*.

4.22. *Triggers*

El sistema de *triggers* es una de las bases para la interacción y la lógica dinámica del juego. Un *trigger* es un objeto que detecta y responde a eventos específicos en el entorno, permitiendo ejecutar acciones cuando se cumplen ciertas condiciones. Los *triggers* pueden activarse o desactivarse según su configuración, y pueden combinarse entre sí para crear interacciones complejas.

La clase base abstracta *Trigger* hereda de *Node2D* y define la estructura común para todos los tipos de *triggers* del juego. Cada *trigger* puede operar en dos modos: **OneWay** (solo se activa una vez) o **TwoWay** (puede alternar entre activado y desactivado). Además, se pueden configurar propiedades como la posibilidad de activarse varias veces, ser desactivable, o estar habilitado/deshabilitado (es decir, que puede activarse y desactivarse o no).

El sistema utiliza (*signals*) para notificar cuando un *trigger* se activa o desactiva, permitiendo que otros nodos reaccionen en consecuencia. Los *triggers* pueden responder a diferentes tipos de eventos: acciones del jugador, cambios en el entorno, condiciones lógicas, etc. También pueden ejecutar distintas acciones al activarse o desactivarse, como iniciar un diálogo o completar un objetivo.

La arquitectura modular y extensible de los *triggers* permite crear fácilmente nuevos tipos de condiciones y acciones, combinando *triggers* básicos para construir sistemas de eventos complejos y adaptados a las necesidades de cada nivel. La idea es construir los niveles desde el editor de Godot utilizando estos *triggers* para definir la lógica del juego de manera visual y sencilla, sin necesidad de programar cada interacción desde cero.

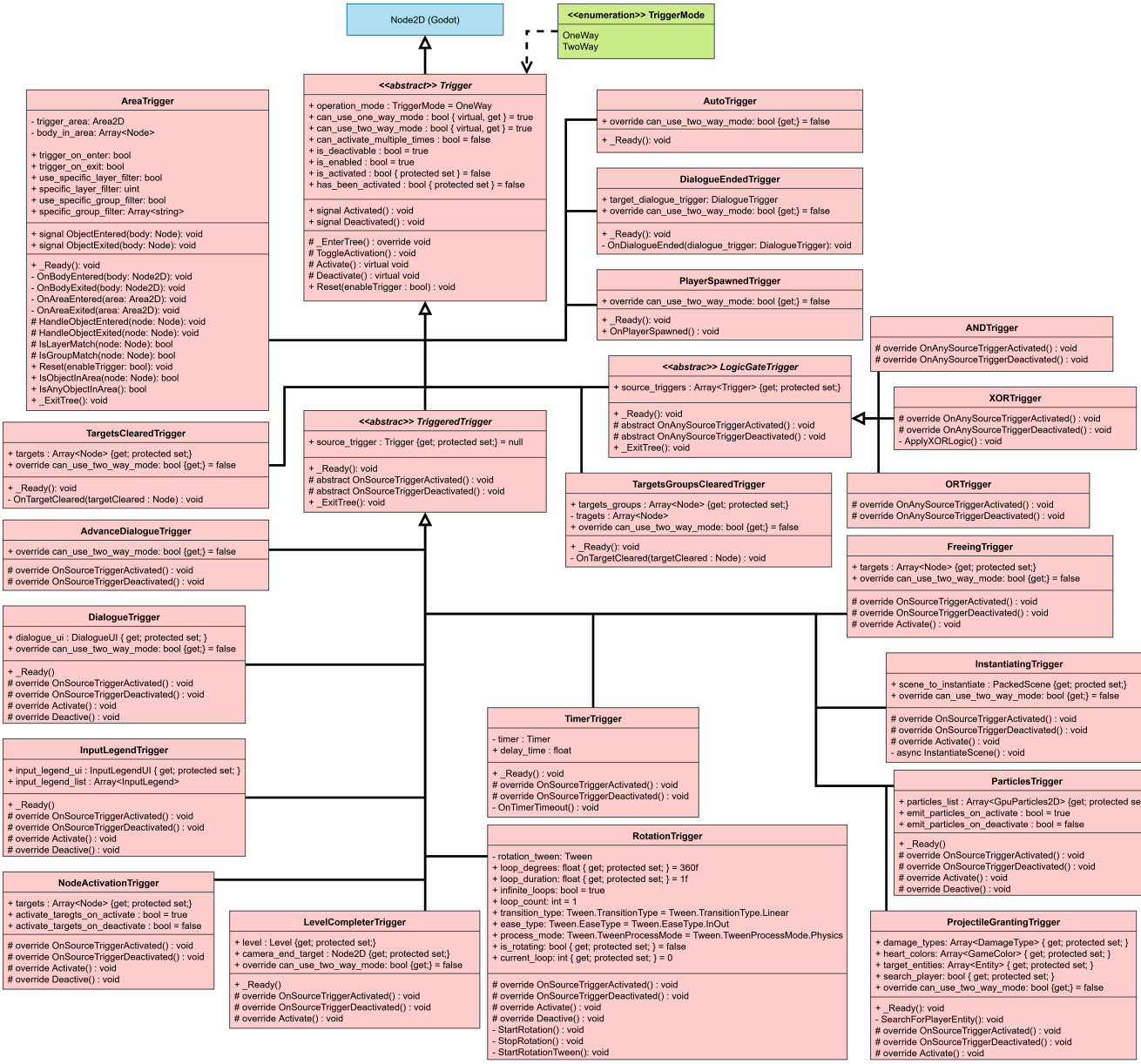


Figura 130: Diagrama de clase de *Trigger* y sus derivados.

A continuación se describen los principales tipos de *triggers* implementados en el juego y sus aplicaciones prácticas:

- ***Trigger* de Area**

  El *AreaTrigger* es uno de los tipos más utilizados de *trigger* en el juego. Su función principal es detectar cuándo un objeto (jugador, enemigo, área, etc.) entra o sale de una zona definida mediante un nodo *Area2D*. Al producirse la entrada o salida, el *trigger* puede

activarse o desactivarse según su configuración.

Este *trigger* permite filtrar los objetos detectados tanto por capas de físicas (*Physics Layers*) como por grupos (*GameGroups*), lo que facilita limitar la activación a entidades concretas (por ejemplo, solo el jugador). Los filtros se configuran desde el editor, permitiendo seleccionar las capas y grupos relevantes para cada caso.

El sistema gestiona internamente una lista de objetos presentes en el área, emitiendo señales (*ObjectEntered*, *ObjectExited*) cada vez que un objeto entra o sale. Además, permite configurar si la activación se produce al entrar (*trigger_on_enter*), al salir (*trigger_on_exit*), o en ambos casos.

El *AreaTrigger* es especialmente útil para crear zonas de activación en el escenario, como inicio o fin de diálogos, portales de fin de nivel o *items*. Su arquitectura flexible y configurable lo convierte en una herramienta fundamental para la interacción dinámica en los niveles.

- ***Trigger* de Jugador *spawneado***

  El *PlayerSpawnedTrigger* es un tipo de *trigger* que se activa automáticamente cuando el jugador aparece en el nivel. Este *trigger* se conecta a la señal *PlayerSpawned* emitida por el sistema de gestión del nivel (*Level*), y al recibirla ejecuta su activación.

  A diferencia de otros *triggers*, el *PlayerSpawnedTrigger* solo puede operar en modo *OneWay*, es decir, se activa una única vez y no puede desactivarse. Esto lo hace ideal para iniciar eventos o acciones que deben ocurrir justo al comenzar la partida, como mostrar un tutorial, iniciar un diálogo, activar enemigos o desbloquear zonas.

  Su implementación es sencilla y directa, facilitando la integración de lógica dependiente del inicio de la partida sin necesidad de comprobaciones adicionales.

- ***Trigger* automático**

  El *AutoTrigger* es un tipo de *trigger* que se activa automáticamente en cuanto está listo en el árbol de nodos, es decir, al instanciarse en la escena. No requiere ninguna condición externa ni interacción del jugador o del entorno para activarse.

  Este *trigger* solo puede operar en modo *OneWay*, activándose una única vez y sin posibilidad de desactivación. Es útil para iniciar eventos, animaciones, diálogos o cualquier

acción que deba ocurrir de forma inmediata al cargar una escena o nivel pero antes del *spawn* del jugador.

También se puede instanciar estando desactivado el nodo, y luego activarlo desde otro sistema para que se active automáticamente en ese momento.

Su implementación es muy sencilla: al entrar en el árbol de nodos (*_Ready()*), llama directamente al método de activación. Esto facilita la integración de lógica que debe ejecutarse automáticamente, sin depender de otros sistemas o condiciones.

- ***Trigger* de diálogo acabado**

  El *DialogueEndedTrigger* es un tipo de *trigger* que se activa automáticamente cuando finaliza un diálogo concreto en el juego. Para ello, se conecta a la señal *DialogueEnded* emitida por la interfaz de diálogos (*DialogueUI*, sección 4.19.4) y comprueba si el diálogo finalizado corresponde al *DialogueTrigger* (sección 4.22.1) objetivo configurado en el editor.

  Este *trigger* solo puede operar en modo *OneWay*, activándose una única vez y sin posibilidad de desactivación. Es especialmente útil para sincronizar eventos que deben ocurrir justo después de que termine un diálogo, como desbloquear zonas, activar enemigos, mostrar tutoriales o avanzar en la narrativa del nivel.

  Su implementación facilita la integración de lógica dependiente de la finalización de diálogos, permitiendo crear cadenas de eventos y progresión narrativa de forma sencilla y modular.

- ***Trigger* de todos los objetivos eliminados**

  El *TargetsClearedTrigger* es un tipo de *trigger* que se activa automáticamente cuando todos los nodos objetivo configurados han sido eliminados del árbol de nodos (mediante *QueueFree* de Godot). Este *trigger* es útil para detectar cuándo el jugador ha derrotado a todos los enemigos, destruido todos los objetos de una zona, o completado cualquier conjunto de objetivos definidos.

  El sistema permite configurar una lista de nodos objetivo desde el editor. Al iniciar la escena, el *trigger* se suscribe al evento nativo *TreeExited* de cada nodo, de modo que

cuando uno de ellos es eliminado, se actualiza la lista interna. Cuando todos los objetivos han sido eliminados, el *trigger* se activa automáticamente.

Existe una variante llamada *TargetsGroupsClearedTrigger*, que permite configurar grupos de nodos objetivo. El sistema recorre todos los grupos y sus hijos, suscribiéndose al evento de eliminación de cada uno. El *trigger* se activa cuando todos los nodos de todos los grupos han sido eliminados.

Estos *triggers* son especialmente útiles para gestionar la progresión de niveles, activar portales de fin de nivel, recompensas, o desencadenar eventos tras limpiar una zona de enemigos u obstáculos.

4.22.1. *Triggered Trigger*

Los *Triggered Trigger* son una subclase especial de *trigger* cuyo propósito es activarse o desactivarse únicamente en respuesta a otro *trigger*. En lugar de detectar eventos del entorno o del jugador directamente, estos *triggers* se conectan a las señales (*Activated* y *Deactivated*) de un *trigger* fuente (*source_trigger*), que se configura desde el editor. Es abstracta, para que se deriven las clases concretas de *trigger* que se necesiten.

Esta arquitectura permite encadenar acciones y crear sistemas de lógica compleja, donde un evento inicial activa una serie de *triggers* secundarios que realizan distintas acciones (como instanciar nodos, eliminar objetos, mostrar mensajes, etc.). El sistema comprueba que el *trigger* fuente esté correctamente configurado y que no sea él mismo, evitando bucles lógicos.

Cada clase derivada de *TriggeredTrigger* implementa los métodos para manejar la activación y desactivación del *trigger* fuente, permitiendo definir la reacción específica ante cada evento. Esto facilita la construcción de cadenas de eventos y la modularidad en el diseño de niveles, ya que los *triggers* pueden combinarse y reutilizarse fácilmente.

Los *Triggered Trigger* son fundamentales para la lógica visual y la interacción avanzada en el juego, ya que la estos *triggers* suelen encargarse más de realizar acciones concretas que de detectar condiciones para activarse.

A continuación se explican los tipos de *Triggered Trigger* implementados en el juego y sus aplicaciones:

- ***Trigger* de diálogo**

El *DialogueTrigger* es un tipo de *Triggered Trigger* diseñado para iniciar y gestionar diálogos en el juego. Este *trigger* se activa en respuesta a otro *trigger* fuente, permitiendo encadenar la aparición de diálogos cuando se entra en un área o se completa un objetivo, por ejemplo.

Al activarse, el *DialogueTrigger* utiliza la interfaz de diálogos (*DialogueUI*, sección 4.19.4) para mostrar una lista de líneas de diálogo (*Dialogue*) previamente configurada desde el editor. El sistema permite definir el contenido, el orden y las condiciones de avance de cada línea (porque cada *Dialogue* es lo que tiene de atributos), integrando la narrativa con la lógica del nivel.

Si el *trigger* fuente se desactiva, el *DialogueTrigger* puede cerrar el diálogo y resetear su estado, permitiendo una gestión flexible de la interacción. Esto facilita la creación de secuencias narrativas, tutoriales y eventos dependientes de la progresión del jugador.

El *DialogueTrigger* está estrechamente vinculado al sistema de diálogos y a otros *triggers* como el *DialogueEndedTrigger*, permitiendo construir cadenas de eventos y progresión narrativa de forma modular y visual desde el editor de Godot.

- ***Trigger* de avance de diálogo**

  El *AdvanceDialogueTrigger* es un tipo de *Triggered Trigger* diseñado específicamente para avanzar una línea de diálogo dentro de la interfaz de diálogos (*DialogueUI*, sección 4.19.4). Este *trigger* debe ser hijo del *DialogueTrigger* correspondiente y se utiliza cuando se requiere que el avance del diálogo dependa de un evento externo, en vez de la entrada directa del jugador.

  Cuando el *trigger* fuente se activa (por ejemplo, al entrar en un área o completar una acción), el *AdvanceDialogueTrigger* emite su señal de activación, lo que provoca que la interfaz de diálogos avance a la siguiente línea. No tiene lógica de desactivación, ya que su único propósito es avanzar el diálogo en ese momento concreto.

  Este sistema permite sincronizar la narrativa con eventos del juego, como esperar a que el jugador llegue a una zona, interactúe con un objeto o complete una acción antes de mostrar la siguiente línea de diálogo. Así, se pueden crear secuencias narrativas más dinámicas y adaptadas al contexto del nivel, sin depender de que el jugador pulse un

botón para avanzar (porque se puede optar por no permitírselo).

- ***Trigger* de leyenda de *input***

  El *InputLegendTrigger* es un tipo de *Triggered Trigger* diseñado para mostrar o ocultar la interfaz de leyendas de *input* (*InputLegendUI*, sección 4.19.4) en respuesta a la activación de otro *trigger*. Este sistema permite informar al jugador sobre las acciones disponibles y sus controles en momentos clave del juego, como tutoriales o introducción de nuevas mecánicas.

  Al activarse, el *InputLegendTrigger* utiliza la lista de leyendas de *input* (*InputLegend*) configurada desde el editor para mostrar los iconos y textos correspondientes en la interfaz. Si la lista está vacía, la interfaz se oculta automáticamente, útil para quitarla de otros previos *triggers*. También funciona así cuando el *trigger* fuente se desactiva.

  Este sistema facilita la presentación dinámica de controles y acciones, adaptándose al contexto del juego y mejorando la accesibilidad y comprensión para el jugador. Es especialmente útil para mostrar tutoriales visuales o enseñar controles en situaciones específicas.

- ***Trigger* completador de nivel**

  El *LevelCompleterTrigger* es un tipo de *Triggered Trigger* diseñado para completar el nivel actual cuando se activa. Este *trigger* se utiliza para finalizar la partida de forma controlada, por ejemplo, al llegar a un portal de fin de nivel (único uso actual hasta ahora), completar todos los objetivos o tras una secuencia narrativa.

  Al activarse, el *LevelCompleterTrigger* llama al método de finalización del nivel (*OnLevelCompleted*) y puede indicar un objetivo especial para la cámara (*camera_end_target*), permitiendo centrar la atención en un punto concreto durante la transición (como el portal de fin de nivel). Esto desencadena la aparición del menú de nivel completado (*LevelCompletedMenu*), la actualización del progreso y la gestión de la transición visual y lógica del juego.

  Este sistema facilita la integración de condiciones de victoria y progresión, permitiendo que la finalización del nivel dependa de cualquier evento o cadena de *triggers* definidos en el nivel.

- ***Trigger* de eliminación de nodos**

  El *FreeingTrigger* es un tipo de *Triggered Trigger* que se utiliza para eliminar (*QueueFree* de Godot) automáticamente una lista de nodos especificados cuando se activa. Este *trigger* es útil para gestionar la eliminación de objetos, enemigos, obstáculos o cualquier nodo del escenario en respuesta a la activación de otro *trigger*, permitiendo construir secuencias y lógica visual de forma modular.

  Al activarse, el *FreeingTrigger* recorre la lista de nodos objetivo configurada desde el editor y aplica *QueueFree* a cada uno, eliminándolos del árbol de nodos de Godot. No tiene lógica de desactivación, ya que su único propósito es realizar la eliminación en el momento de la activación.

  Este sistema facilita la gestión dinámica del escenario, permitiendo que la eliminación de nodos dependa de cualquier evento o cadena de *triggers*, como completar objetivos, finalizar diálogos o activar zonas especiales.

- ***Trigger* de instanciación de nodos**

  El *InstantiatingTrigger* es un tipo de *Triggered Trigger* que se utiliza para instanciar automáticamente una escena (*PackedScene*) cuando se activa. Este *trigger* permite crear nuevos objetos, enemigos, efectos visuales o cualquier nodo en el escenario en respuesta a la activación de otro *trigger*, facilitando la construcción de eventos dinámicos y secuencias visuales.

  Al activarse, el *InstantiatingTrigger* instancia la escena configurada desde el editor y la añade como hijo del nodo padre del propio *trigger*, posicionando el nuevo nodo en la misma ubicación global. No tiene lógica de desactivación, ya que su único propósito es realizar la instanciación en el momento de la activación.

  Este sistema es especialmente útil para crear oleadas de enemigos, efectos especiales, objetos interactivos o cualquier elemento que deba aparecer en el escenario como resultado de una cadena de *triggers*.

- ***Trigger* de activación/desactivación de nodos**

  El *NodeActivationTrigger* es un tipo de *Triggered Trigger* que permite activar o desactivar dinámicamente una lista de nodos especificados cuando se activa o desactiva el *trigger*.

Utiliza el método *SetActive* implementado en las utilidades (*ProcessUtils*, sección 4.4.5), lo que garantiza que la visibilidad, el procesamiento y las colisiones de los nodos se gestionen correctamente y de forma recursiva.

Este sistema es especialmente útil para controlar la aparición y desaparición de elementos en el nivel de manera cómoda desde el editor de Godot. Por ejemplo, al inicio del nivel se puede utilizar un *AutoTrigger* junto con este *trigger* en modo desactivar para ocultar todos los elementos que no deben estar activos, y luego, mediante otros *triggers* de este tipo, ir activando los nodos necesarios conforme avanza la partida. Así, todos los elementos están instanciados y localizables desde el comienzo, facilitando la edición y el mantenimiento del nivel desde el editor.

El *NodeActivationTrigger* permite configurar si los nodos deben activarse al activar el *trigger*, desactivarse al desactivarlo, o ambas opciones, proporcionando flexibilidad para gestionar la lógica funcional del escenario.

- ***Trigger* de partículas**

  El *ParticlesTrigger* es un tipo de *Triggered Trigger* que permite controlar sistemas de partículas (*GpuParticles2D*) en respuesta a la activación o desactivación de otro *trigger*. Este sistema es especialmente útil para gestionar efectos visuales como explosiones, destellos, humo o cualquier animación de partículas que deba sincronizarse con eventos del juego.

  Al activarse, el *ParticlesTrigger* recorre todos los sistemas de partículas hijos y activa o desactiva su emisión según la configuración de sus propiedades. Esto permite mostrar efectos visuales en momentos clave, como la implementación que se ha hecho en el portal de fin de nivel (sección 4.23.4) y el *Spawner* del jugador (sección 4.23.3).

  El sistema está preparado para ser ampliado con más propiedades y configuraciones, aunque actualmente solo gestiona la emisión de partículas. Pero no se han necesitado más funcionalidades hasta ahora.

- ***Trigger* obsequiador de proyectiles**

  El *ProjectileGrantingTrigger* es un tipo de *Triggered Trigger* que permite otorgar nuevos tipos de daño de proyectil (*DamageType*) y colores de corazones (*GameColor*) a una lista

de entidades objetivo cuando se activa. Este sistema es fundamental para implementar mecánicas de mejora, progresión o recogida de *items* que amplían las capacidades ofensivas del jugador o de los enemigos.

Al activarse, el *ProjectileGrantingTrigger* recorre la lista de entidades objetivo y añade los tipos de daño y colores de corazones configurados desde el editor a sus respectivos gestores de proyectiles (*ProjectileManager*). Además, puede configurarse para buscar automáticamente al jugador y añadirlo a la lista de entidades objetivo.

Este *trigger* está estrechamente vinculado al sistema de *items* (sección 4.23.2), ya que permite que al recoger un *item* se otorguen nuevas habilidades de disparo al jugador, el cual es el uso principal que se le ha dado hasta ahora.

- ***Trigger* rotador**

  El *RotationTrigger* es un tipo de *Triggered Trigger* que permite aplicar rotaciones animadas a cualquier nodo cuando se activa. Este *trigger* utiliza interpolaciones (*Tween*) para rotar el nodo alrededor de su eje, pudiendo configurar el ángulo de rotación, la duración de cada ciclo, el tipo de transición, el modo de interpolación y si la rotación debe repetirse indefinidamente o un número concreto de veces.

  Al activarse, el *RotationTrigger* inicia la animación de rotación, incrementando el ángulo en cada ciclo y repitiendo la operación según la configuración. Al desactivarse, detiene la rotación y cancela cualquier animación en curso. Este sistema es especialmente útil para crear movimientos orbitales, rotaciones continuas o efectos visuales. Es utilizado actualmente en grupos de asteroides, haciendo que se desplazen en órbitas circulares infinitas alrededor de un punto central (aplicado en el nivel 4 y 5).

  Combinando varios *RotationTrigger* sobre diferentes nodos o ejes, es posible construir sistemas de movimiento complejos y dinámicos, como grupos de asteroides que rotan sobre un eje y que a su vez estos grupos roten alrededor de un punto central (esto se puede ver en el nivel 5).

- ***Trigger* de tiempo**

  El *TimerTrigger* es un tipo de *Triggered Trigger* que permite activar acciones tras un retraso temporal configurable. Este *trigger* utiliza un nodo *Timer* nativo de Godot, que

inicia la cuenta atrás cuando el *trigger* fuente se activa. Al finalizar el tiempo especificado (*delay_time*), el *TimerTrigger* se activa, permitiendo que otros *triggers* o acciones se ejecuten.

Es especialmente útil para crear secuencias temporizadas o para espaciar eventos en el tiempo sin depender de la entrada directa del jugador. Por ejemplo, puede utilizarse para mostrar mensajes tras una pausa, activar enemigos con retardo, o sincronizar efectos visuales.

Si el *trigger* fuente se desactiva antes de que termine el tiempo, el temporizador se detiene y no se activa el *trigger*. Esto permite controlar la lógica temporal de forma precisa y flexible desde el editor.

4.22.2. *Trigger* puerta lógica

Los *Logic Gate Trigger* son una subclase especial de *trigger* que permiten combinar múltiples *triggers* fuente y evaluar condiciones lógicas para decidir su activación. Estos *triggers* permiten construir sistemas de lógica compleja y dependencias entre eventos en el nivel de forma visual y modular desde el editor de Godot.

Cada *Logic Gate Trigger* mantiene una lista de *triggers* fuente (*source_triggers*) y se conecta a sus señales de activación y desactivación. Cuando cualquiera de los *triggers* fuente cambia de estado, la puerta lógica evalúa su condición y decide si debe activarse o desactivarse según la operación lógica configurada en su implementación concreta.

Este sistema es especialmente útil para crear mecanismos que requieran la combinación de varios eventos, como puertas que solo se abren si se activan varios interruptores, zonas que requieren cumplir varias condiciones simultáneas, o secuencias que dependen de la resolución de varios objetivos.

La arquitectura de las puertas lógicas permite implementar fácilmente diferentes tipos de operaciones (AND, OR, XOR, etc.), simplemente sobrescribiendo los métodos de evaluación en las clases derivadas.

A continuación se describen los tres tipos de puertas lógicas implementadas en el juego:

- ***Trigger* AND**

El *ANDTrigger* es una puerta lógica que se activa únicamente cuando todos sus *triggers* fuente están activados. Es útil para crear mecanismos que requieren cumplir varias condiciones simultáneas, como zonas que requieren completar varios objetivos antes de avanzar. Si alguno de los *triggers* fuente se desactiva, el *ANDTrigger* se desactiva automáticamente si está en modo *TwoWay*.

- ***Trigger* OR**

  El *ORTrigger* es una puerta lógica que se activa cuando al menos uno de sus *triggers* fuente está activado. Es útil para situaciones donde basta con cumplir una de varias condiciones para activar una acción, como llegar cualquiera de un grupo de áreas específicas. Si todos los *triggers* fuente se desactivan, el *ORTrigger* se desactiva.

- ***Trigger* XOR**

  El *XORTrigger* es una puerta lógica que se activa cuando un número impar de sus *triggers* fuente está activado. Es útil para crear mecanismos que requieren alternancia o lógica exclusiva, como activar una acción solo si se cumple una condición pero no varias a la vez. Si el número de *triggers* fuente activados es par, el *XORTrigger* se desactiva.

4.23. Elementos del escenario

Esta sección recoge los elementos misceláneos diseñados específicamente para los escenarios de los niveles. Se trata de componentes que aportan funcionalidad, interacción o efectos visuales únicos en cada nivel.

La característica principal de estos elementos es que su funcionamiento suele ser una composición de varios sistemas: combinan el uso de *triggers*, utilidades y otros módulos del juego junto con *scripts* propios que gestionan sus animaciones, lógica interna o interacción con el jugador y las entidades. De este modo, se consigue que cada elemento sea fácilmente configurable y ampliable desde el editor, manteniendo la modularidad y la coherencia con el resto del proyecto.

A continuación se describen los principales elementos implementados para los escenarios y su integración con el sistema general:

4.23.1. Límites del mapa

Los límites del mapa son elementos esenciales en cada nivel, encargados de definir las fronteras físicas del escenario y evitar que el jugador salga del área jugable. Se implementan mediante cuatro paredes (*BoundaryWall*), una para cada lado del nivel, instanciadas y configuradas desde el editor.

Cada *BoundaryWall* utiliza un nodo *Area2D* con una forma de colisión (*WorldBoundaryShape2D*) que detecta cuando una entidad entra en contacto con el límite. Al producirse la colisión, la pared aplica daño de tipo *Intersection* con conjunto vacío (*empty set*) a la entidad, lo que provoca su muerte inmediata (ver sección 4.10.1). Este comportamiento garantiza que cruzar el límite del escenario tenga consecuencias inmediatas y claras.

Visualmente, cada pared incluye partículas animadas (*GpuParticles2D*) del símbolo del conjunto vacío (en gran cantidad) y una línea (*Line2D*) que se muestra cerca del jugador cuando se aproxima al límite, reforzando la percepción visual de la frontera. La posición y longitud de la línea se actualizan dinámicamente en función de la posición del jugador y la orientación de la pared, creando un efecto de advertencia antes de cruzar el límite y que delimita claramente el área jugable ya que las partículas no lo definen bien.

El sistema de límites del mapa es un ejemplo de composición de elementos: combina la detección de áreas, la gestión de partículas y la lógica de daño. Aunque es la única excepción en la que todo es manejado desde su propio *script* y no se ha usado ningún *trigger*, pero podría implementarse de ese modo.

Para ver más sobre los límites del mapa, consultar en el GDD (sección B, sección 9.7.3).

Figura 131: Árbol de nodos de una *BoundaryWall*.

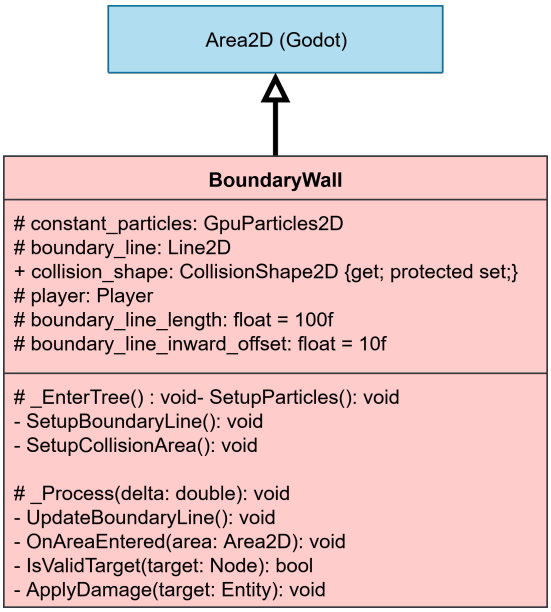


Figura 132: Representación UML de la clase *BoundaryWall*.

4.23.2. *Items*

Los *items* son elementos interactivos que el jugador puede recoger en los niveles para obtener nuevos tipos de proyectiles o colores de corazones. Su funcionamiento se basa en la composición de varios sistemas: la clase *ProjectileGrantingItem* (heredada de la abstracta *Item*) gestiona la parte visual y la reacción ante la activación, mientras que la concesión de mejoras y efectos se realiza mediante una cadena de *triggers* conectados.

Cada *item* incorpora un *activation_trigger*, normalmente un *AreaTrigger* circular que detecta la entrada del jugador en el área del *item*. Al activarse, se encadenan otros *Triggered Trigger* que gestionan la aparición de partículas (*ParticlesTrigger*), la concesión de proyectiles y colores (*ProjectileGrantingTrigger*), y la eliminación del propio *item* (*FreeingTrigger*) tras un breve retardo (*TimerTrigger*). Esta arquitectura modular permite configurar y ampliar fácilmente los efectos y recompensas de cada *item* desde el editor, sin modificar el código principal.

Destaca el uso del efecto de luz parpadeante, implementado mediante la clase *FlickeringLight*. Este nodo, hijo de *ProjectileGrantingItem*, utiliza animaciones de parámetros de *shader* para crear un resplandor dinámico sobre el *item*. El efecto es configurable desde el editor, permitiendo ajustar color, intensidad, radio y velocidad del parpadeo. La animación se inicia automáticamente al cargar el *item* y se detiene al recogerlo, ocultando la luz y el *sprite* del pro-

yectil mediante el método *OnActivationTriggerActivated()* y mostrando unas partículas. Esto proporciona una retroalimentación visual clara, indicando que el *item* ha sido recogido y la mejora otorgada.

La integración de *FlickeringLight* se realiza de forma modular, permitiendo reutilizar el efecto en otros elementos del escenario si se desea. El sistema aprovecha el animador de *sprites* (*SpriteAnimator*) y el shader de resplandor descrito en la sección 4.5.2.

En resumen, el *item* gestiona únicamente la parte visual, ocultando sus *sprites* al ser recogido y delegando el resto de la lógica a los *triggers*.

Para más información sobre los *items*, consultar el GDD (sección B, sección 9.6).

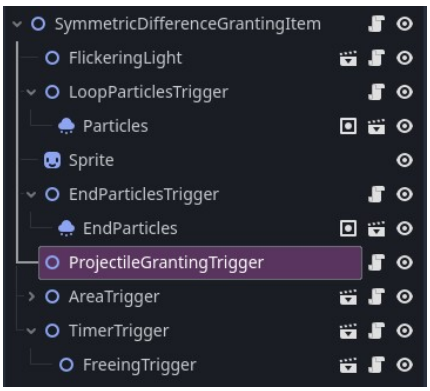


Figura 133: Árbol de nodos de un *ProjectileGrantingItem* mostrando la composición de *triggers*.

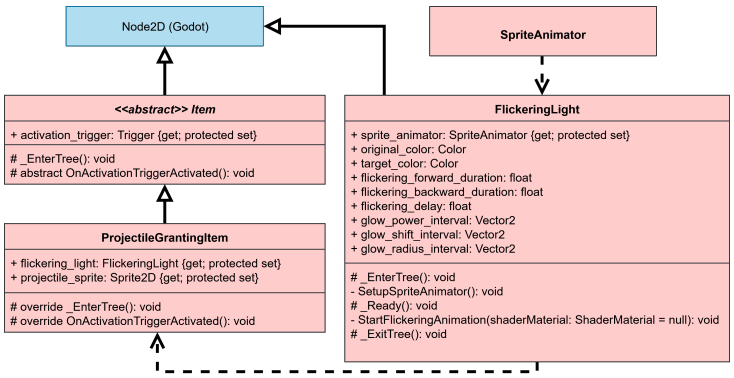


Figura 134: Diagrama de clase de *Item* y sus derivados.

4.23.3. *Spawner* del jugador

El *Spawner* del jugador es el elemento encargado de gestionar la animación de aparición del jugador al inicio de cada nivel. Su funcionamiento se basa en una composición de efectos

visuales y sonoros que refuerzan la entrada del jugador en el escenario, aportando identidad y dinamismo al comienzo de la partida.

La escena del *Spawner* está formada por varios nodos: partículas en espiral (*SpiralParticles*), partículas de explosión (*ExplosionParticles*), un animador de destello (*SparkAnimator*) y un gestor de audio (*SFXManager*). Al iniciar la animación, el sistema realiza una secuencia sincronizada: primero muestra el destello del *spark* mediante una animación de transparencia, reproduce el sonido de carga, activa las partículas en espiral y anima el brillo del destello con el *shader* de resplandor (*Glow*). Tras completar la animación, se detienen las partículas y el sonido, y se emite una señal para indicar que la aparición ha finalizado.

Al finalizar la aparición, el *Spawner* ejecuta una animación de cierre: reproduce el sonido de explosión, muestra el destello invertido y activa las partículas de explosión, reforzando visualmente la aparición del jugador.

La lógica de animación se implementa en la clase *PlayerSpawner*, que hereda de *Spiral* y coordina los distintos nodos y señales. El uso de corutinas (*await ToSignal*) permite sincronizar los efectos visuales y sonoros de forma precisa, asegurando una experiencia fluida y atractiva.

Este sistema aporta una transición visual clara al inicio de la partida, mejorando la inmersión y la percepción de calidad del juego.

Para más detalles sobre el *Spawner* del jugador, consultar el GDD (sección B, sección 9.7.1).

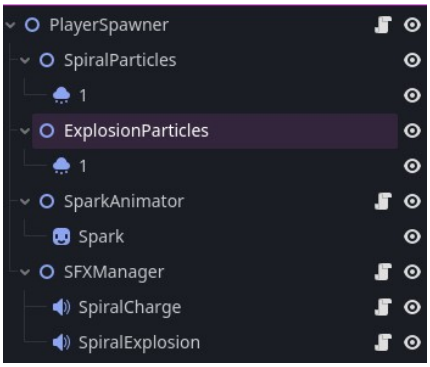


Figura 135: Árbol de nodos del *PlayerSpawner*.

4.23.4. Portal de fin del nivel

El portal de fin del nivel (*EndSpiral*) es el elemento que marca la finalización de la partida y permite al jugador completar el nivel. Su diseño combina efectos visuales, sonoros y lógica de

activación mediante *triggers*, creando una experiencia de cierre memorable y coherente con la estética del juego.

La escena del portal está compuesta por partículas en espiral (*SpiralParticles*), partículas de explosión (*ExplosionParticles*), un animador de destello (*SparkAnimator*), un gestor de audio (*SFXManager*) y un *AreaTrigger* circular que detecta la entrada del jugador. Al aparecer en el nivel, el portal inicia una animación de destello y explosión, seguida de un bucle de animaciones de resplandor que mantienen el efecto visual activo.

Cuando el jugador entra en el área del portal, el *AreaTrigger* activa el sistema, deteniendo las partículas en espiral, reproduciendo el sonido de explosión y mostrando la animación de cierre. Además, se activa el *LevelCompleterTrigger*, que finaliza el nivel y puede centrar la cámara en el portal durante la transición.

La lógica de animación y activación se implementa en la clase *EndSpiral*, que hereda de *Spiral* y gestiona la secuencia de efectos y señales. El sistema utiliza corutinas para sincronizar las animaciones y permite configurar los parámetros visuales y de sonido desde el editor.

Este portal aporta una retroalimentación visual y sonora clara al completar el nivel, reforzando la sensación de logro y cierre para el jugador.

Para más detalles sobre el portal de fin del nivel, consultar el GDD (sección B, sección 9.7.2).

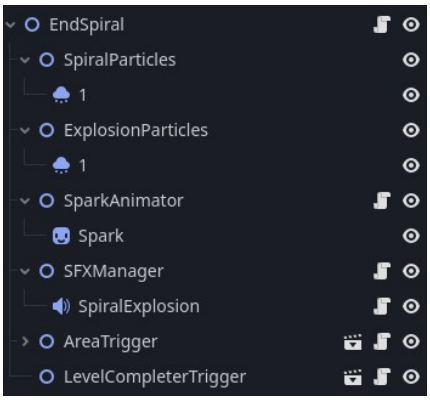


Figura 136: Árbol de nodos del *EndSpiral*.

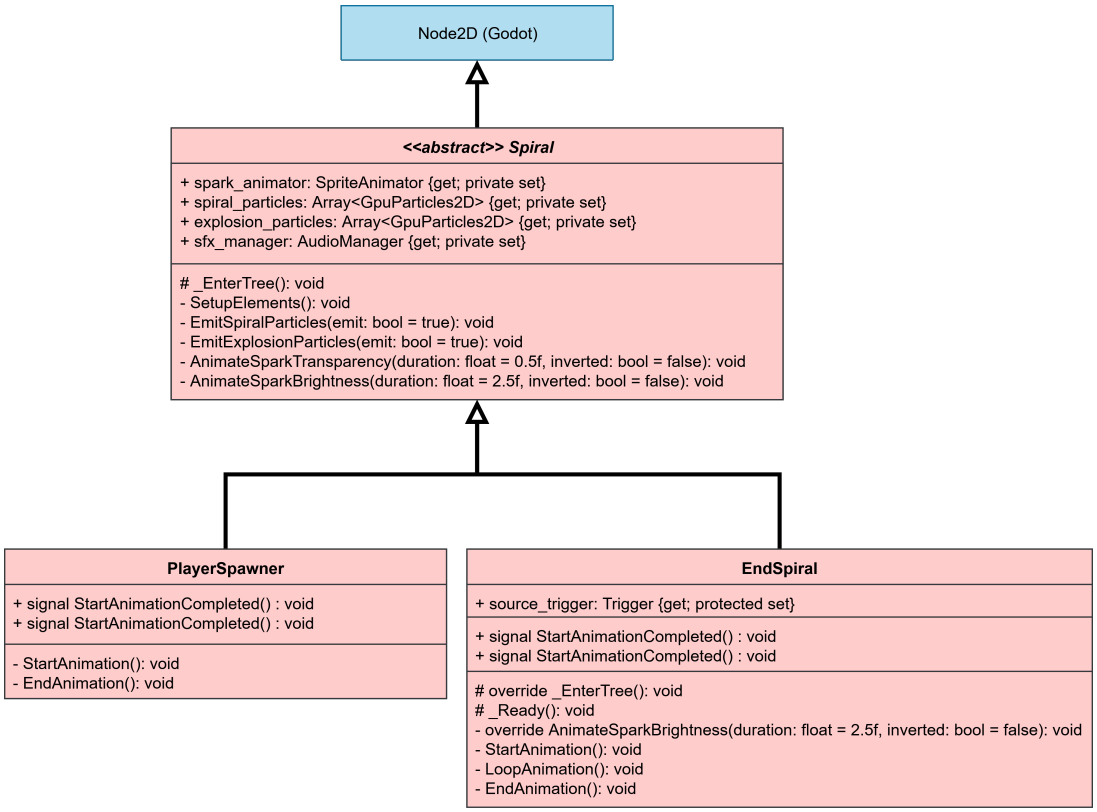


Figura 137: Representación UML de la clase *Spiral* y sus derivados.

5

Pruebas y Correcciones

En la fase final del desarrollo de *Set Wars* se realizaron pruebas con usuarios con el objetivo de evaluar la calidad de la experiencia de juego y obtener retroalimentación que permitiera identificar posibles mejoras. Este tipo de pruebas es fundamental en el desarrollo de videojuegos, ya que permite contrastar el diseño teórico con la experiencia real de los jugadores, valorando aspectos como la jugabilidad, la claridad de las mecánicas y el nivel de disfrute durante la sesión.

Se optó por realizar las pruebas en un entorno reducido, con un número limitado de participantes (dos *testers*), suficiente para obtener observaciones cualitativas valiosas. Es importante señalar que todo el trabajo de diseño y preparación de las pruebas de experiencia de usuario ya está implementado en este proyecto, lo que permitirá en el futuro ampliar el número de participantes y realizar un análisis más robusto y representativo (se ha añadido como línea futura este aspecto en la sección 6.2).

Para complementar estas impresiones, se empleó un cuestionario estandarizado que cuantifica la experiencia del jugador siguiendo métricas reconocidas.

Además de evaluar la experiencia mediante el cuestionario, los *testers* desempeñaron un papel fundamental en la detección de errores y en la identificación de posibles mejoras en el diseño. Durante las sesiones se prestó atención a fallos técnicos, problemas de usabilidad o comportamientos inesperados. Asimismo, se recogieron sugerencias directas de los participantes sobre aspectos que podrían ajustarse o modificarse. Esta retroalimentación cualitativa complementó los datos obtenidos con los cuestionarios, ofreciendo una visión más completa sobre el estado del prototipo y las áreas a mejorar.

5.1. Respuestas al cuestionario

Cada participante completó una sesión de juego en la que se les permitió interactuar libremente con el prototipo, explorando sus mecánicas y resolviendo los desafíos que se presentaban.

Al finalizar la sesión de juego, se aplicó el Game Experience Questionnaire (GEQ), *Core Module*. Cada dimensión del cuestionario se evalúa mediante una serie de afirmaciones (5 por dimensión) que los participantes deben valorar según su grado de acuerdo durante la sesión de juego.

Para la medición se utilizó una *escala Likert* de 1 a 5, donde:

Valor	**Descripción**
1	totalmente en desacuerdo / nada de acuerdo
2	en desacuerdo / poco acuerdo
3	neutral / moderadamente de acuerdo
4	de acuerdo / bastante de acuerdo
5	totalmente de acuerdo / completamente de acuerdo

Figura 138: Escala de valoración para el cuestionario GEQ.

Y las afirmaciones y valoraciones son las siguientes por cada dimensión:

Pregunta	**Tester 1**	**Tester 2**
Sentí que podía controlar y dominar el juego.	5	5
Me sentí capaz de superar los retos del juego.	4	4
Disfruté sintiéndome hábil mientras jugaba.	5	5
Considero que tomé buenas decisiones durante la partida.	5	5
Sentí que progresaba a medida que jugaba.	5	5

Figura 139: GEQ - Competence: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Me sentí inmerso en el mundo del juego.	5	3
Me pareció que la historia o el entorno era convincente.	5	5
Olvidé por momentos el mundo real mientras jugaba.	5	3
Los elementos visuales y sonoros reforzaron la experiencia.	5	5
Me sentí parte del entorno del juego.	5	3

Figura 140: GEQ - Sensory & Imaginative Immersion: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Estaba totalmente concentrado en la partida.	5	5
Perdí la noción del tiempo mientras jugaba.	4	5
Me resultó fácil mantenerme enfocado en el juego.	5	5
No me distraje con elementos externos mientras jugaba.	5	5
Las tareas del juego fluían de manera natural.	5	5

Figura 141: GEQ - Flow: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Me sentí frustrado durante la partida.	2	4
Algunos momentos me resultaron molestos o incómodos.	1	3
Me irrité con las mecánicas o retos del juego.	4	2
Hubo momentos en los que sentí ansiedad o estrés.	1	1
Algunas dificultades me parecieron injustas.	1	4

Figura 142: GEQ - Tension / Annoyance: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Los retos me parecieron apropiadamente difíciles.	5	4
Me sentí retado, pero no abrumado.	5	4
La dificultad era equilibrada y estimulante.	5	4
Tuve que esforzarme para superar los retos.	4	5
Me motivó la dificultad a seguir jugando.	5	5

Figura 143: GEQ - Challenge: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Me sentí contento o satisfecho jugando.	5	5
Disfruté de la experiencia de juego.	5	5
Me sentí motivado para seguir jugando.	5	5
La experiencia me resultó divertida.	5	5
Sentí emociones positivas durante la partida.	5	5

Figura 144: GEQ - Positive Affect: afirmaciones valoradas por los *testers*.

Pregunta	Tester 1	Tester 2
Sentí ansiedad o incomodidad jugando.	1	1
Me sentí decepcionado o desanimado en algún momento.	1	1
Experimenté emociones negativas durante la partida.	2	1
Me molestaron algunos elementos del juego.	4	1
Hubo momentos de frustración intensa.	1	2

Figura 145: GEQ - Negative Affect: afirmaciones valoradas por los *testers*.

5.2. Retroalimentación y mejoras

Durante las sesiones de prueba, los *testers* no solo evaluaron la experiencia de juego en términos generales, sino que también señalaron diversos aspectos susceptibles de mejora o que merecían especial atención para pulir la experiencia final. Estos comentarios resultaron esenciales para detectar detalles que, de otro modo, podrían haber pasado desapercibidos y constituyen una base valiosa para la iteración del diseño.

Los principales comentarios recibidos fueron los siguientes:

- Se detectó un fallo técnico en el que, al mover la cámara, la mirilla de apuntado y algunos objetos en movimiento titileaban en pantalla. Este problema estaba relacionado con la sincronización entre el sistema de cámara y los objetos en cada *frame*, y fue corregido ajustando la lógica en Godot. Además, se mejoró el control de la mirilla con ratón, pues con mando no se presentaba este error.

- Un tester valoró muy positivamente la experiencia al no incluir flashes ni elementos que pudieran afectar a su sensibilidad visual, dado que era epiléptico. La paleta en blanco y negro, junto con el uso cromático diferenciado para corazones y proyectiles, favoreció la comprensión de la información esencial en pantalla, algo especialmente relevante en un juego frenético.

- Uno de los *testers* detectó que en el nivel 4 era posible saltarse una sección. Este problema se solucionó de manera sencilla modificando ligeramente la estructura de esa parte del escenario.

- Se apreció que los diálogos pudieran avanzarse manualmente mediante una tecla, lo que permitió a cada jugador adaptar el ritmo de lectura a sus propias necesidades sin sentirse presionado.

- Resultó útil que la salud del jugador se mostrase en pequeño junto a la nave, además de en el marcador principal, ya que esto evitaba desviar la mirada hacia un extremo de la pantalla en momentos de alta tensión.

- Se destacó la posibilidad de ajustar la sensibilidad de la mirilla al jugar con mando, lo que redujo el riesgo de mareo y proporcionó un control más cómodo y preciso.

- Se señaló que la curva de dificultad era adecuada en general, salvo en el nivel 5, que resultaba excesivamente complicado en comparación con los anteriores. Tras los ajustes pertinentes, el nivel mantiene un reto elevado, pero equilibrado con la progresión establecida en los cuatro niveles previos.

- La música y la ambientación fueron valoradas como factores que reforzaban la inmersión. El diseño de la dificultad progresiva en cada nivel también contribuyó a mantener esta sensación de concentración.

- Finalmente, se apreció que la existencia de tipos de proyectiles y diferentes colores de corazones aportaba variedad y dinamismo, así como un componente estratégico adicional al poder recuperar salud en determinadas circunstancias.

5.3. Análisis de resultados y conclusiones

Una vez completadas las sesiones de prueba, se procedió al análisis de los datos recogidos mediante el cuestionario GEQ. Dado que participaron dos *testers*, los resultados se evaluaron principalmente de forma cualitativa, observando las tendencias y puntuaciones de cada dimensión, aunque se presentan también los valores numéricos para cada participante con el objetivo de identificar patrones y áreas de mejora.

El análisis se realizó siguiendo estos pasos:

1. Se recopilaron todas las respuestas de los *testers* en las tablas correspondientes a cada dimensión del GEQ.

2. Se calculó, para cada dimensión, el valor promedio de los dos participantes, con el fin de obtener un perfil general de la experiencia de juego.

3. Se identificaron los aspectos mejor valorados, así como aquellos que recibieron puntuaciones más bajas, para localizar posibles puntos de frustración o dificultades en el juego.

4. Se complementó esta información con la retroalimentación obtenida por los *testers* como justificación de algunas respuestas o porque se les preguntó aspectos importantes a destacar por su parte, especialmente en relación a errores técnicos, problemas de usabilidad o sugerencias de mejora.

A partir de este análisis, se pudo extraer un perfil detallado de la experiencia de los jugadores:

- **Competence:** los *testers* indicaron sentirse capaces de controlar el juego y progresar en los retos, con puntuaciones altas en esta dimensión.
- **Sensory & Imaginative Immersion:** la inmersión sensorial y narrativa fue valorada positivamente, aunque se sugirieron mejoras en algunos elementos visuales para reforzar la experiencia.
- **Flow:** los participantes lograron concentrarse y mantener la atención durante la partida.
- **Tension / Annoyance:** se identificaron algunos momentos de frustración vinculados a la dificultad puntual de ciertas mecánicas.
- **Challenge:** la dificultad percibida fue adecuada, generando motivación sin resultar excesivamente complicada.
- **Positive Affect:** la experiencia global fue divertida y motivadora, con emociones positivas predominantes.
- **Negative Affect:** se observaron puntuaciones bajas en esta dimensión, lo que indica que las emociones negativas fueron mínimas y controladas salvo excepciones puntuales.

En conclusión, los resultados del GEQ y la retroalimentación cualitativa permitieron validar el diseño de *Set Wars* en términos de jugabilidad, inmersión y disfrute, al tiempo que identificaron áreas específicas que podrían beneficiarse de ajustes o mejoras adicionales. Este proceso confirma la utilidad del GEQ como herramienta estandarizada para evaluar la experiencia del jugador y apoyar decisiones de diseño basadas en evidencia.

6

Conclusiones y Líneas Futuras

En este apartado se recogen las conclusiones generales derivadas del desarrollo del proyecto, poniendo en perspectiva los logros alcanzados, las principales dificultades superadas y el aprendizaje personal adquirido durante el proceso. Asimismo, se plantean posibles líneas de mejora y ampliación que podrían orientar el futuro crecimiento del juego.

6.1. Conclusiones generales

A pesar de que la idea original del proyecto contemplaba un juego más ambicioso, capaz de enseñar una mayor variedad de conceptos matemáticos, la decisión de reducir el alcance a un proyecto más acotado pero pulido y centrado resultó ser una elección acertada. Esta reducción permitió concentrarse en los aspectos esenciales del juego, garantizando un producto final acabado, coherente y de calidad.

Se han cumplido los objetivos principales definidos en el anteproyecto. Aunque no se han implementado algunas de las propuestas indicadas como posibles (por ejemplo, el modo infinito), el resultado final supera las expectativas iniciales en cuanto a experiencia de usuario y calidad de la jugabilidad. Se ha logrado dar brillo y profundidad al juego, lo que ha requerido un esfuerzo significativo tanto en el diseño como en el desarrollo, no solo como producto jugable sino también en la implementación técnica y programación de todos sus sistemas. El proyecto destaca por su acabado profesional y por ofrecer una experiencia envolvente, divertida y frenética, reflejando plenamente la idea original y el trabajo realizado.

Es importante destacar que, aunque anteriormente he desarrollado otros juegos, todos ellos fueron realizados en equipo y en Unity, mientras que los proyectos en Godot hasta la fecha se limitaron a bocetos o pruebas. Este proyecto constituye mi primer juego completo desarrollado

de manera individual en Godot, con una idea única concebida y ejecutada por mí, lo que ha supuesto un reto significativo y una experiencia de aprendizaje muy valiosa.

Asimismo, todo el diseño artístico del juego (salvo mínimas excepciones) ha sido realizado por mí, principalmente con Krita. De igual manera, he prestado especial atención a la selección de recursos externos, procurando utilizar únicamente aquellos con licencias libres o que requieran únicamente atribución (*CC0* y *CC-BY 4.0*), incluyendo audio, música, tipografías, etc. Este cuidado garantiza que el proyecto sea completamente publicable y utilizable, respetando la propiedad intelectual y asegurando que todo el contenido con licencia pueda ser utilizado legalmente.

Finalmente, cabe resaltar que la aplicación de metodologías y técnicas aprendidas durante la carrera (como *Scrum*, bocetos, diagramas de flujo y planificación de tareas) ha contribuido de manera significativa a la organización y ejecución del proyecto. Estas herramientas permitieron estructurar el desarrollo, optimizar el tiempo y mejorar la calidad tanto del diseño como de la implementación final.

6.2. Líneas futuras

A pesar de que el proyecto actual ha logrado cumplir con sus objetivos principales, existen diversas oportunidades para ampliar y mejorar la experiencia de juego. Las siguientes propuestas representan posibles mejoras, expansiones o nuevas funcionalidades que podrían incorporarse en versiones futuras:

- **Modo infinito:** Este modo permitiría que los jugadores se enfrenten a desafíos continuos sin límite de niveles, aumentando la rejugabilidad mediante sistemas de puntuación y récords. Su inclusión reforzaría la competitividad y prolongaría la experiencia del juego, convirtiéndose en una característica central para los usuarios más habituales.

- **Más niveles y contenido adicional:** Ampliar el número de niveles más allá de los cinco actuales, introduciendo escenarios variados, nuevos enemigos, jefes, mecánicas de juego adicionales y sistemas de progresión más complejos. Esto enriquecería la experiencia, diversificaría los retos y permitiría aprender conceptos matemáticos de forma más amplia y profunda.

- **Nuevas mecánicas:** Incorporar conceptos matemáticos más avanzados, como conjuntos con elementos repetidos, también conocidos como *multiconjuntos* o *bags* [23], y productos cartesianos, para ofrecer dinámicas de juego más variadas y educativas. Esto abriría la posibilidad de desarrollar mecánicas estratégicas más complejas y originales.

- **Restricción de enemigos dentro del mapa:** se planteó implementar que los enemigos no pudieran atravesar los límites del escenario, dado que en la lógica del juego todo lo que queda fuera de ellos está condenado a desaparecer. Sin embargo, esta mecánica resultó compleja de integrar en el comportamiento de los enemigos, consumiendo más tiempo del previsto y sin alcanzar un resultado satisfactorio. Aun así, el hecho de que los enemigos puedan traspasar los límites no afecta de forma negativa a la jugabilidad, manteniéndose la experiencia igualmente disfrutable.

- **Mayor número de *testers*:** Como prioridad en futuras versiones, se realizará un testeo con un mayor número de usuarios para obtener una validación más significativa sobre la experiencia de juego, ya que está implementado todo el trabajo de diseño y preparación de las pruebas de experiencia de usuario.

- **Mayor soporte de idiomas:** Expandir el juego para incluir más idiomas, facilitando el acceso a un público internacional y mejorando la experiencia de usuarios.

- **Mayor accesibilidad:** Implementar opciones para mejorar la accesibilidad visual, como paletas de colores adaptadas para daltonismo y otras variaciones, garantizando que el juego sea disfrutable para un público más amplio.

- **Más ajustes y personalización:** Añadir opciones técnicas y de interfaz adicionales, como la visualización de *FPS*, control de *V-Sync*, configuración de gráficos y personalización de controles, ofreciendo un mayor nivel de control sobre la experiencia de juego.

- **Modo editor de niveles:** Posibilitar que los jugadores diseñen y compartan sus propios niveles, fomentando la creatividad y la rejugabilidad dentro de la comunidad.

- **Integración de música personalizada:** Permitir que los usuarios puedan añadir canciones propias para jugar niveles personalizados, al estilo de juegos como *Beat Hazard Ultra*, incrementando la inmersión y la interacción personal con el juego.

6.3. Aprendizaje personal adquirido

La realización de este proyecto ha supuesto un aprendizaje tanto técnico como personal. A continuación se resumen los aspectos más relevantes:

- **Aprendizajes técnicos:** profundización en la arquitectura por escenas y nodos, uso de *singletons* y módulos, comunicación entre sistemas, resolución de problemas prácticos (abstracción de *inputs*, sincronización por *frame*, gestión de estados globales), así como diseño accesible y coherencia audiovisual.

- **Gestión del trabajo y organización:** práctica de metodologías de desarrollo iterativo, dividiendo el proyecto en tareas asumibles y estableciendo prioridades. Esto permitió mantener el control del alcance y avanzar con constancia, reduciendo la complejidad de los problemas a resolver y adaptándose a los cambios repentinos.

- **Dirección creativa y diseño:** exploración de un estilo visual y sonoro propio, equilibrando referencias arcade clásicas con una estética más abstracta y matemática. Este proceso ayudó a reforzar la importancia de la coherencia artística como parte integral de la experiencia de juego. Buscando tener un estilo definido, se optó por un diseño minimalista y claro, evitando la sobrecarga visual y facilitando la comprensión de las mecánicas. Pero a la vez, se intentó dotar al juego de personalidad y carácter mediante el uso de diseños únicos, animaciones fluidas y efectos visuales atractivos.

- **Reflexión y proyección personal:** consolidación de la confianza en la capacidad de construir un proyecto completo de forma individual, afianzando el criterio para tomar decisiones de calidad y alcance. Este proceso cierra una etapa formativa en la universidad y, al mismo tiempo, marca una dirección clara hacia el desarrollo profesional en el ámbito de los videojuegos.

Referencias

[1] Paul R. Halmos. *Naive Set Theory*. Springer, 1960.

[2] Thomas Jech. *Set Theory*. 3rd. Springer, 2002.

[3] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 7th. McGraw-Hill, 2012.

[4] Joseph W. Dauben. *Georg Cantor: His Mathematics and Philosophy of the Infinite*. Princeton University Press, 1990.

[5] John Venn. «On the Diagrammatic and Mechanical Representation of Propositions and Reasonings». En: *Philosophical Magazine* 10 (1880), págs. 1-18.

[6] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. 5th. Addison Wesley, 2003.

[7] Godot Engine. *Godot Documentation*. Último acceso: agosto de 2025. 2014. URL: https://docs.godotengine.org/.

[8] David R. Michael y Sande Chen. *Serious Games: Games That Educate, Train, and Inform*. Thomson Course Technology, 2006.

[9] Clark C. Abt. *Serious Games*. Viking Press, 1970.

[10] Ben Sawyer y Peter Smith. «Serious Games: Improving Public Policy through Game-based Learning and Simulation». En: *Woodrow Wilson International Center for Scholars*. 2002.

[11] Elizabeth A. Boyle et al. «An update to the systematic literature review of empirical evidence of the impacts and outcomes of computer games and serious games». En: *Computers & Education* 94 (2016), págs. 178-192. DOI: 10.1016/j.compedu.2015.11.003.

[12] OECD. *PISA 2022 Results: The State of Learning and Equity in Education*. Inf. téc. Paris: Organisation for Economic Co-operation y Development, 2023. URL: https://www.oecd.org/pisa/publications/pisa-2022-results.htm.

[13] Ken Schwaber y Jeff Sutherland. *The Scrum Guide: The Definitive Guide to Scrum*. Scrum.org, 2020.

[14] Jeff Sutherland. *Scrum: The Art of Doing Twice the Work in Half the Time*. New York: Crown Business, 2014.

[15] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Upper Saddle River, NJ: Addison-Wesley, 2012.

[16] W. A. IJsselsteijn, Y. A. W. de Kort y K. Poels. *The Game Experience Questionnaire*. Inf. téc. Technische Universiteit Eindhoven, 2013.

[17] Grady Booch et al. *Object-oriented analysis and design with applications*. Addison-Wesley Professional, 2007.

[18] Chris Gane y Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, 1979.

[19] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2003.

[20] Wikipedia. *Electronic music*. Último acceso: agosto de 2025. 2025. URL: https://en.wikipedia.org/wiki/Electronic_music.

[21] Wikipedia. *Synthwave*. Último acceso: agosto de 2025. 2025. URL: https://en.wikipedia.org/wiki/Synthwave.

[22] Wikipedia. *Drone music*. Último acceso: agosto de 2025. 2025. URL: https://en.wikipedia.org/wiki/Drone_music.

[23] Wikipedia. *Multiset*. Último acceso: agosto de 2025. 2025. URL: https://en.wikipedia.org/wiki/Multiset.

Apéndice A
High Concept Document

High Concept

Set Wars: Cardinal Bounds

Escrito por Mario Cortés Herrera

Versión del Documento: 1.0

Motivación

Mi idea nace de querer enseñar de forma indirecta al jugador, donde no se sienta presionado y pueda disfrutar de su progreso. Este pensamiento viene dado a que en la actualidad, siendo cada vez más jóvenes, nos sentimos frustrados por no poder aprender de forma rápida y útil. Este juego quiero que sirva de ejemplo de cómo jugando el aprendizaje es más efectivo e incluso interesante y atractivo.

En este caso, voy a enfocarme en querer enseñar ciertos aspectos matemáticos. En un principio tengo pensado enfocarlo al aprendizaje de Teoría de Conjuntos, pero esto podría expandirse a otras áreas.

Descripción Breve y Género

Consiste en un juego de naves y disparos en el espacio, el jugador tendrá que controlar una nave la cuál será capaz de moverse por el escenario, disparar (o posiblemente otras habilidades), para hacer frente a diferentes obstáculos y amenazas, las cuáles podrán destruirla.

Combina acción frenética y conceptos de teoría de conjuntos. Deberás gestionar los elementos de la nave como conjuntos matemáticos, enfrentando a los enemigos con operaciones como unión, intersección y diferencia para ganar avanzar.

Sus **dimensiones son 2D con perspectiva top-down (cenital).**

Título del Juego

Set Wars: Cardinal Bounds

Set Wars asienta la idea de que son guerras de conjuntos, como el tema principal que se intenta enseñar.

Cardinal Bounds queda como subtítulo sobre “Poner límites a los cardinales”, debido a que el juego trata de eliminar a los enemigos y la cardinalidad en los conjuntos juega un papel importante.

Tagline (o eslogan)

"Un universo donde las naves son conjuntos y sus elementos definen sus límites."

Público objetivo

El perfil va dirigido a personas que quieran meterse en el mundo de la Teoría de Conjuntos o que tengan ya cierta experiencia y quieran aprender sobre el tema mientras se divierten con un juego desafiante.

Mecánicas Principales

La jugabilidad busca tener estilo propio de que tanto la nave del jugador como los enemigos sean/posean conjuntos matemáticos; donde los elementos que tengan en su conjunto tengan un valor.

Algunos elementos pueden ser una representación de “salud” de un enemigo/jugador, la cual si se pierde totalmente lo elimina del escenario.

También los elementos pueden representar ciertos tipos de características que se poseen, como ataques y movimientos, o también según la cardinalidad del conjunto podrían representar el tamaño de los enemigos o cierto aspecto importante a tomar en cuenta.

Referente a los ataques, se planea utilizar diferentes tipos de operaciones de conjuntos para combatir o realizar ciertas acciones, por ejemplo, el uso de la Intersección o la Diferencia podría servir como posible medio de ataque para hacer perder salud.

Se busca poder hacer niveles con final (objetivo mínimo) o incluso poder hacer un modo infinito en el que solo acaba cuando el jugador pierde.

Estilo Visual y Temática

La ambientación será basada en una mezcla de aquellos juegos espaciales del mismo género mezclado con una estética entre simplista pero elegante con el propósito de buscar el simbolismo matemático y dar esa sensación de abstracción.

Diferencias Clave

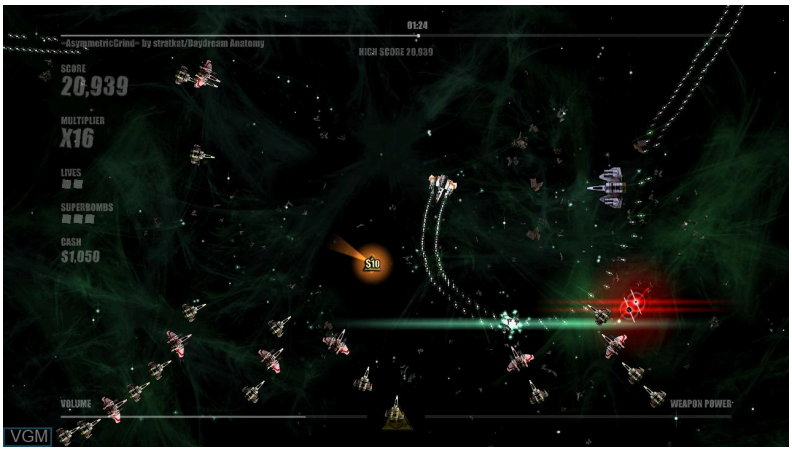
- Introduce conceptos de teoría de conjuntos en un contexto de acción rápida y frenética como lo es el shooter de naves.
- Combina aprendizaje matemático con mecánicas de juego adictivas.
- Posible narrativa referente a los conjuntos y el motivo de la guerra entre naves.

Plataforma

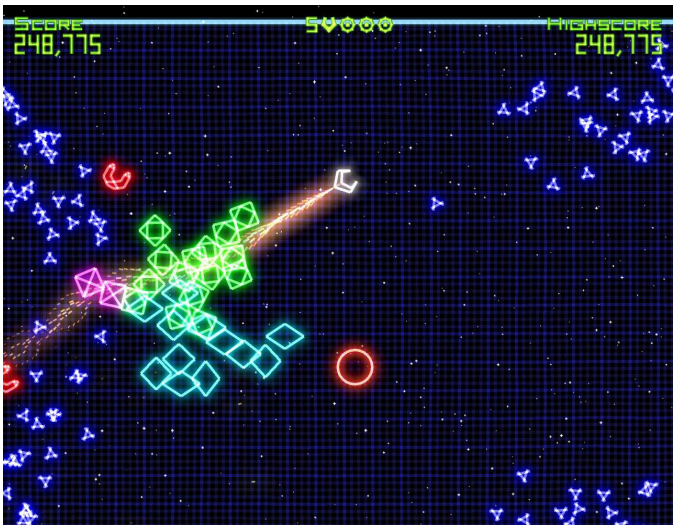
PC, principalmente control por teclado y ratón, pero también posiblemente con mando. Diseñado en Godot con C#.

Inspiración y Juegos Comparables

Beat Hazard Ultra



Geometry Wars: Retro Evolved 2



Apéndice B
Game Design Document

Game Design Document

Set Wars: Cardinal Bounds

Escrito por Mario Cortés Herrera

Versión del Documento: 1.0

Índice

1. Descripción general del juego..... 205
1.1 Título..... 205
1.2 Género..... 205
1.3 Plataformas..... 205
1.4 Herramientas..... 205
1.5 Idioma..... 205
2. Objetivo y Visión General..... 206
2.1 Público Objetivo..... 206
2.2 Estilo Visual..... 206
2.3 Inspiraciones..... 206
2.4 Diferencias Clave..... 209
3. Controles del Juego..... 210
3.1 Asignación de Controles..... 210
4. Pantalla de Inicio y Título..... 212
5. Interfaces..... 214
5.1 Botones..... 214
5.2 Otros componentes..... 216
5.3 Menús..... 216
5.3.1 Menú Principal..... 216
5.3.2 Menú de Selección de Niveles..... 218
5.3.3 Menú de Ajustes..... 220
5.3.4 Menú de Pausa..... 222
5.3.5 Menú de Fin del Juego..... 223
5.3.6 Menú de Nivel Completado..... 225
5.3.7 Créditos..... 226
5.4 Diagramas de Flujo..... 227
5.5 Pantalla de Carga..... 229
5.6 Pantalla de Arranque (Splash Boot)..... 230
6. Cámara del Juego..... 231
7. Sistema HUD..... 232
7.1 Indicador de Salud..... 232
7.1.1 Indicador de Salud Seguidor..... 232
7.2 Selector de proyectiles..... 233
7.3 Mira telescópica de Apuntado..... 234
7.4 Diálogo del Juego..... 234
7.5 Leyenda de Controles..... 235
8. Personaje del Jugador..... 236
8.1 Desplazamiento..... 236
8.2 Rotación..... 236
8.3 Disparo..... 237
8.4 Cambiar tipo y color de proyectil..... 237
9. Mecánicas del Juego..... 238
9.1 Vida..... 238

9.2 Daño....238
9.3 Muerte....239
9.4 Proyectiles....240
9.4.1 Tipos de Proyectiles....241
9.4.1.1 Diferencia....241
9.4.1.2 Unión....241
9.4.1.3 Intersección....242
9.4.1.4 Diferencia Simétrica....242
9.4.1.5 Complemento....243
9.5 Enemigos....244
9.5.1 Reglas Básicas de Comportamiento....244
9.5.2 Tipos de Enemigos....244
9.5.2.1 Asteroide....244
9.5.2.2 Enemigo Básico....245
9.5.2.3 Enemigo con Cañón Dual....246
9.5.2.3 Enemigo con Alternador de Fuego Cruzado....246
9.5.3 Diagrama de Flujo del Comportamiento....246
9.6 Items....247
9.6.1 Item de Nuevo Proyectil....248
9.6.2 Item de Nuevo Color para Proyectil....248
9.7 Elementos de Entorno....249
9.7.1 Portal de Spawn del Jugador....249
9.7.2 Portal de Fin de Nivel....250
9.7.3 Límites del Mapa....250
10. Niveles....252
10.1 Nivel 1: Origin of Difference....252
10.2 Nivel 2: Intersecting Lifeforms....253
10.3 Nivel 3: The Union of Power....253
10.4 Nivel 4: Complementary Decisions....254
10.5 Nivel 5: Asymmetric Rain....255
11. Sistema de Guardado....256

1. Descripción general del juego

Juego serio que busca enseñar y aplicar la teoría de conjuntos mediante mecánicas interactivas en un entorno lúdico. El jugador controla una nave espacial que representa un conjunto y debe interactuar con otros conjuntos (enemigos, beneficios, obstáculos) usando operaciones matemáticas para avanzar y sobrevivir.

1.1 Título

Set Wars: Cardinal Bounds

Set Wars es el título principal, y referencia que son “Guerras de Conjuntos”.

Cardinal Bounds es el subtítulo, y hace referencia a poner límites a los cardinales de los conjuntos porque una mecánica básica del juego es que las naves se destruyen si su conjunto está vacío (como si se quedaran sin vida).

1.2 Género

Shooter espacial en 2D con perspectiva top-down (cenital).

1.3 Plataformas

Principalmente **PC**, pero admite controles tanto de teclado con ratón como de mando (*joypad*).
Para Windows, Linux y se intentará que también macOS.

1.4 Herramientas

Se hará uso del motor de videojuegos **Godot** para la creación del videojuego, utilizando como lenguaje de programación en la parte de scripting **C#** y posiblemente se haga uso de **GDScript** en las partes donde se haga obligatorio por la herramienta (implementación de shaders por ejemplo).

También se hará uso de la herramienta de diseño artístico **Krita** para la creación y edición de la mayoría (o posiblemente todos) los elementos de arte del juego, es decir, los assets.

Como sistema de guardado e historial de cambios del proyecto se hará uso de un repositorio de **GitHub** donde se alojarán los archivos.

1.5 Idioma

El juego está disponible tanto en **inglés** como en **español**.

2. Objetivo y Visión General

El objetivo del juego es superar niveles mediante la aplicación de la teoría de conjuntos en combates frenéticos de disparos. Se busca fomentar el aprendizaje a través de la jugabilidad, debido a que las mecánicas del juego se forman utilizando diferentes características de esta rama matemática.

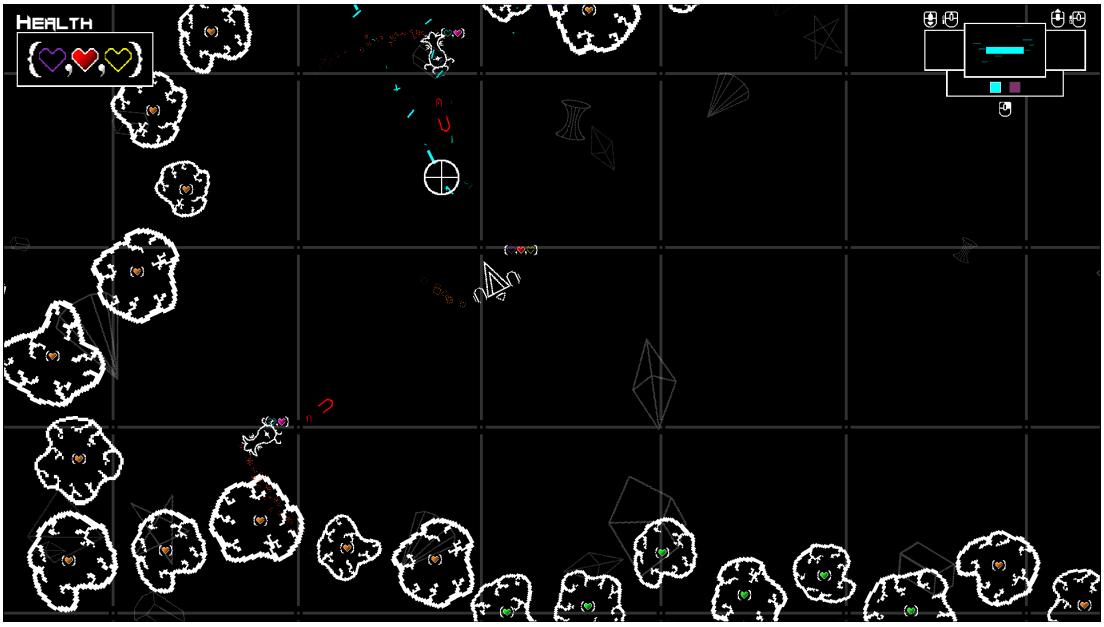
2.1 Público Objetivo

El perfil va dirigido a personas que quieran meterse en el mundo de la teoría de conjuntos o que tengan ya cierta experiencia y quieran aprender sobre el tema mientras se divierten con un juego desafiante.

2.2 Estilo Visual

La ambientación será basada en una mezcla de aquellos juegos espaciales del mismo género (retrofuturista) mezclado con una estética entre simplista pero elegante con el propósito de buscar el simbolismo matemático y dar esa sensación de abstracción.

Este es un ejemplo de una escena del juego para entender este estilo artístico:



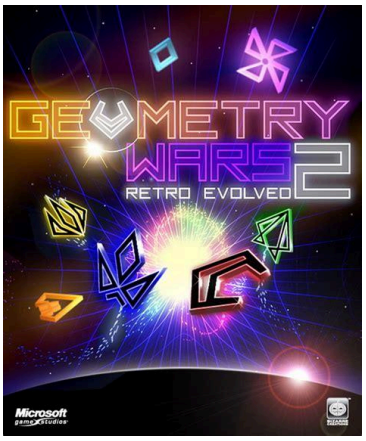
2.3 Inspiraciones

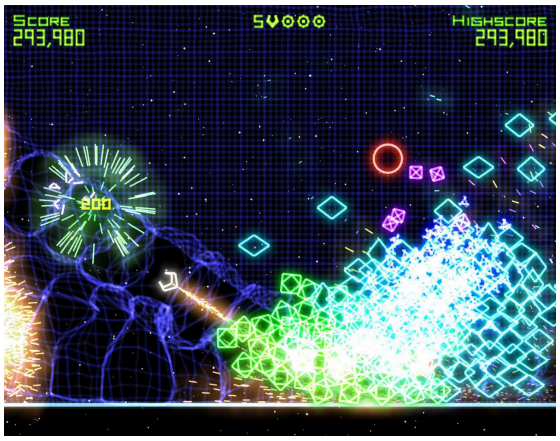
Las principales fuentes de inspiración de las que surge *Set Wars* son:

- **Beat Hazard Ultra (Cold Beam Games, 2011):** Este título se caracteriza por el uso de la música como elemento central en los combates espaciales, generando una experiencia dinámica y visualmente impactante. Su estilo cromático intenso y el diseño de los enemigos constituyen una influencia directa en *Set Wars*,

especialmente como referencia en el dinamismo audiovisual y la relación entre estímulo sonoro y acción jugable.

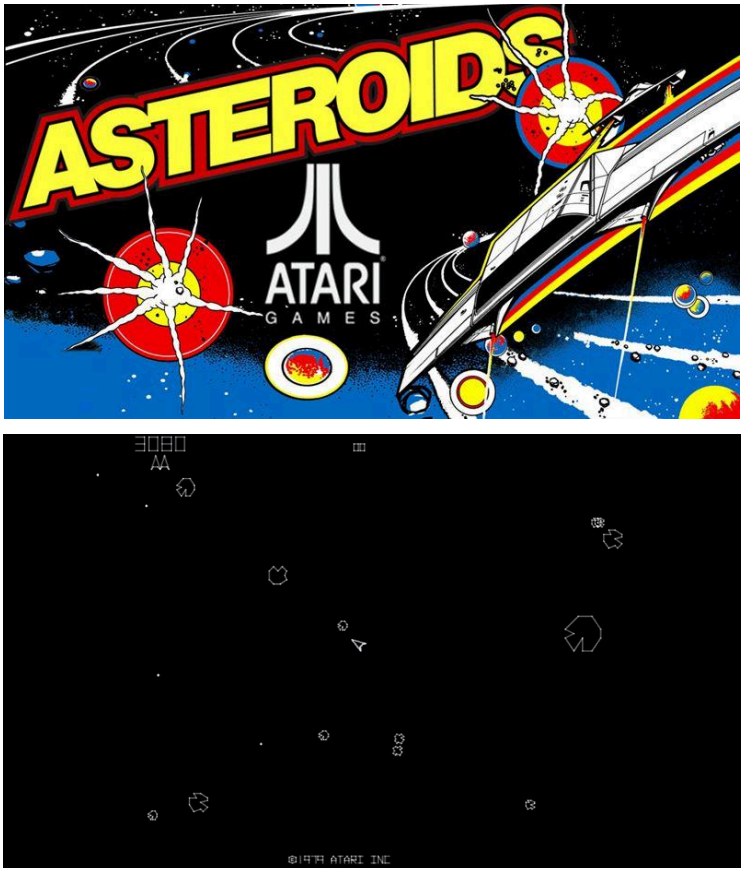
- **Geometry Wars: Retro Evolved 2 (Bizarre Creations, 2008):** El enfoque visual de este juego, basado en figuras geométricas y un minimalismo marcado, conecta de manera clara con la dimensión matemática de *Set Wars*. El uso de un fondo cuadriculado que remite a un plano cartesiano y la simplicidad de sus elementos gráficos han sido claves en la definición estética y conceptual del proyecto.



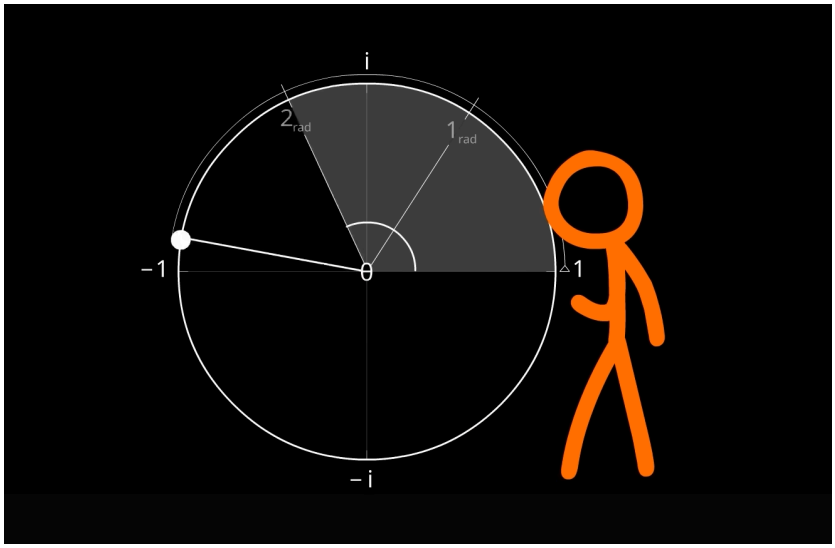


- **Asteroids (Atari, 1979):** Este juego es pionero en el género, debido a las limitaciones de la época sus colores destacan en negro y blanco, pero *Set Wars* se beneficia de esto porque en el juego lo único que tiene color son los corazones y

proyectiles. Los asteroides y la nave del jugador tienen cierta inspiración también de este juego aunque no fuese la principal.



- **"Animation vs Math" Vídeo de YouTube (Alan Becker, 2023):** Este recurso audiovisual constituye un ejemplo de cómo los conceptos matemáticos pueden trasladarse a representaciones interactivas y creativas. A diferencia de otros enfoques educativos más estáticos, esta propuesta demuestra la capacidad de las matemáticas para generar dinámicas de acción y combate, lo que refuerza la idea de aplicarlas en la jugabilidad y en las herramientas disponibles en Set Wars.



2.4 Diferencias Clave

- Introduce conceptos de teoría de conjuntos en un contexto de acción rápida y frenética como lo es el shooter de naves.

- Combina aprendizaje matemático con un diseño accesible y entretenido.

- Los niveles desarrollados constituyen la experiencia principal del juego. A diferencia de los *space shooters* tradicionales, centrados únicamente en el disparo constante, aquí cada nivel introduce mecánicas específicas y elementos de tipo puzzle. El jugador debe no solo enfrentarse a la acción, sino también aprender, decidir y aplicar lo que ha asimilado previamente, moviéndose por el escenario y resolviendo situaciones de forma estratégica.

3. Controles del Juego

Durante una partida, el jugador puede realizar las siguientes acciones principales con su nave:

- Desplazarse libremente en cualquier dirección (incluyendo diagonales).
- Rotar la nave sobre su eje para apuntar hacia cualquier ángulo mediante el uso de una mira telescópica.
- Aumentar la velocidad de desplazamiento de la nave.
- Disparar proyectiles en la dirección en la que apunta la nave.
- Cambiar entre diferentes tipos de proyectiles disponibles.
- Cambiar entre diferentes tipos de colores disponibles para los proyectiles.
- Avanzar en los diálogos.
- Pausar el juego / Reanudar el juego.
- Navegar por la interfaz.
- Pulsar un botón de la interfaz.

3.1 Asignación de Controles

El uso de teclado con ratón es la forma predeterminada del juego, esta es la lista de controles:

Acción	Asignación
Mover nave hacia arriba	**W**
Mover nave hacia abajo	**S**
Mover nave hacia la izquierda	**A**
Mover nave hacia la derecha	**D**
Mover mira telescópica	**Movimiento del ratón**
Rotación de la nave	**Automática hacia la mira telescópica**
Aumentar velocidad de desplazamiento (mantener)	**Shift**
Disparar	**Click izquierdo**
Cambiar tipo de proyectil (siguiente)	**Rueda del ratón hacia arriba**
	Botón lateral 1 (Botón 4)
Cambiar tipo de proyectil (anterior)	**Rueda del ratón hacia abajo**
	Botón lateral 2 (Botón 5)
Cambiar el color del proyectil (siguiente)	**Click derecho**

Avanzar diálogo	**Space**
Pausar el juego / Reanudar el juego	**Esc**
Navegar por la interfaz	**Movimiento del ratón**
Pulsar un botón de la interfaz	**Clic izquierdo**

También tienes la opción de jugar usando el mando (joypad), con la siguiente lista de controles:

Acción	Asignación (XBOX/PlayStation)
Mover nave	**Joystick izquierdo**
Mover mira telescópica	**Joystick derecho**
Rotación de la nave	**Automática hacia la mira telescópica**
Aumentar velocidad de desplazamiento (mantener)	**LT / L2**
	Pulsar Joystick izquierdo / L3
Disparar	**RT / R2**
Cambiar tipo de proyectil (siguiente)	**RB / R1**
Cambiar tipo de proyectil (anterior)	**LB / L1**
Cambiar el color del proyectil (siguiente)	**X /□**
Avanzar diálogo	**A / ×**
Pausar el juego / Reanudar el juego	**Joypad Button 6 / Start / Menu**
Navegar por la interfaz	**Joystick izquierdo**
	Cruceta
Pulsar un botón de la interfaz	**A / ×**

Nota: Todas las interfaces del juego hacen uso de los iconos del mando de Xbox para explicar los controles del juego. A pesar de ello, el juego es controlable con mando de PlayStation y de Nintendo Switch (según las configuraciones que se pueden hacer con Godot).

4. Pantalla de Inicio y Título

Al iniciar el juego se presentará la pantalla del **menú principal tras una animación** (animación que es la misma que luego se encuentra como transición en otras partes del juego y como pantalla de carga).

Desde un comienzo observamos como se presenta la fuente de letras que nos acompaña por todo el juego como parte de su identidad: **Motorblock**.

La elección estética de esta fuente viene de estar cuidando **la coherencia visual,** al ser una tipografía fuerte, geométrica y algo técnica.
Esto sugiere que el juego tiene un aspecto visual muy centrado en las matemáticas debido a sus formas rectas, gruesas y firmes, lo que proyecta cierta solidez, estructura y seriedad.
No quise optar por una fuente decorativa ni futurista en exceso, porque el juego no busca una estética exagerada, sino una más sobria y coherente.
A pesar de esta decisión, hay un único caso donde se ha optado por elegir una segunda fuente: **Crimson Pro**. La cuál se utiliza en los diálogos del juego para facilitar la lectura.

El fondo del menú principal no es simplemente decorativo, ya forma parte activa de la atmósfera del juego. Se compone de un **plano infinito** de **cuadrados** que evocan el espacio ilimitado de un **plano cartesiano**, reforzando visualmente la idea de un universo regido por reglas matemáticas. Este patrón no solo sitúa al jugador en un entorno abstracto pero estructurado, sino que conecta directamente con los fundamentos del juego, basados en lógica, conjuntos y relaciones espaciales.

Acompañando este plano, una animación constante de **partículas** tridimensionales recorre la pantalla. Estas partículas, con **formas geométricas variadas y movimientos impredecibles** (tanto en dirección como en rotación), aportan profundidad. Aparecen y

desaparecen de forma aleatoria, generando una sensación de flujo constante, como si el universo del juego estuviera vivo y siempre en movimiento.

Este fondo no solo se presenta en el menú principal, sino también durante el propio gameplay, manteniendo una **continuidad visual** que refuerza la estética general del juego. El fondo actúa como un puente entre la presentación y la experiencia jugable, envolviendo al jugador en el **mismo entorno visual** desde el primer momento.

A nivel sonoro, el menú principal reproduce la pieza musical *Dronescape_031* , una composición ambiental que refuerza aún más la atmósfera del juego. Se trata de un paisaje sonoro **etéreo y envolvente**, donde capas de sonido se entrelazan lentamente, como **si flotaran en un espacio sin gravedad**.
Esta elección musical no solo acompaña al fondo visual, sino que **invita a una pausa, a una contemplación**, como si el jugador estuviera asomándose a un universo infinito, abstracto y en constante movimiento.
No busca llamar la atención, sino **sumergir**. Genera una sensación de espacio amplio, sugerente y enigmático; *justo lo que busca transmitir este juego***,** *donde lo visual, lo espacial y lo lógico se entrelazan***.**

Dronescape_031 by jalastram (CC-BY 4.0 www.freesound.org/s/750853)

5. Interfaces

La interfaz del juego mantiene un estilo visual minimalista y formal, en armonía con la naturaleza abstracta de las matemáticas. El fondo de juego es predominantemente negro con tonos grises, mientras que los elementos de la interfaz (botones, textos, iconos) se presentan siempre en blanco puro, generando un contraste claro y elegante.

5.1 Botones

Los botones son la parte de las interfaces que el usuario puede interactuar. La mayoría presentan el color **blanco** como color primario, el **naranja tenue** como indicador de estar **seleccionado** y el **naranja puro** como indicador de que se está **pulsando** un botón.

Ejemplo de los tres colores que tiene un botón según cada estado.

Cuando se utiliza teclado y ratón, la navegación por las interfaces es haciendo uso del ratón para colocarlo encima de cualquier botón y el click izquierdo para poder pulsarlo.

Si se utiliza mando (joypad), la interfaz auto seleccionará un botón de ella por defecto (lo que se llama *grab focus*) y, mediante el uso del joystick izquierdo o la cruceta, se podrá navegar entre los botones, para después poder pulsarlos con el botón de acción (A / ✕).

Existen diferentes tipos de botones que el usuario puede manejar:

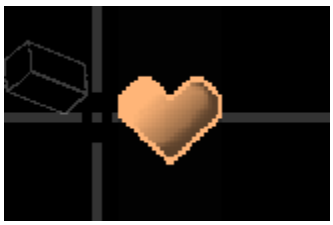
- **Botón básico:** Elemento interactivo más recurrente en el juego. Puede presentarse en diferentes formas: solo texto, solo icono o una combinación de ambos. Su función principal es ejecutar una acción cuando el jugador lo pulsa.

- **Slider:** Elemento presente en el menú de ajustes. Puede manipularse arrastrando con el ratón o, en el caso del mando, seleccionándolo y moviendo el stick o la cruceta en dirección vertical. Incluye un valor numérico en la parte superior que se actualiza dinámicamente y refleja el valor actual del control.

- **CheckButton:** Elemento pulsable similar al botón básico, pero acompañado de una casilla cuadrada que alterna entre dos estados (marcada o desmarcada). Se utiliza para activar o desactivar valores binarios dentro de los ajustes.

- **Selection:** Conjunto de botones agrupados bajo un texto descriptivo que indica la categoría de la elección. Al seleccionar un botón, este permanece resaltado en naranja para indicar su estado activo, mientras los demás se muestran en blanco. Solo puede estar seleccionado un botón del grupo en un momento dado indicando cual opción es la elegida.

- **Scrollbar:** Elemento presente en los créditos. Es una barra de desplazamiento vertical que permite navegar por el contenido. Puede manipularse arrastrando con el ratón, usando la rueda o, en el caso del mando, enfocando la barra y desplazándola de forma similar al slider. Su movimiento traslada el contenido de los créditos en el eje vertical. Visualmente es un corazón con un fondo negro muy transparente que, debido a como son los créditos, queda estéticamente bien.



5.2 Otros componentes

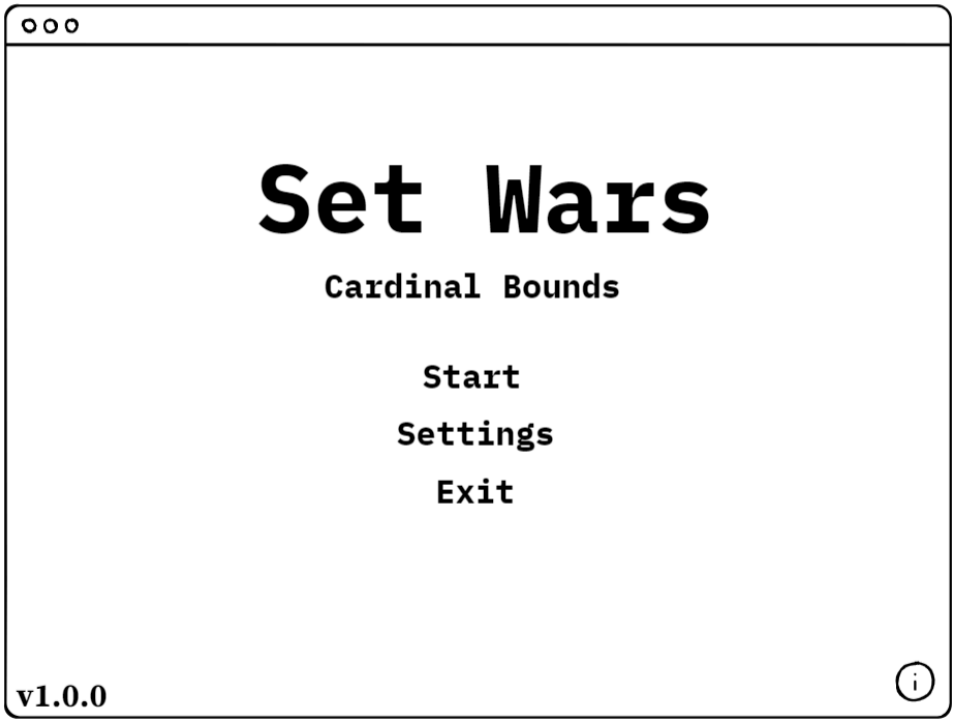
Otros componentes de las interfaces aparte de los botones son los *InputIcon,* estos son un contenedor formado por un texto y uno o dos iconos (según hace falta). El texto muestra una acción del juego que se puede realizar, mientras que los iconos muestran cuáles son los botones con los que se puede hacer. Según el actual tipo de entrada (mando o teclado/ratón) y el idioma, los iconos y el texto cambian dinámicamente. Los textos se pueden omitir y solo dejar los iconos, aspecto que se utiliza en el Selector de proyectiles (**7.2**).

El mismo InputIcon con los dos tipos de entradas.

5.3 Menús

Los menús del juego son el principal punto de interacción entre el jugador y las configuraciones o recursos disponibles. Cada menú está organizado de manera lógica, proporcionando únicamente la información o las opciones necesarias sin sobrecargar la experiencia visual.

5.3.1 Menú Principal



Boceto del Menú Principal.

Versión final del Menú Principal en Inglés.

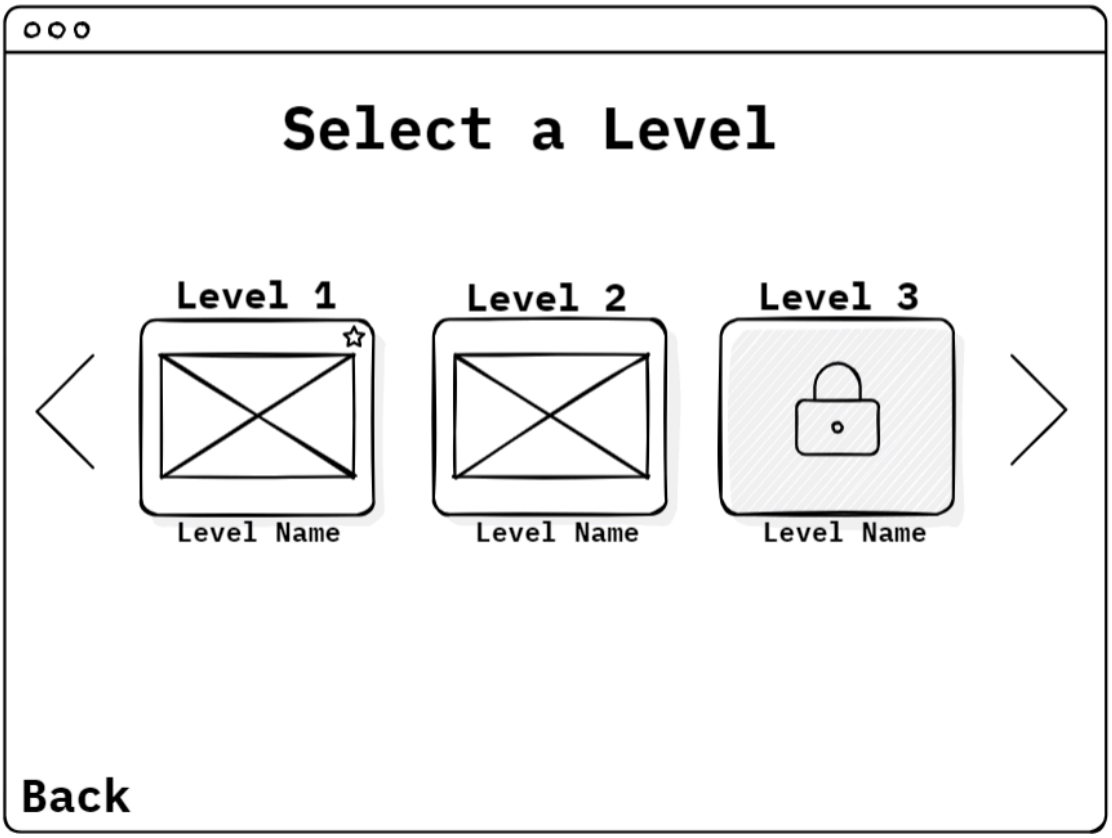
Versión final del Menú Principal en Español.

El menú principal presenta cuatros botones: Iniciar, Ajustes, Salir y los Créditos (la *i* presente abajo a la derecha):

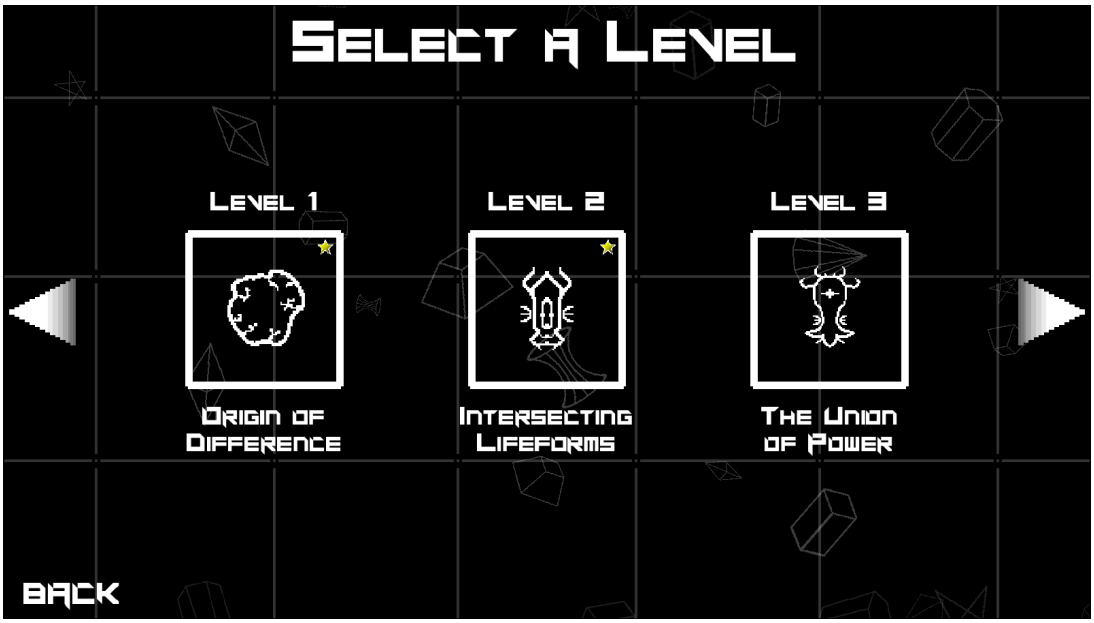
- Pulsar el botón de **Iniciar** hará que se navegue al menú de Selección de Nivel.
- Pulsar el botón de **Ajustes** conducirá al menú de Ajustes del juego.
- Pulsar el botón de **Salir** cerrará el juego después de una animación similar a la de cuando se inicia/transiciona.
- Pulsar el botón de **Créditos** llevará a la interfaz de Créditos del juego.

También se puede observar la **versión** actual del juego en la esquina inferior izquierda y el cómo predomina por encima de los botones el **título del juego**.

5.3.2 Menú de Selección de Niveles



Boceto de la Selección de Niveles.



Versión final del Menú de Selección de Niveles en Inglés, con algunos niveles completados y/o desbloqueados.

Versión final del Menú de Selección de Niveles en Español, con ningún nivel completado.

El menú de selección de niveles presenta una **lista de niveles disponibles**.

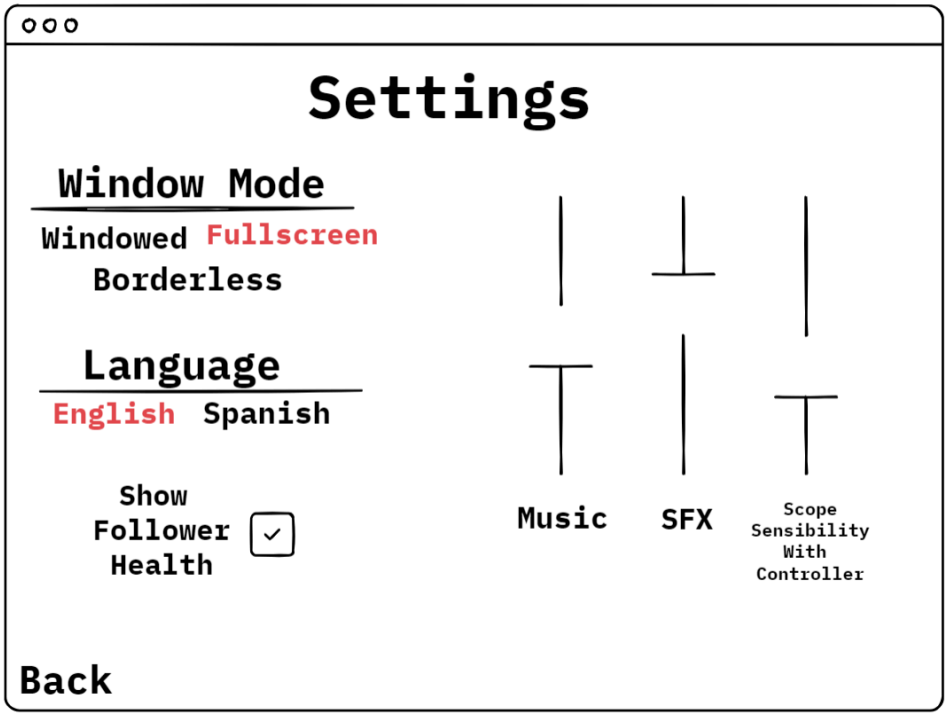
El menú de selección de niveles presenta una lista organizada que permite al usuario **visualizar el progreso** dentro del juego. Los niveles que todavía no han sido completados aparecen bloqueados, lo cual se indica mediante un icono de **candado** que cumple la función de transmitir de manera clara e intuitiva que el acceso a dichos niveles aún no está disponible. Esta **restricción** se mantiene hasta que el jugador haya superado las etapas anteriores, garantizando así una progresión ordenada y coherente lo que hará que su aprendizaje sea continuo y claro.

Una vez **desbloqueado un nivel**, el usuario puede visualizar una interfaz específica que ofrece información relevante antes de iniciar el nivel. Dicha interfaz incluye el número correspondiente al nivel, un título descriptivo que anticipa los retos o la **temática** que se encontrará en esa fase, así como un sprite representativo de uno de los enemigos que aparecerán durante el desarrollo del nivel. Estos elementos cumplen una **doble función**: por un lado, aportan información práctica al jugador y, por otro, contribuyen a reforzar la ambientación y el contexto narrativo del juego.

Adicionalmente, el menú incorpora un **sistema de navegación** mediante **flechas** situadas en los extremos derecho e izquierdo de la pantalla. Gracias a este mecanismo, el usuario puede desplazarse fácilmente a través del conjunto de niveles disponibles y obtener una visión más amplia del contenido total que ofrece el juego, mejorando así la usabilidad y la experiencia de interacción con la interfaz.

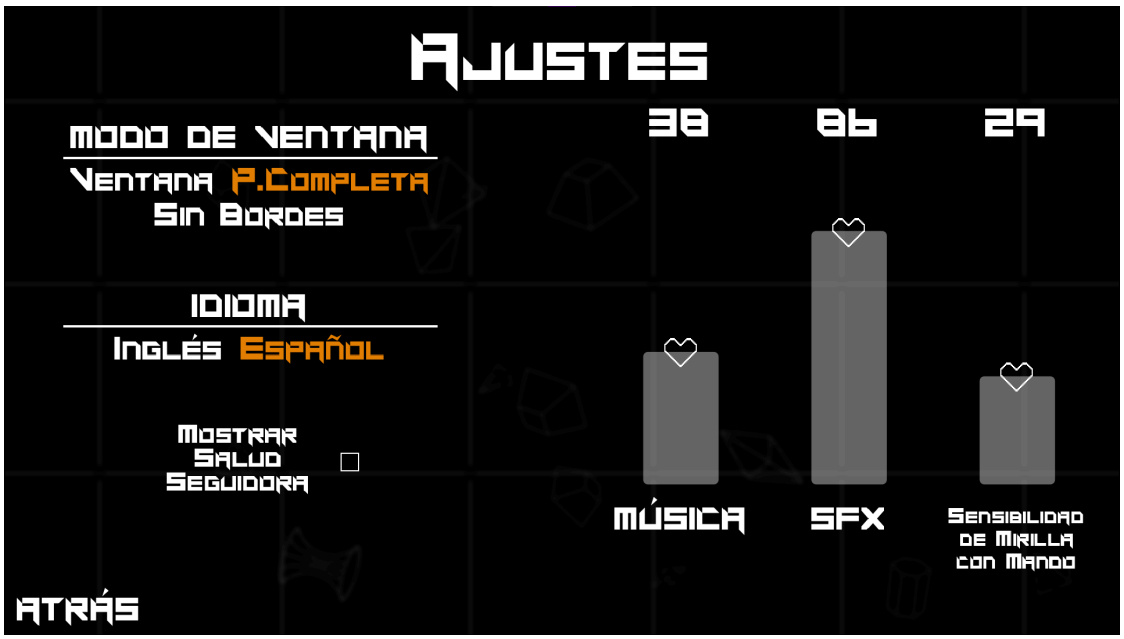
Por último, hay un botón para **volver hacia atrás** al menú principal, que se encuentra en la esquina inferior izquierda.

5.3.3 Menú de Ajustes



Boceto del Menú de Ajustes.

Versión final del Menú de Ajustes en Inglés.



Versión final del Menú de Ajustes en Español.

El menú de ajustes se compone de un conjunto de herramientas para ajustar el juego a gusto del jugador. Entre los diferentes componentes podemos encontrar:

- **Cambio del Modo de Ventana**: Permite cambiar la ventana del juego a Pantalla Completa, Ventana (que es reajustable después como cualquier programa) y Pantalla Completa Sin Bordes.

- **Selección de Idioma:** El jugador podrá seleccionar entre los diferentes idiomas disponibles en el juego, actualmente entre inglés y español.

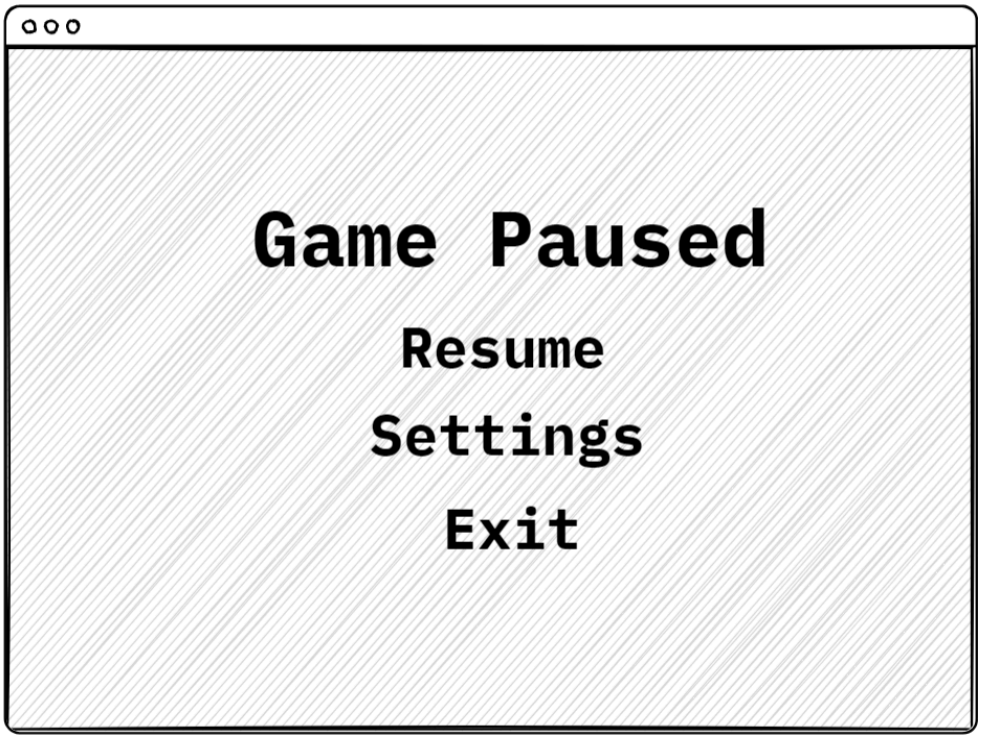
- **Cambio de Volumen de Música y SFX:** Estos dos deslizadores permiten ajustar el volumen de la música y los efectos de sonido del juego en una escala del 0 al 100.

- **Cambio de la Sensibilidad de la Mirilla:** Este deslizador permite ajustar la velocidad con la que la mirilla telescópica se mueve cuando usas un mando y la controlas con el joystick.

- **Checkbox de Mostrar Salud Seguidora:** Este botón permite alternar entre si quieres visualizar en tu nave del jugador una interfaz con los corazones que tienes de la misma manera que los enemigos poseen. Esta opción es un extra a gusto del jugador debido a que ya existe una interfaz con la salud del jugador en la parte superior izquierda de la pantalla.

Al pulsar el botón de Atrás se regresa al menú previo desde el que se accedió a los ajustes (menú principal o menú de pausa).
Cualquiera de los ajustes son aplicados y guardados al mismo momento de ser pulsados, no hace falta pulsar un botón de Aplicar Cambios.

5.3.4 Menú de Pausa



Boceto del Menú de Pausa.

Versión final del Menú de Pausa en Inglés.

Versión final del Menú de Pausa en Español.

El menú de pausa lo podemos encontrar si realizamos la acción de pausar el juego mientras estamos en una partida. El gameplay se detendrá y el menú aparecerá.

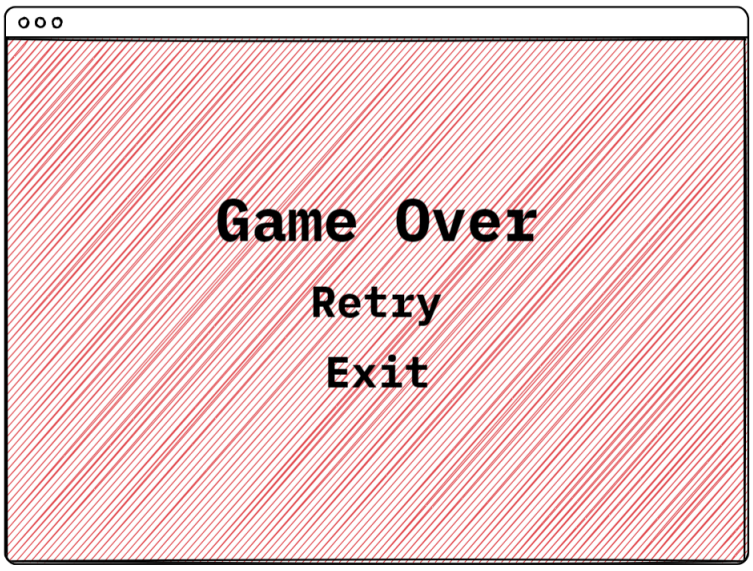
Además de servir como sistema de pausa, el menú contiene los siguientes elementos:

- **Botón de Reanudar:** Este cerrará el menú de pausa y reanuda la partida. Hay que destacar que si en el menú de pausa se realiza la acción de pausar también se reanudará la partida sin tener que hacer uso de este botón.

- **Botón de Ajustes:** Este botón navega al menú de ajustes de la misma forma que el menú principal posee.

- **Botón de Salir:** Sale del nivel actual y te envía al menú de Selección de niveles.

5.3.5 Menú de Fin del Juego



Boceto del Menú de Fin del Juego.

Versión final del Menú de Fin del Juego en Inglés.

Versión final del Menú de Fin del Juego en Español.

El menú de Fin del Juego se presenta una vez el jugador ha muerto en un nivel.
El fondo rojo semitransparente está diseñado para reforzar la sensación de peligro, fracaso y finalización de la partida.
Esta interfaz incluye los siguientes elementos:

- **Botón de Reintentar:** Reinicia el nivel actual, comenzando desde el principio como si se hubiera seleccionado desde el menú de Selección de Niveles.

- **Botón de Salir:** Sale del nivel actual y te envía al menú de Selección de niveles.

5.3.6 Menú de Nivel Completado

Boceto del Menú de Nivel Completado.

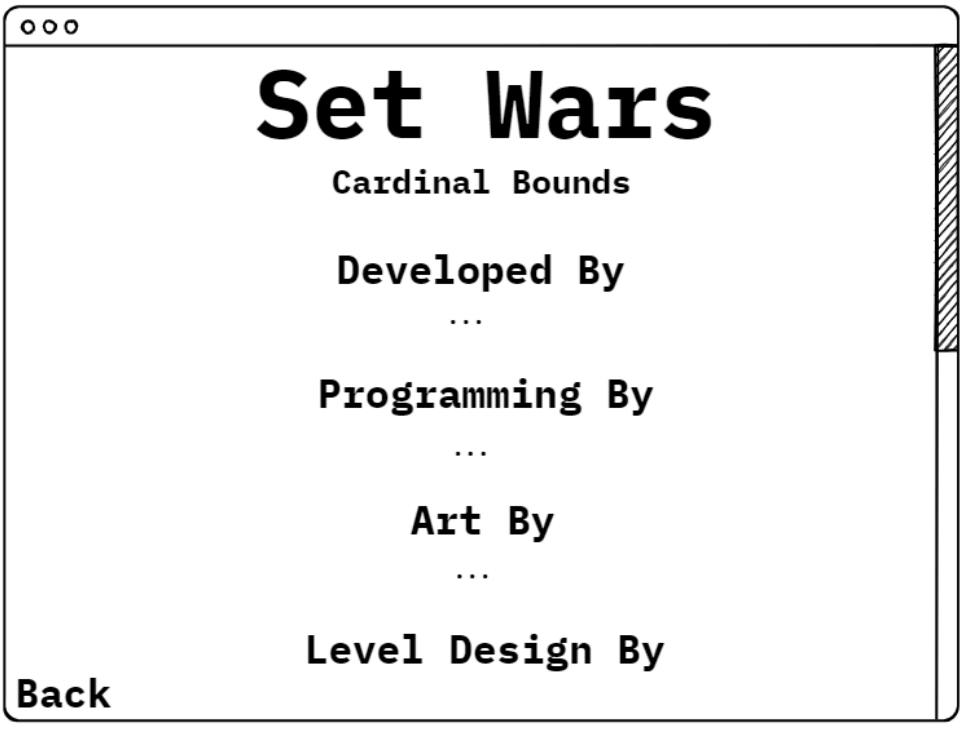
Versión final del Menú de Nivel Completado en Inglés.

Versión final del Menú de Nivel Completado en Español.

El menú de Nivel Completado se muestra cuando el jugador logra finalizar un nivel.

Este menú es muy similar al de Fin del Juego, ya que cuenta con dos botones que cumplen las mismas funciones. La única diferencia es que, en este caso, el botón para reiniciar el nivel se llama *Reiniciar* en lugar de *Reintentar*. Otra diferencia es que esta vez el fondo es negro transparente, queriendo transmitir una sensación de calma y finalización.

5.3.7 Créditos



Boceto de los Créditos, en la parte inicial.

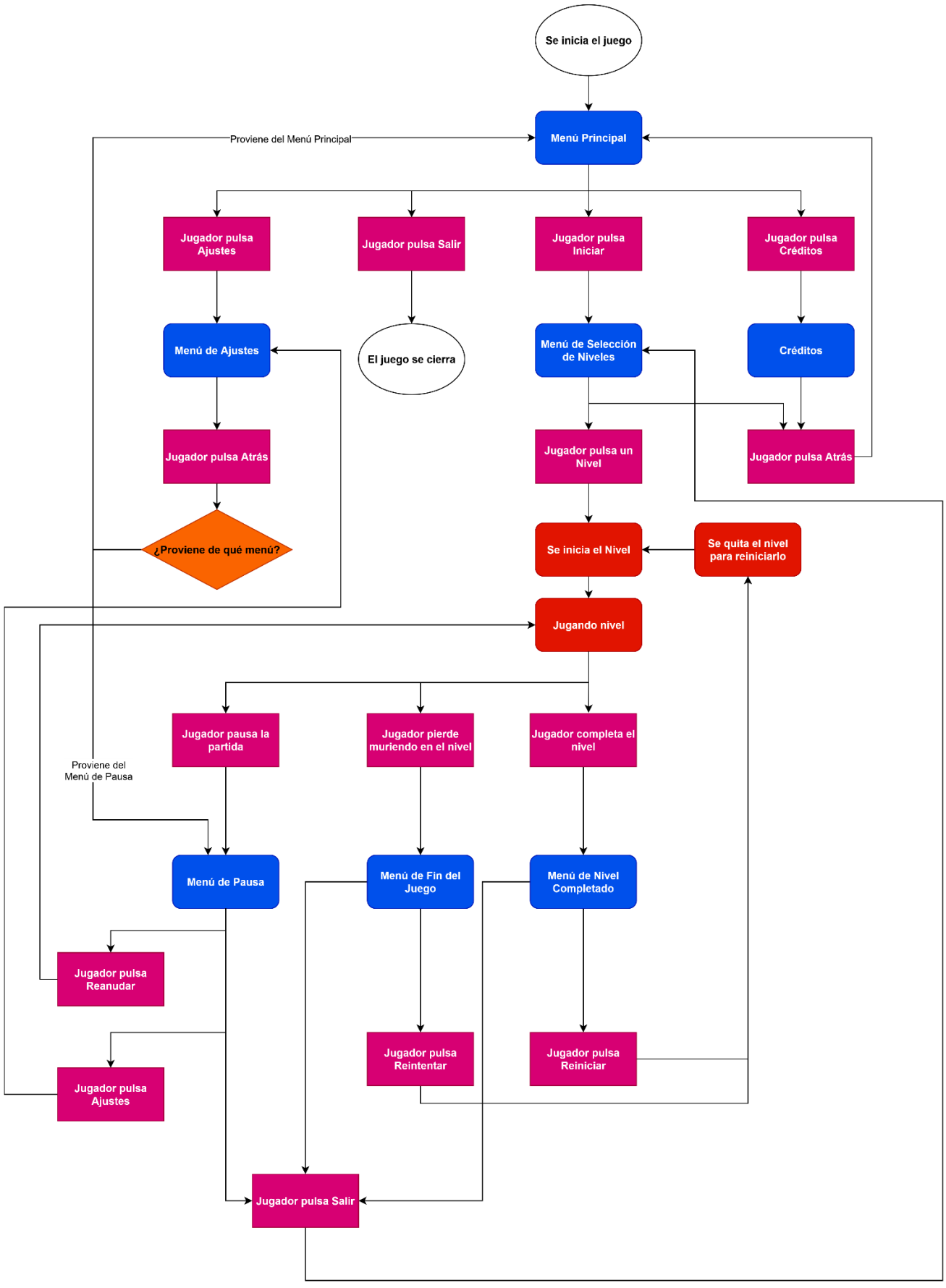
Versión final de los Créditos en Inglés, en la parte inicial.

Versión final de los Créditos en Español, en la parte final.

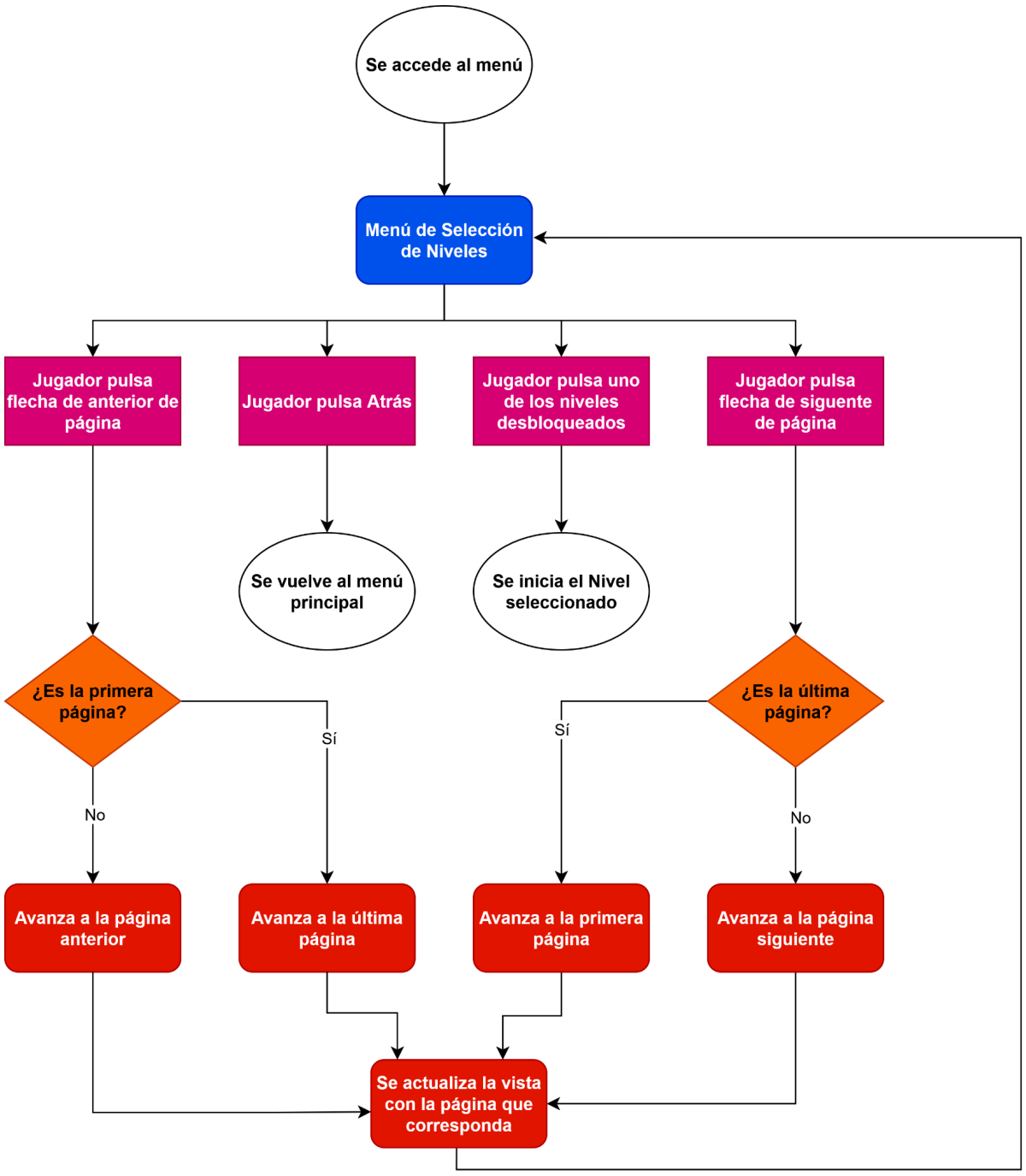
Los créditos son navegables con una *scrollbar* vertical con la que se puede navegar por ellos y ver todo su contenido. También se dispone de un botón de volver atrás, que regresa al menú principal.

5.4 Diagramas de Flujo

A continuación se presenta un diagrama de flujo general de toda la navegación entre los diferentes menús del juego:


Se inicia el juego
Menú Principal
Proviene del Menú Principal
Jugador pulsa Ajustes
Jugador pulsa Salir
Jugador pulsa Iniciar
Jugador pulsa Créditos
Menú de Ajustes
El juego se cierra
Menú de Selección de Niveles
Créditos
Jugador pulsa Atrás
Jugador pulsa un Nivel
Jugador pulsa Atrás
¿Proviene de qué menú?
Se inicia el Nivel
Se quita el nivel para reiniciarlo
Jugando nivel
Jugador pausa la partida
Jugador pierde muriendo en el nivel
Jugador completa el nivel
Proviene del Menú de Pausa
Menú de Pausa
Menú de Fin del Juego
Menú de Nivel Completado
Jugador pulsa Reanudar
Jugador pulsa Reintentar
Jugador pulsa Reiniciar
Jugador pulsa Ajustes
Jugador pulsa Salir

El siguiente es un diagrama de flujo enfocado más específico en la navegación dentro del menú de Selección de Niveles:

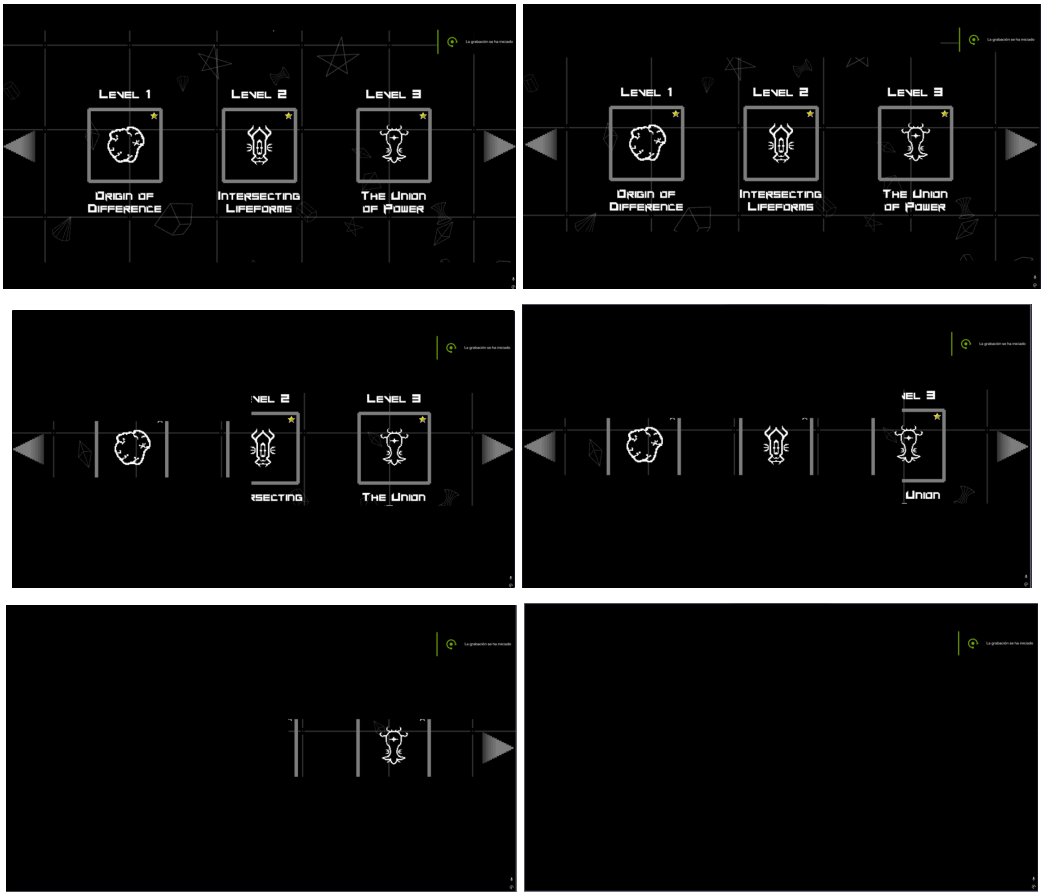


5.5 Pantalla de Carga

La pantalla de carga del juego consiste en una **animación** de 10 barras rectangulares negras que **cubren y destapan la pantalla** (animadas por pares), dejando un fondo negro un instante hasta que el nivel o las interfaces se han cargado por completo.

También se puede apreciar esta animación de cubrir la pantalla cuando salimos del juego, antes de cerrarse por completo y, antes de ver el menú principal al iniciar el juego también

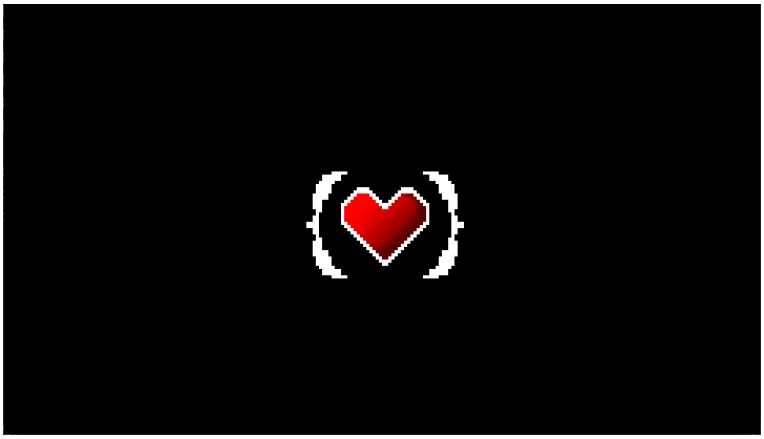
podemos ver la animación donde se van destapando a modo de despedida y presentación respectivamente.



Ejemplo de la animación de transición cuando se elige un nivel, se puede ver como la pantalla va cubriéndose de más barras negras.

5.6 Pantalla de Arranque (Splash Boot)

Cuando se abre el juego, durante unos instantes (lo que tarde el programa en poder ser iniciado) se aprecia una pantalla de *splash* con el icono del juego.



Pantalla de arranque del juego mostrando el logo de este.

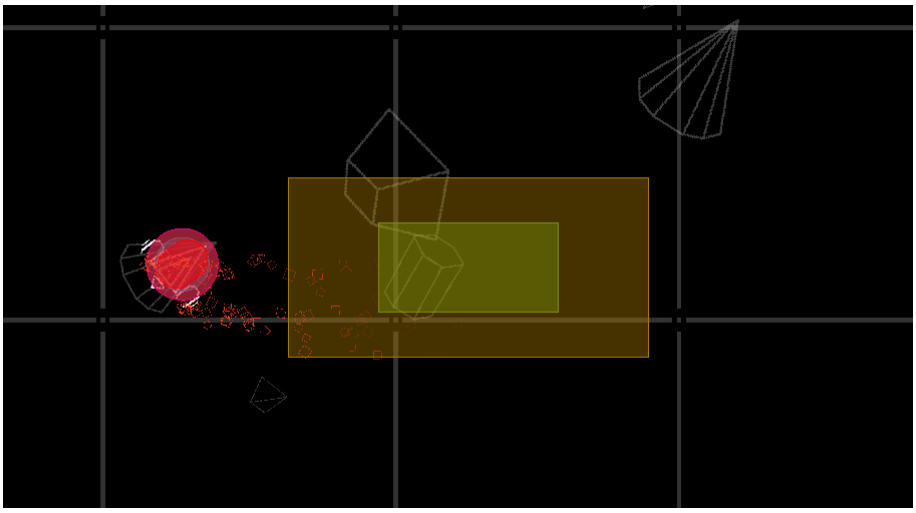
6. Cámara del Juego

La cámara del juego es en **tercera persona** y presenta una **perspectiva cenital**.

Está diseñada para **seguir al jugador** de forma suave, priorizando siempre la visibilidad del entorno y del propio personaje. Combina el seguimiento dinámico con zonas estáticas en momentos clave del juego, logrando una experiencia visual coherente y sin desorientar al jugador.

La cámara utiliza un sistema de áreas de colisión (para saber cuándo moverse):

- **Área de Entrada:** Cuando el jugador entra en esta área, la cámara deja de seguirlo.
- **Área de Salida:** Cuando el jugador sale de esta área, la cámara vuelve a seguirlo.



En color amarillo claro el área de entrada y en oscuro el de salida.

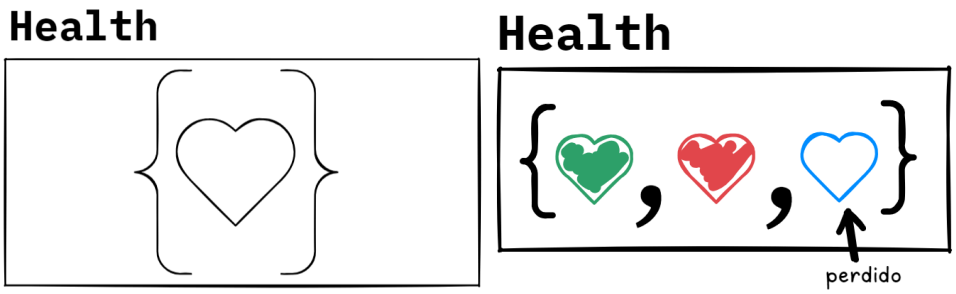
7. Sistema HUD

El sistema HUD (Head-Up Display) representa la interfaz visual que se mantiene visible para el jugador durante el transcurso del juego, mostrando información esencial relacionada con su estado actual.

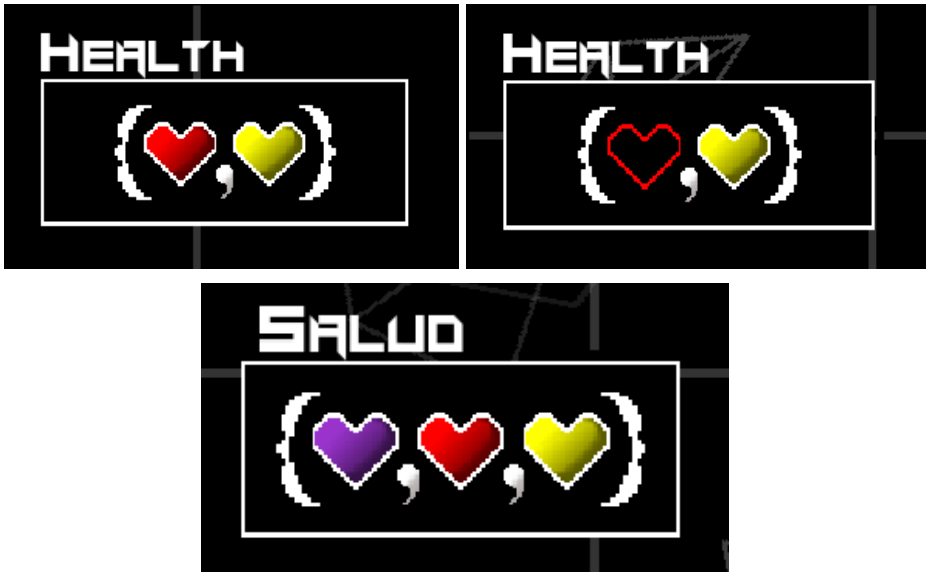
7.1 Indicador de Salud

Se representa mediante **corazones** dentro de unas llaves “{}” y separados por comas, para representar que son elementos dentro de un conjunto. Estos corazones son de **diferentes colores** porque los elementos de un conjunto nunca se repiten.

Si el jugador pierde alguno de los corazones se mostrará una **silueta del color del corazón** sin el relleno este. Si todos los corazones están vacíos, significa que la nave los ha perdido todos, lo que provoca su destrucción y el fin de la partida.



Boceto del Indicador de Salud.



Se encuentra en la parte superior izquierda de la pantalla.

7.1.1 Indicador de Salud Seguidor

El mismo indicador de salud aparecerá mucho **más pequeño y siguiendo al jugador** de la misma forma que se presenta la salud de los enemigos. Esta interfaz está disponible a ser

activada o no desde el menú de opciones, debido a que su uso es un extra visual que no aporta más información que la ya dada por el indicador de salud normal.

Su propósito es una mezcla de comodidad, para el usuario de poder ver su salud sin apartar tanto la vista de la nave, y de coherencia estética, para que el jugador coincida en tener esta interfaz como los enemigos.

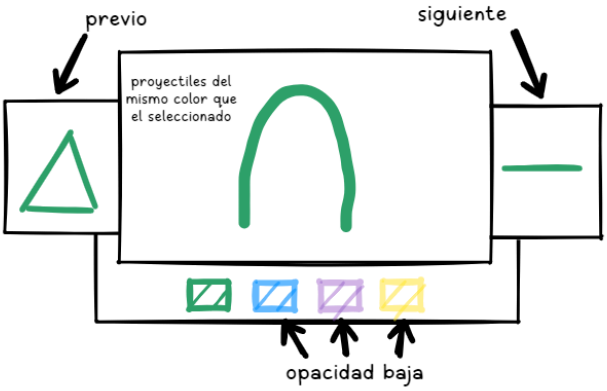
7.2 Selector de proyectiles

Muestra el **tipo** y **color** de proyectil que actualmente el jugador puede disparar.

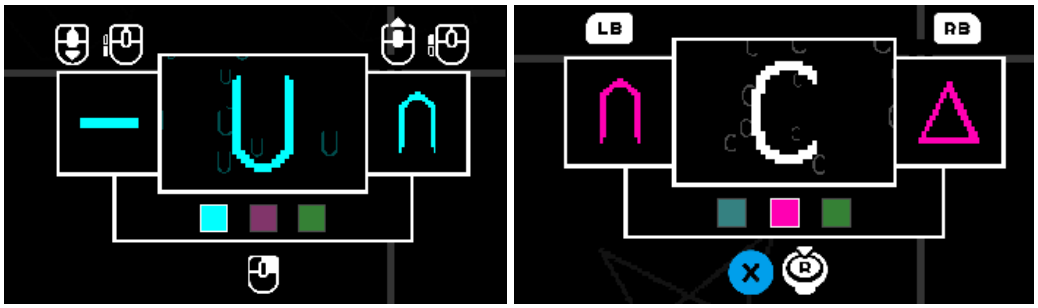
Además, se visualizan los proyectiles adyacentes en el arsenal disponible: el tipo anterior aparece a la izquierda, y el siguiente a la derecha del actual.

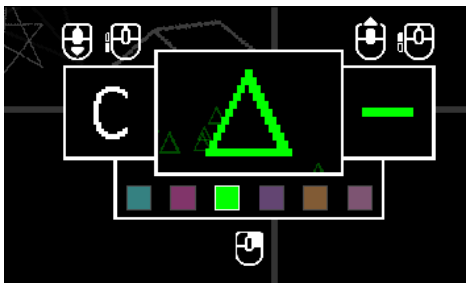
Los colores disponibles que se pueden disparar se ven presentados por una **fila de cuadrados** de colores, el color **actual seleccionado** se diferencia del resto ya que no tiene una opacidad baja, que lo hace más destacable. Los tipos de proyectiles también **se ajustan al color** seleccionado para que sea más fácil ver cuál es el actual (salvo excepciones, como el *complemento*).

Cuando el jugador cambia de proyectil, esta interfaz se actualiza en tiempo real para reflejar la nueva selección.



Boceto del Selector de Proyectiles.





Se encuentra en la parte superior derecha de la pantalla.

7.3 Mira telescópica de Apuntado

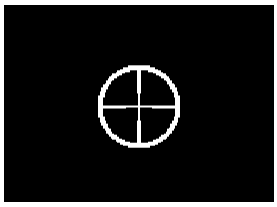
La mira telescópica es un elemento de interfaz visual que cumple una función clave en la jugabilidad: sirve como referencia para la rotación y dirección de disparo de la nave del jugador.

Este se encuentra siempre visible en pantalla y puede desplazarse libremente por el escenario bajo los límites visuales de la cámara. Su posición se utiliza como punto de referencia para que la nave gire y apunte en esa dirección específica, independientemente del desplazamiento.

El método de control de la mira telescópica depende del dispositivo de entrada:

- Con teclado y ratón, la mira telescópica sigue directamente el movimiento del ratón.
- Con mando, se controla mediante el joystick derecho.

Aunque forma parte de la interfaz, la mira telescópica tiene impacto directo en el comportamiento del jugador, por lo que es considerado tanto parte del HUD como del sistema de control del personaje.

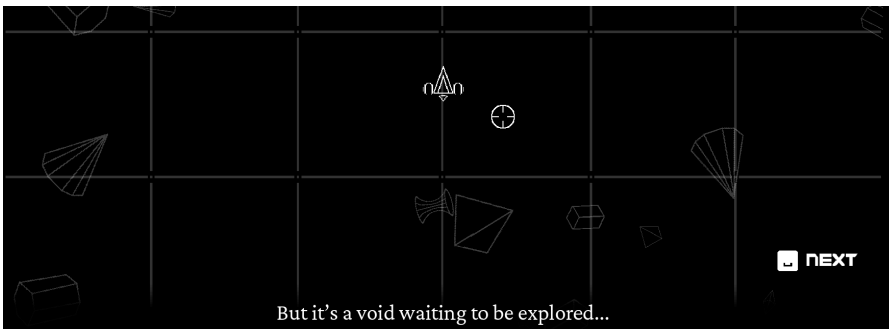


Diseño final de la mira telescópica.

7.4 Diálogo del Juego

Durante el **transcurso del juego**, aparecerán diferentes diálogos para **informar y/o ayudar** al jugador según la situación. Estos textos podrán avanzarse **manualmente** o automáticamente tras ciertos **eventos** del juego (depende del diálogo).

Esta interfaz presenta un fondo de degradado negro para facilitar la lectura del texto sobre el escenario de juego. También hay una leyenda de con qué input poder avanzar si el diálogo lo permite.

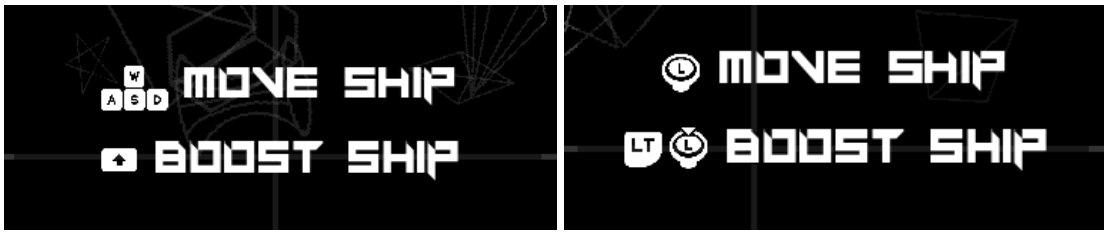


Se encuentra en la parte inferior completa de la pantalla.

7.5 Leyenda de Controles

También durante el **transcurso del juego** aparecerán diferentes mensajes de ayuda informando de ciertas acciones junto a uno o varios iconos (de teclado/ratón o de mando, según el control actual) para informar al jugador de ciertos controles que hasta el momento no se le había enseñado.

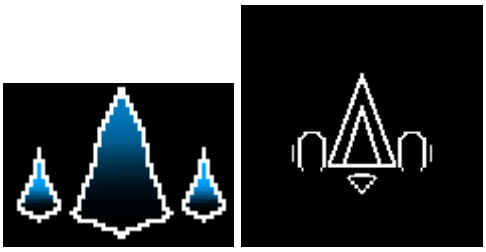
Esta interfaz presenta un fondo de degradado negro para facilitar la lectura del texto sobre el escenario de juego. Aparece y desaparece según se precise durante el transcurso del nivel.



Se encuentra en la parte superior central de la pantalla.

8. Personaje del Jugador

El jugador es una **nave** que presenta diferentes acciones a poder realizar para hacerse camino a través de los enemigos. Los enemigos **intentarán destruirla** mientras esta tendrá que defenderse aplicando la teoría de conjuntos para poder llegar a vencerlos.



Boceto y diseño final del personaje.

8.1 Desplazamiento

El jugador puede mover la nave en todas las direcciones: hacia **adelante (arriba),** hacia **atrás (abajo),** a la **izquierda** y a la **derecha.** Además, puede moverse en **diagonal** combinando dos direcciones simultáneamente. Tiene libertad de movimiento para esquivar obstáculos y para avanzar por el escenario.

Además se puede **aumentar la velocidad** del desplazamiento manteniendo pulsado el control correspondiente.

No existe aceleración ni inercia; la nave alcanza su velocidad máxima instantáneamente cuando se pulsa una dirección y se detiene de inmediato al soltar el control, teniendo la velocidad adecuada si se está usando el aumento de velocidad o no.

8.2 Rotación

El jugador puede rotar la nave tanto a la **izquierda** como a la **derecha**, pudiendo tener la nave apuntando hacia cualquiera de los 360º grados respecto del eje Z. Esta rotación es indefinida en ambas direcciones, sin ningún tipo de restricción o límite.

La nave utiliza la posición de la **mira telescópica** para obtener la **referencia** con la que determina hacia dónde debe rotar y apuntar.

Hay que tener en cuenta que el desplazamiento definido previamente no es afectado por la rotación. Es decir, la dirección hacia la que mira la nave no modifica los controles de desplazamiento. Por ello, aunque la nave esté rotada, los controles seguirán desplazándola en cualquier dirección según el eje global, no según la orientación de la nave.

8.3 Disparo

El jugador puede hacer que la nave dispare: este disparo se realizará comenzando desde la nave hacia la dirección en la que esta se encuentre apuntando (el ángulo según la rotación definida previamente).
Lo que será disparado es un proyectil que se explica en detalle en el punto **9.3**

8.4 Cambiar tipo y color de proyectil

El jugador dispone de diferentes **tipos** de proyectiles y siempre cuenta con **uno activo** con el que puede disparar. En cualquier momento, puede cambiar al proyectil **anterior o siguiente de su arsenal**, siempre que haya más tipos disponibles en el nivel, modificando así el tipo actual.

De forma similar, el jugador puede cambiar entre distintos **colores** de proyectil, seleccionando en cada momento **el color activo**.

Estas funcionalidades permiten adaptar los disparos a diferentes situaciones del juego, especialmente cuando ciertos elementos reaccionan de manera distinta según el color y tipo utilizado.

9. Mecánicas del Juego

Cuando nos encontramos jugando un nivel del juego, se nos presenta un escenario en el espacio donde está la nave del jugador, que será la que podremos controlar. En dicho escenario irán apareciendo distintos enemigos con los que interactuar.

Algunos aspectos importantes que debemos explicar son los siguientes:

9.1 Vida

El sistema de vida del jugador es similar al de los enemigos (en general, nos referimos a todos ellos como "**entidades**"). Todos tienen su vida representada como un **conjunto matemático**, donde los **corazones** son los **elementos**. Estos corazones siempre son de **diferentes colores**, reflejando la naturaleza propia de los conjuntos.

A través de los diferentes proyectiles u otras formas de daño (**9.2**), las entidades pueden **perder** o **ganar** corazones.

Para explicar correctamente cómo funcionan los proyectiles y las distintas formas de daño, debemos tener en cuenta estos detalles. Consideraremos *S* como el conjunto de corazones que una entidad posee en ese momento (es decir, los que no ha perdido):

- Cuando *S* no ha perdido ningún corazón, es equivalente a **su** conjunto universo (*U*).
- Cuando *S* ha perdido algún corazón, **su** complemento son los corazones perdidos.
- Si a *S* se le aplica una operación (un daño) donde el conjunto implicado contiene corazones que **no pertenecen al universo** de *S*, los efectos dependerán de la operación concreta que se haya aplicado.

9.2 Daño

El daño en el juego es el concepto de aplicar operaciones de conjunto sobre la salud de una entidad.
Un daño es aplicable a una entidad de diferentes formas:

- Por contacto con un **proyectil (9.4).**
- Por contacto con **otra entidad** que aplica daño al contacto (asteroides).
- Por contacto con **los límites del mapa** (**9.7.3**).

Los tipos de daño se explican en el apartado de proyectiles, pero hay que tener en cuenta que todo daño aplicado en el juego necesita:

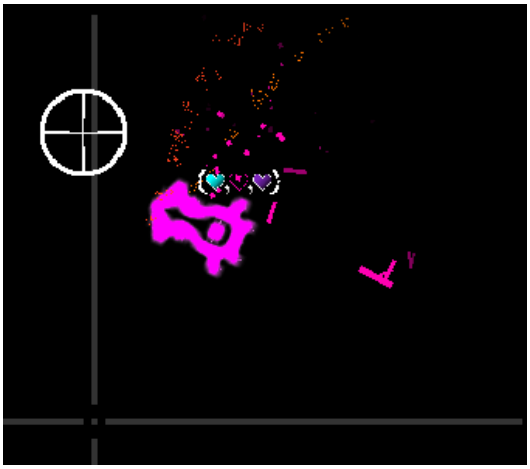
- Un **objetivo** que va a recibir el daño.
- El **tipo de daño** que se le va a aplicar (el tipo de operación).

- Los **conjuntos de corazones que se involucran** en la operación: los de la entidad (obligatorios, actúan como primer operando) y los que el daño aplica (no siempre obligatorios, actúan como segundo operando).

Cuando una entidad recibe un daño, esta realiza una animación en su sprite donde se le aplica un efecto de brillo con distorsión ("*glow*") y se cambia su color al del primer corazón presente en el conjunto del segundo operando, es decir, un proyectil.

Debido a que normalmente el segundo operando es solo un corazón el conjunto, siempre se entiende bien qué daño se recibe.

Si el conjunto es vacío o el tipo de daño no aplica segundo operando (el complemento), el color es blanco.



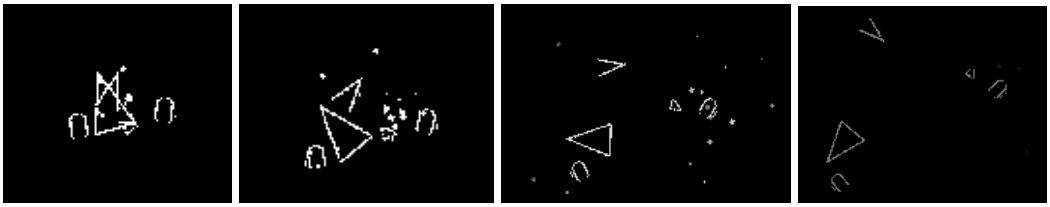
Enemigo recibiendo daño por una Diferencia magenta.

Jugador recibiendo daño por una Unión roja.

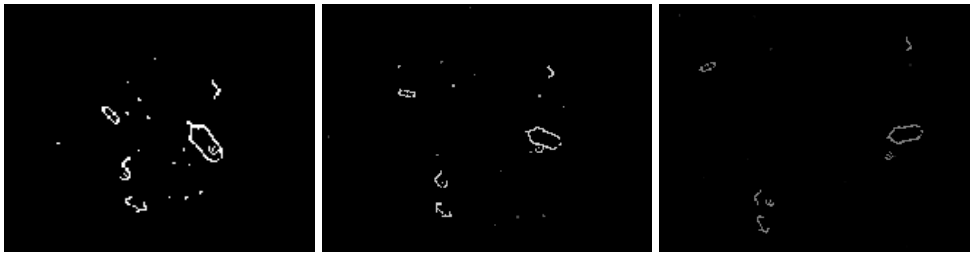
9.3 Muerte

Si una entidad **pierde todos los corazones** muere al instante y esta realiza una **animación** donde se rompe en pedazos y desaparece del escenario.

En el caso del jugador es así como **pierde la partida**, mostrando la pantalla de fin del juego después de la animación de muerte.

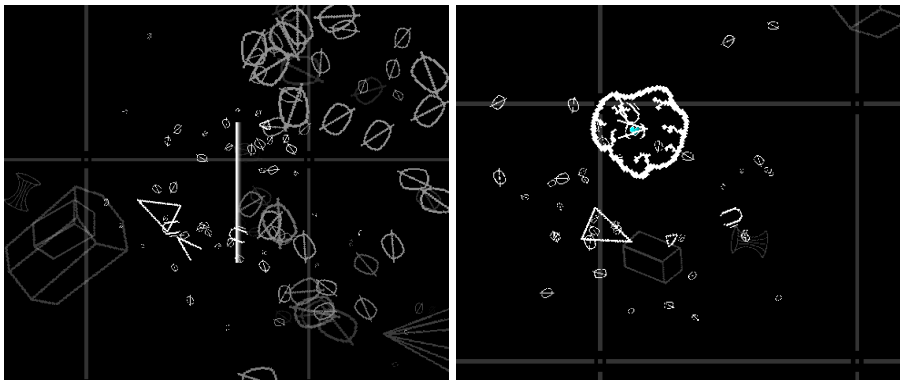


Muerte del jugador.



Muerte de un enemigo.

Si un jugador muere por aplicarle un daño donde el segundo operando era un conjunto vacío, las partículas de la animación serán símbolos de conjuntos vacíos.



Muerte del jugador por conjunto vacío.

9.4 Proyectiles

Las **entidades pueden disparar** diferentes tipos de proyectiles. Estos se **desplazan de forma continua** en la dirección en la que fueron disparados, manteniéndose en movimiento hasta **impactar con un objeto** o hasta que se agote su **tiempo de vida** (en caso de tener uno).

Cada proyectil **contiene un corazón de un cierto color,** el cual determina su color, pintando su apariencia con el tono correspondiente.

El juego está preparado para que un proyectil pueda llevar **más de un corazón**, aunque por el momento, en la práctica, **solo se hace uso de proyectiles con un único corazón** debido a limitaciones de que puede ser muy difícil que el jugador configure qué tantos colores usar se ha restringido a que solo haya uno, pero podría considerarse como una futura implementación, sobretodo porque lo enemigos sí que podrían configurarse con esto.

Existen múltiples tipos de proyectiles. Cada tipo se distingue por el **efecto de daño** que aplica a la entidad con la que impactan.

9.4.1 Tipos de Proyectiles

Cada tipo de proyectil **representa una operación matemática de la teoría de conjuntos**, y su aspecto visual se corresponde con el símbolo matemático que identifica dicha operación.

El efecto de daño que produce un proyectil está determinado por la aplicación lógica de su operación sobre el **conjunto de corazones de salud** de la entidad impactada (S), en relación con el **conjunto de corazones que porta el proyectil** (P).

9.4.1.1 Diferencia

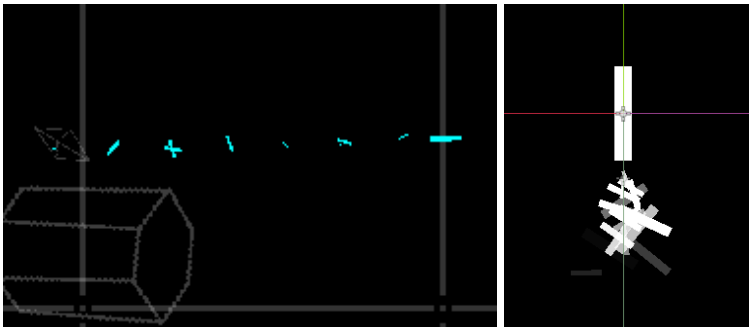
El proyectil eliminará del conjunto de salud de la entidad todos los corazones que coincidan con los que lleva el proyectil.

Casos especiales del proyectil:

- Si solo lleva un corazón, como mucho solo quitará uno.
- Si lleva algún corazón que no pertenece al universo de la entidad, se ignora el color.

Ejemplo:

Salud de la entidad: {💙,💚,💛}
Proyectil: {💛}
Salud de la entidad tras el impacto: {💙,💚}



Vista del proyectil durante el gameplay y en el editor.

9.4.1.2 Unión

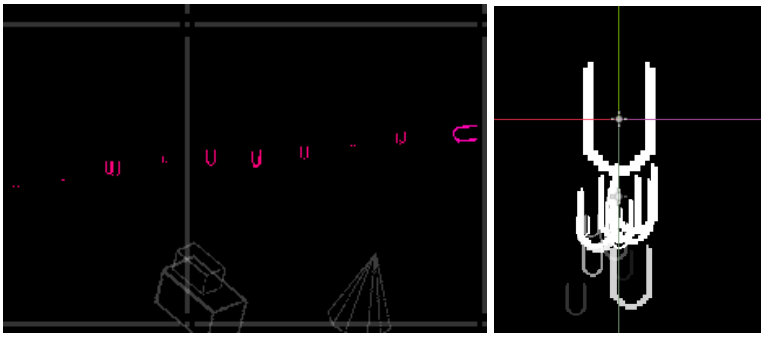
El proyectil añadirá al conjunto de salud de la entidad todos los corazones que lleva este, siempre que no estén ya presentes (ya que los conjuntos no repiten elementos).

Casos especiales del proyectil:

- Si solo lleva un corazón, como mucho solo se recuperará uno.
- Si lleva algún corazón que no pertenece al universo de la entidad, se ignora el color.

Ejemplo:

Salud de la entidad: {💙,💚}
Universo de la Entidad: {💙,💚,💛}
Proyectil: {💛}
Salud de la entidad tras el impacto: {💙,💚,💛}



Vista del proyectil durante el gameplay y en el editor.

9.4.1.3 Intersección

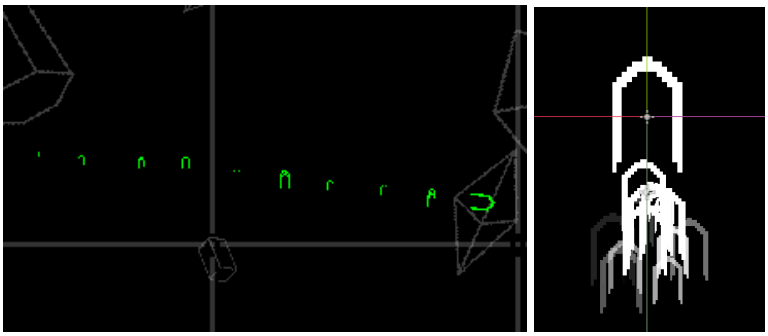
La salud de la entidad quedará reducida únicamente a los corazones que **coincidan** con los del proyectil, eliminando el resto.

Casos especiales del proyectil:

- Si solo lleva un corazón, la entidad conservará solo ese (si coincide).
- Si lleva algún corazón que no pertenece al universo de la entidad, ese color se ignora. Pero si **ningún corazón del proyectil coincide con los de la entidad**, esta muere, ya que la intersección resultante sería el conjunto vacío.

Ejemplo:

Salud de la entidad: {💙,💚,💛}
Proyectil: {💚}
Salud de la entidad tras el impacto: {💚}



Vista del proyectil durante el gameplay y en el editor.

9.4.1.4 Diferencia Simétrica

Este proyectil es algo complejo: Elimina de la entidad los mismos corazones que este tiene y que la entidad todavía no ha perdido (como la Diferencia) y restaura a la entidad los mismos corazones que este tiene y que la entidad sí tenía perdidos (como la Unión).

Casos especiales del proyectil:

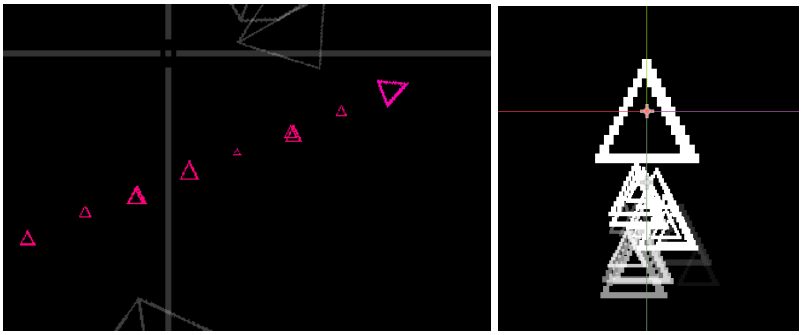
- **Si el proyectil solo lleva un corazón,** el efecto parece que es solo una Diferencia o una Unión según el estado de la entidad:
    - Si la entidad tiene (no perdido) el mismo color de corazón que el proyectil, lo perderá.
    - Si la entidad tiene perdido el mismo color de corazón que el proyectil, lo recuperará.
- Si el proyectil lleva algún corazón que no pertenece al universo de la entidad, dicho corazón se ignora completamente en el cálculo del efecto.

1. Ejemplo:
    Salud de la entidad: {💙,💚,💛}
    Universo de la Entidad: {💙,💚,💛,🧡}
    Proyectil: {💛}
    Salud de la entidad tras el impacto: {💙,💚}

Se elimina 💛 (está en ambos conjuntos).
Se ignora 🧡 (estaba en el universo pero no en la salud y no lo llevaba el proyectil).

2. Ejemplo:
    Salud de la entidad: {💙,💚}
    Universo de la Entidad: {💙,💚,💛,🧡}
    Proyectil: {💛}
    Salud de la entidad tras el impacto: {💙,💚,💛}

Se recupera 💛 (está solo en el proyectil).
Se ignora 🧡 (estaba en el universo pero no en la salud y no lo llevaba el proyectil).



Vista del proyectil durante el gameplay y en el editor.

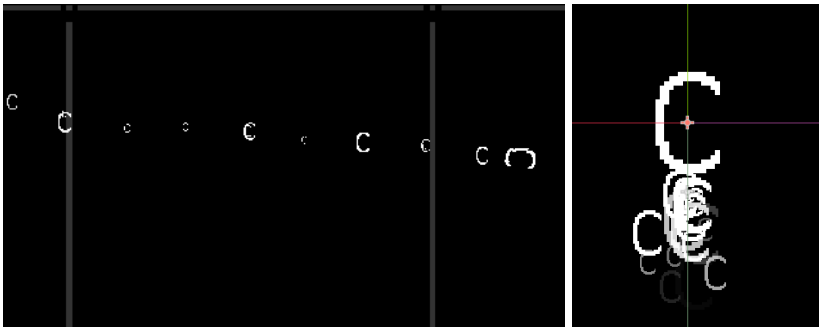
9.4.1.5 Complemento

El proyectil eliminará del conjunto de salud **todos los corazones presentes** y añadirá todos los que **le faltaban** para completar su conjunto universo. Se trata de un proyectil **que no lleva corazones consigo**, y su efecto depende únicamente del estado actual de salud de la entidad que recibe el impacto.

Dado que no contiene corazones, su color **es siempre blanco**, representando la neutralidad del complemento respecto al contenido del proyectil

Ejemplo:

Entidad: {💙,💚}
Universo de la Entidad: {💙,💚,💛}
Resultado tras el impacto: {💛}



Vista del proyectil durante el gameplay y en el editor.

9.5 Enemigos

En el juego, los enemigos funcionan como entidades autónomas que toman decisiones simples basadas en su entorno y en reglas predefinidas. Cada tipo de enemigo tiene su propia personalidad y forma de interactuar con el jugador y el mundo.

9.5.1 Reglas Básicas de Comportamiento

- **Visión:** Los enemigos pueden "ver" o detectar al jugador si este se encuentra dentro de un área determinada (como un radar o un campo de visión). Si el jugador está fuera de ese rango, el enemigo no suele reaccionar.

- **Movimiento:** Una vez que el enemigo detecta al jugador, decidirá moverse según sus reglas de comportamiento. Pueden perseguir, huir u orbitar alrededor del jugador según a qué distancia están de él. Cada enemigo puede tener mayor o menor velocidad.

- **Ataque:** Si el enemigo ha detectado al jugador, empezará a atacar. Según el enemigo, podrá disparar **un concreto arsenal**, con sus **proyectiles** teniendo una cierta **velocidad**, con una concreta **cadencia de disparo** y cada enemigo puede disparar **uno o varios proyectiles a la vez**. Además, hay enemigos que al **contacto** hacen daño.

9.5.2 Tipos de Enemigos

A lo largo de los niveles se nos van presentando enemigos de forma progresiva y que van teniendo diferentes características.

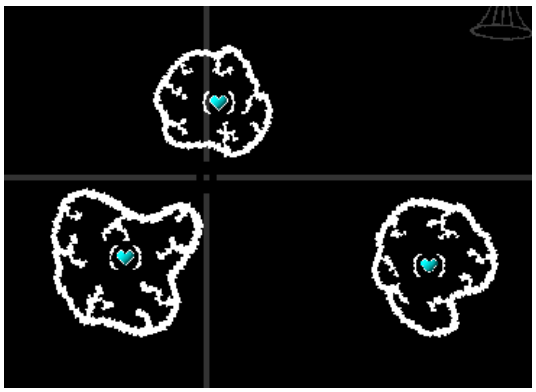
9.5.2.1 Asteroide

No es realmente un enemigo inteligente. No persigue ni ataca al jugador. Simplemente rota sobre sí mismo sin desplazarse y, si choca con el jugador, le hace daño mortal (lo destruye por completo). Es un obstáculo peligroso. Los hay de muchos tamaños.

El daño mortal se debe a que los asteroides hacen **daño por contacto** (9.2), y están programados para aplicar un daño de Intersección con un conjunto vacío como segundo operando, es decir, quitar todos los corazones.

En niveles posteriores, los asteroides son capaces de **moverse orbitando** alrededor de un punto fijo, normalmente en grupos. Esto se logra mediante *triggers* de rotación.
En ocasiones, estos grupos de asteroides que orbitan forman parte de un conjunto mayor que también rota, provocando que los subgrupos giren al mismo tiempo que el grupo principal. De este modo se generan patrones de movimiento más complejos y dinámicos.

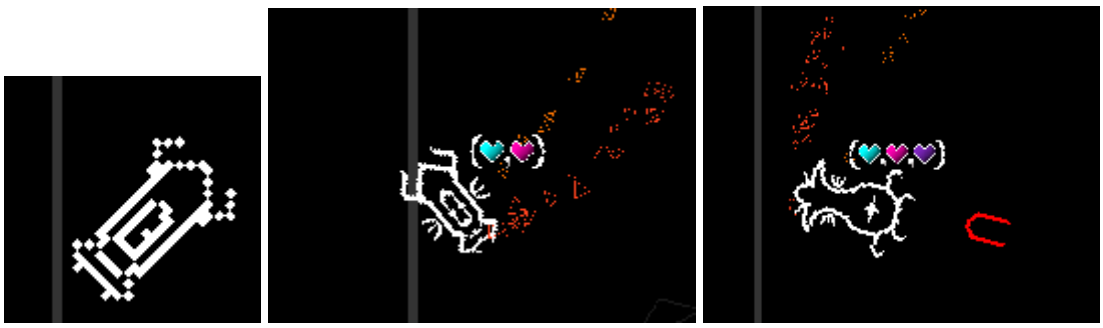
Los asteroides sirven para moldear la estructura del escenario de cada nivel (hasta que acaben destrozados), a veces su presencia sirve para representar **barreras** o puntos de interés.



9.5.2.2 Enemigo Básico

Este enemigo sí detecta al jugador. Si lo ve, puede atacar disparando mientras le persigue, y si está muy cerca del jugador empezará a orbitar alrededor de él. Si el jugador se acerca mucho a este, intentará mantener la distancia alejándose. Si no lo ve en su rango se queda quieto. Su comportamiento es más reactivo y directo.

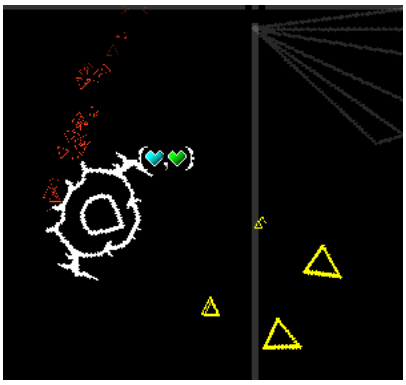
Tienen dos diseños, y según el nivel intentan así indicar y diferenciar lo que pueden disparar, pero no es siempre relevante, es cuestión de estética.



Boceto del Enemigo Básico 1 y diseño final de Enemigo Básico 1 y 2.

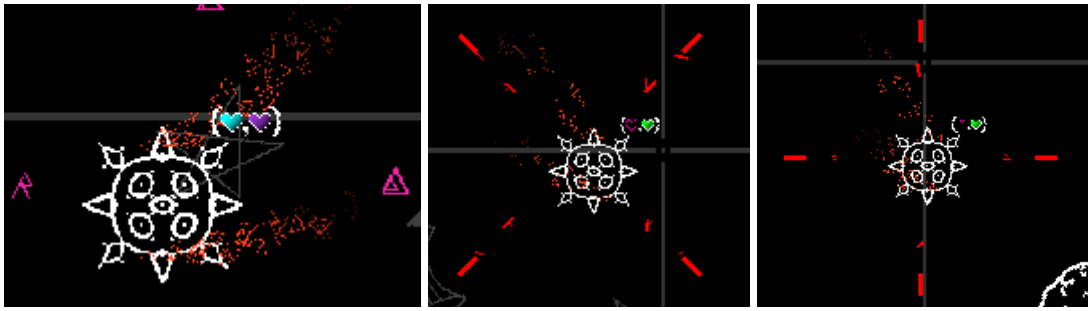
9.5.2.3 Enemigo con Cañón Dual

Es muy parecido al enemigo básico, pero es mucho más grande y lento (avanzando y orbitando). Además, mantiene una distancia con el jugador mayor. Por otro lado, es capaz de disparar dos proyectiles a la vez, por lo que hay que tener más cuidado esquivando.



9.5.2.3 Enemigo con Alternador de Fuego Cruzado

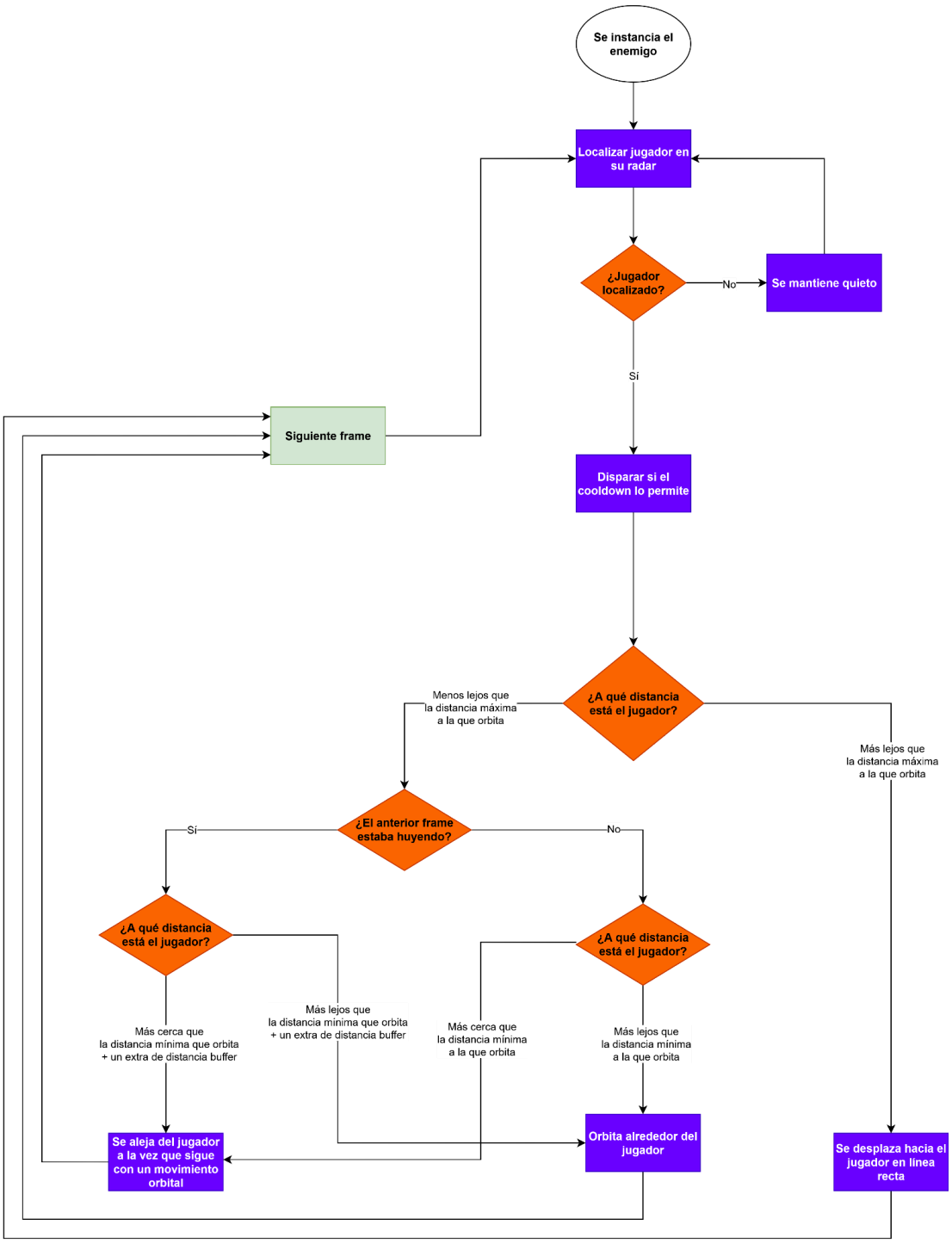
Este enemigo es más grande y lento que el Enemigo con Cañón Dual. No es capaz de rotar sobre sí mismo, por lo que, aunque avanza, nunca está apuntando hacia ti. Pero es capaz de disparar cuatro proyectiles a la vez, primero en las direcciones que forman una cruz y alterna después a las cuatro direcciones a una forma de equis. De ahí su nombre.



9.5.3 Diagrama de Flujo del Comportamiento

Todos los enemigos tienen el mismo patrón de comportamiento en sí, pero cada uno luego tiene ajustados ciertos valores acorde a qué tipo es. Ya sea la velocidad de movimiento, la cadencia de disparo, la máxima y mínima distancia a la que están del jugador, etc.

Este es el diagrama de flujo que todos los enemigos (menos los asteroides) presentan:

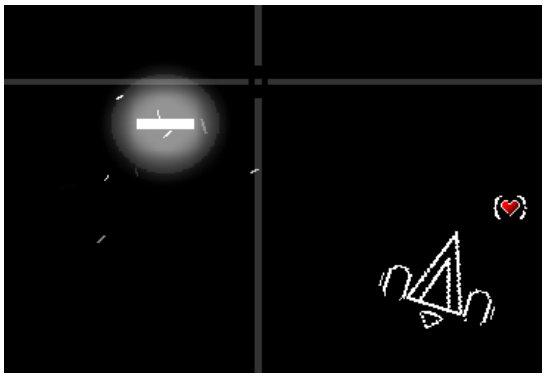


9.6 Items

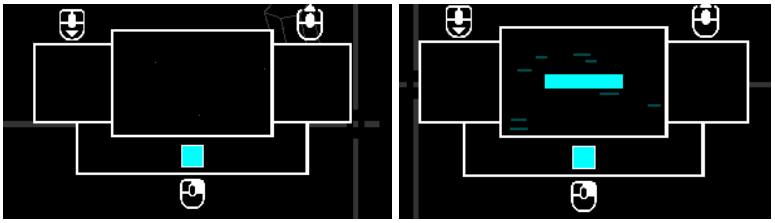
A veces, en los niveles podemos encontrar diferentes ítems, es decir, objetos que podemos recoger y que otorgan al jugador un efecto al ser recogidos. El propósito de los ítems es el de que a medida que se avanza por un nivel se obtenga más armamento acorde al progreso de este.

9.6.1 Item de Nuevo Proyectil

Este ítem otorga al jugador un nuevo tipo de proyectil con el que podrá disparar.
El nuevo proyectil se agrega en la interfaz de proyectiles (7.2) una vez es recogido.



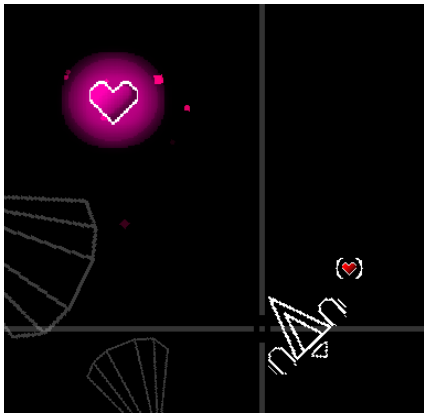
Ejemplo de ítem que otorga el proyectil de Diferencia al arsenal.



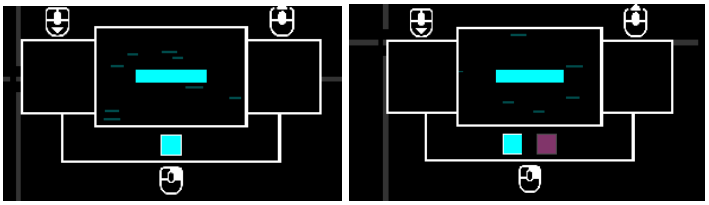
Interfaz de proyectiles antes y después de recoger el ítem.

9.6.2 Item de Nuevo Color para Proyectil

Este ítem otorga al jugador un nuevo color con el que podrá disparar sus proyectiles.
El color se agrega en la interfaz de proyectiles (7.2) una vez es recogido.



Ejemplo de ítem que otorga el color magenta al arsenal.



Interfaz de proyectiles antes y después de recoger el ítem.

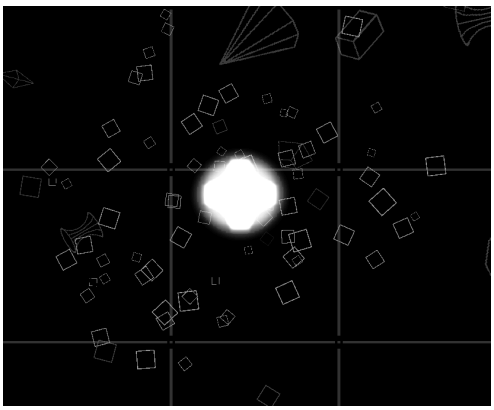
9.7 Elementos de Entorno

En cada nivel podemos encontrar diferentes elementos que forman parte del escenario y que sirven para diferentes cosas.

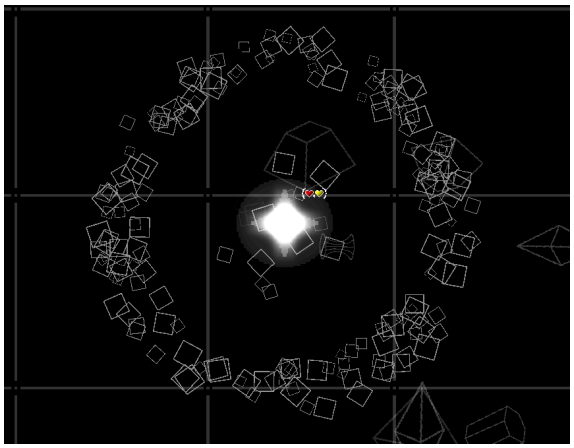
9.7.1 Portal de Spawn del Jugador

Este portal es una animación que aparece justo al iniciar un nivel. Es un destello que empieza a brillar cada vez más acompañado de una partículas que giran orbitando hacia el centro (formando una espiral animada) y acompañado de un sonido de como si fuese una fuente de energía cargándose.

Una vez pasan unos segundos, el jugador es visible, el destello empieza a atenuarse y las partículas dejan de emitirse. Dando paso a un aro de partículas que se expanden hacía afuera como una explosión acompañadas de un sonido que lo refleja.



Animación del portal antes de aparecer el jugador.
Nota: debido a que es una imagen, las partículas parecen no formar una espiral, pero se mueven como una.



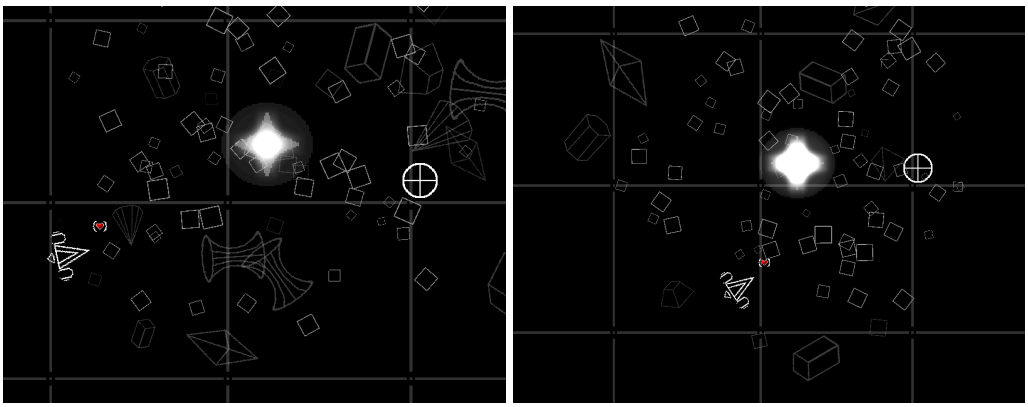
Animación final de explosión una vez el jugador aparece.

9.7.2 Portal de Fin de Nivel

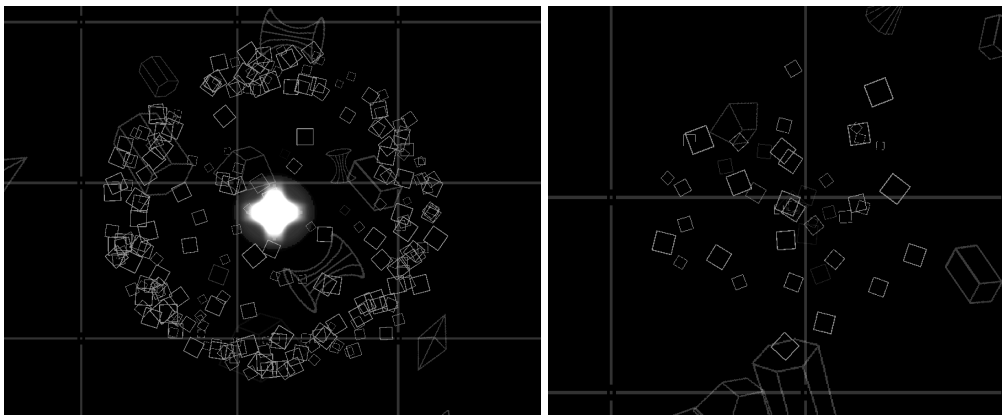
Este portal es similar al de spawn, lo que ocurre es que, en lugar de ser una animación temporal, permanece de manera constante en el escenario una vez aparece.

Las partículas que giran orbitando hacia el centro lo hacen en sentido antihorario a diferencia del otro portal. Además, el destello del centro tiene una animación constante de parpadeo (alternando el brillo).

El jugador puede interactuar con este acercándose al centro, esto ocasiona que el portal desaparezca junto al jugador con una animación de explosión como la que hace el otro portal al final de la animación y, tras esto, el nivel se completa.



Diferentes estados del destello animado.



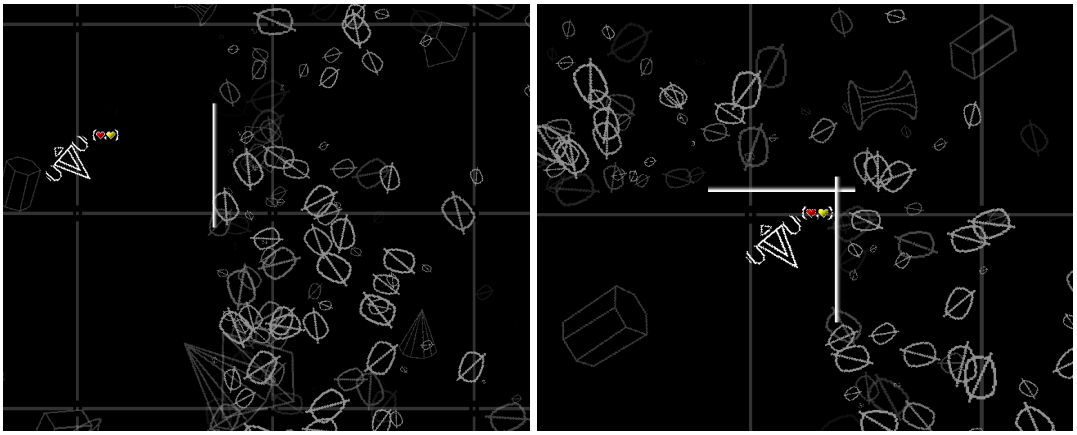
Animación una vez el jugador entra en el portal.

9.7.3 Límites del Mapa

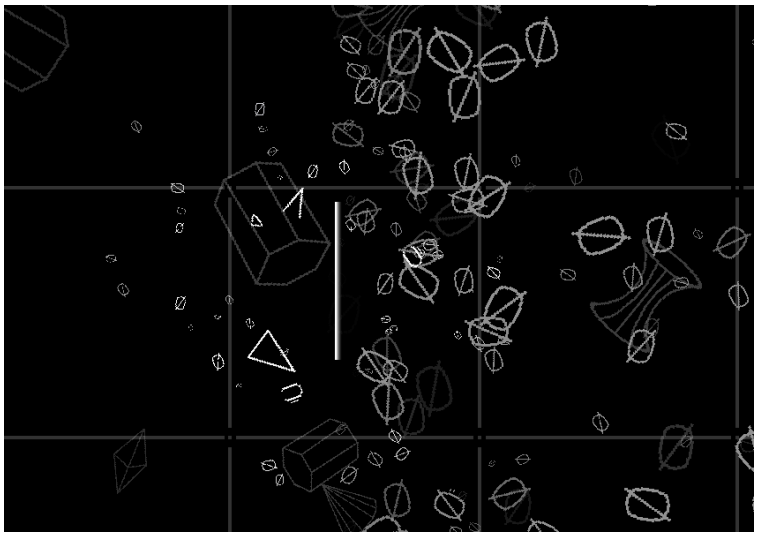
Cada nivel tiene unos límites que no se pueden exceder con el jugador. Estos están representados por un campo enorme de partículas que son el símbolo del conjunto vacío. Se les llama en el juego por el nombre de ***límites de vacío***.

Debido a que son solo partículas, hay una línea blanca por cada “pared” de los límites que sigue al jugador para darle referencia de donde está el límite exacto.

Una vez el jugador toca una de estas líneas, muere al instante, pues se le aplica un daño por contacto de tipo Intersección con el conjunto vacío.



Ejemplo de una y de dos paredes límite.



El jugador muriendo por un límite.
Se pueden apreciar partículas de conjunto vacío en la animación de muerte.

10. Niveles

El juego presenta un total de cinco niveles. Cada nivel solo es jugable si se ha completado anterior, para así restringir al jugador a aprender de manera progresiva.
Al principio del juego solo está disponible el nivel 1.

Cada nivel pretende enseñar ciertos aspectos del juego, ya sean los controles, tipos de proyectil, enemigos u otras mecánicas relevantes. Cada uno tiene su propia canción, que pretende transmitir el estilo y vibra del nivel.

Todo nivel comienza desde un cierto punto de partida en el mapa (siempre es en el centro), donde el portal de spawn genera al jugador; y se completa cuando el jugador accede a un portal de fin de nivel, el cual a veces no aparece hasta cierto momento en el nivel y otras veces se encuentra bloqueado desde el principio por asteroides que el jugador no puede destruir en un comienzo hasta que no consigue suficiente arsenal.

Si el jugador muere en un nivel, se muestra la pantalla de Game Over y este debe decidir si intentarlo de nuevo, lo que reinicia el nivel desde el principio.

Cada nivel puede jugar tantas veces como se quiera, no tiene puntos de control (*checkpoints*) y no son demasiado largos.

Todos los niveles tienen los mismos límites del mapa (**9.7.3**), donde el tamaño de este no varía entre cada uno.

El icono de cada nivel en el menú de selección de niveles es siempre el nuevo enemigo que aparece en ese nivel. Además, el nombre del nivel siempre es un juego de palabras donde se hace referencia de cierta forma a lo que ocurre en él junto a que se dice el nombre de uno de los proyectiles.

10.1 Nivel 1: Origin of Difference

Este nivel es un tutorial de los controles básicos del juego.

Enseña al jugador a:

- **Controles**: Mover la nave, aumentar velocidad, apuntar, disparar, cambiar de color para los proyectiles.
- **Tipos de proyectiles:** Comienza sin ninguno, pero después se le otorga y enseña la Diferencia, el más básico.
- **Colores de Proyectiles:** Comienza con el azul, aunque hasta que no tiene la Diferencia no puede disparar. Luego se le otorga el magenta.
- **Enemigos:** Se presentan los Asteroides, útiles para aprender a disparar y entender cómo los proyectiles les afectan. También se explican los límites del mapa.
- **Salud:** Se explica al jugador el concepto de salud en el juego y cómo puede morir. En este nivel solo tiene como salud el corazón rojo.

Canción del nivel: *Electric Dreams by Scott Buckley (CC-BY 4.0 www.scottbuckley.com.au)*

Icono del nivel en el menú de selección.

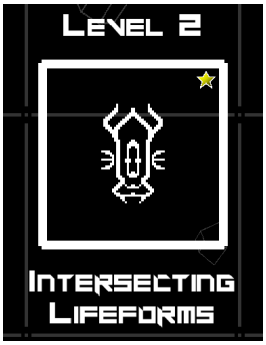
10.2 Nivel 2: Intersecting Lifeforms

Presenta los primeros enemigos que pueden dispararte, siendo el primer nivel donde se puede morir proyectiles y no por contacto. También es la primera vez que el jugador maneja más de un tipo de proyectil a la vez para poder atacar, siendo necesario aprender a combinarlos.

Enseña al jugador a:

- **Controles**: Cambiar de tipo de proyectil.
- **Tipos de proyectiles:** Comienza con la Diferencia, y después se le otorga y explica la Intersección. Además, los enemigos pueden disparar Diferencias y Uniones, por lo que se explica también la Unión para que el jugador pueda recuperar salud.
- **Colores de Proyectiles:** Siempre es azul y magenta desde el comienzo.
- **Enemigos:** Se presentan los enemigos básicos (el diseño 1) y más asteroides.
- **Salud:** El jugador tiene un corazón rojo y amarillo en el nivel.

Canción del nivel: *Neon by Scott Buckley (CC-BY 4.0 www.scottbuckley.com.au)*



Icono del nivel en el menú de selección.

10.3 Nivel 3: The Union of Power

Este nivel es un refuerzo del nivel 2, pero con mayor dificultad porque son los enemigos los que hacen uso de la Intersección.
Al principio del nivel, el jugador está atrapado en un área reducida por los asteroides y donde debe lidiar con poder eliminar todos los enemigos que van apareciendo.

Después debe salir del área gracias al color verde y dar un recorrido por el resto del mapa hasta el portal final mientras encuentra más enemigos.

Enseña al jugador a:

- **Tipos de proyectiles:** Solo se juega con la Diferencia
- **Colores de Proyectiles:** Comienza con el azul y magenta, luego se otorga el verde.
- **Enemigos:** Se presenta el segundo diseño del enemigo básico, pero también hay de diseño 1 y asteroides.
- **Salud:** El jugador tiene un corazón rojo, amarillo y morado en el nivel.

Canción del nivel: *Password Infinity by Evgeny Bardyuzha (www.pixabay.com)*

Icono del nivel en el menú de selección.

10.4 Nivel 4: Complementary Decisions

Un nivel muy especial donde se enseñan a la vez dos nuevos tipos de proyectiles que congenian muy bien. Al principio se enseña al jugador a esquivar asteroides que orbitan y al final se le acaba enseñando como destruir enemigos combinando estos proyectiles.

Enseña al jugador a:

- **Tipos de proyectiles:** Comienza sin ninguno, pero después se le otorga y enseña el Complemento y la Unión. Al final se le proporciona también la Intersección.
- **Colores de Proyectiles:** Comienza con azul, magenta y verde hasta final.
- **Enemigos:** Se presentan los enemigos de cañón dual y los asteroides pudiendo orbitar. También hay enemigos básicos.

Canción del nivel: *Toads And Frogs by Electronic Senses (www.pixabay.com)*

Icono del nivel en el menú de selección.

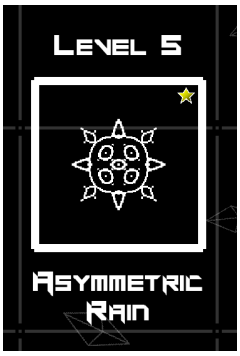
10.5 Nivel 5: Asymmetric Rain

Nivel final, aporta un desafío complicado a la vez que te enseña las últimas mecánicas. Destaca el hecho de que hay muchos enemigos en el nivel, ocurriendo que el jugador puede acabar atrapado entre tanto asteroides y proyectiles.

Enseña al jugador a:

- **Tipos de proyectiles:** Comienza con el Complemento, al momento proporciona y explica la Diferencia Asimétrica. Y al final también se usa la Intersección.
- **Colores de Proyectiles:** Comienza con el azul y magenta hasta el final.
- **Enemigos:** Se presentan los enemigos con alternador de fuego cruzado, aunque aparecen también todos los demás (excepto el enemigo básico de diseño 2).

Canción del nivel: *For Three Reasons by Electronic Senses (www.pixabay.com)*



Icono del nivel en el menú de selección.

11. Sistema de Guardado

El sistema de guardado es automático y se divide en dos partes:

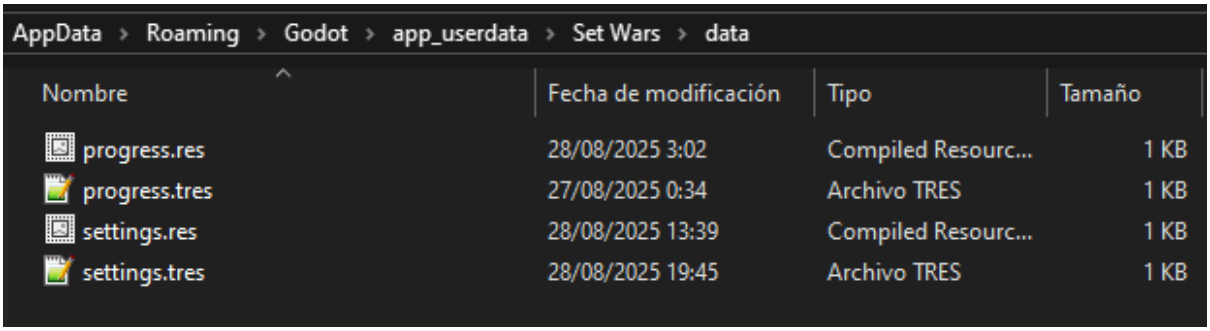
- **Progreso:** Se guardan los niveles que se han completado (para mostrar la estrella en el menú de selección de nivel) y los niveles que están desbloqueados.
  Estos datos se actualizan solos cuando el jugador completa alguno de los niveles, añadiendo a la lista los nuevos datos.
  No hay forma de borrar estos datos dentro del juego, una vez completado/desbloqueado algo no se puede borrar a menos que se manipule el archivo de guardado.

- **Ajustes:** Se actualizan cada vez que el jugador cambia algo en el menú de ajustes. Estos datos incluyen: el volumen de música y efectos de sonido (sfx), la sensibilidad de la mirilla usando el mando, el modo de ventana del juego, el idioma y si la interfaz de salud seguidora se muestra o no.

Para el sistema de guardado se utiliza el método nativo de Godot, donde en una dirección específica "*oculta*" se guardan los datos del juego. Dentro del directorio se crea una carpeta llamada *data* la cual crea dentro los dos archivos de datos.

Cuando se está jugando en modo debug (desde el editor) el juego crea y utiliza archivos *.tres*, nativos de Godot que son fáciles de leer y editar.

Pero si se juega en modo producción, el juego usa archivos *.res*, que son lo mismo que los *.tres* pero con formato binario para dificultar la lectura y edición de un usuario externo además de que Godot los manipula más rápido.



Ruta de los archivos en Windows.

Apéndice C
Manual de Instalación

Este manual explica cómo instalar y ejecutar **Set Wars: Cardinal Bounds**. Se ofrecen dos versiones del juego, una para Windows y otra para Linux, ambas disponibles para su descarga a través de Google Drive.

Antes de comenzar, asegúrate de tener acceso al enlace proporcionado; si el enlace está protegido, necesitarás permisos para visualizar y descargar los archivos.

1. **Descargar el juego:** Accede al siguiente enlace de Google Drive para descargar la versión correspondiente a tu sistema operativo.

   Windows https://drive.google.com/file/d/18CJ60OGR53H0jXEkl3dAy39g2EFqK62S/view?usp=sharing

   Linux: https://drive.google.com/file/d/1uaOAe3d4xRd4yHg0iR3COL5V6C3tHiX4/view?usp=sharing

2. **Extraer el contenido:** El archivo descargado está comprimido (*.rar*), haz clic derecho sobre él, selecciona una opción para extraer su contenido y elige una ubicación donde dejar los archivos extraídos.

3. **Ejecutar el juego:** Una vez extraído, localiza el archivo ejecutable correspondiente (*Set Wars.exe* en Windows o *Set Wars.x86_64* en Linux). Haz doble clic sobre él para iniciar el juego.

Además, hay que tener en cuenta que algunos antivirus pueden marcar los archivos ejecutables como potencialmente peligrosos debido a que no son ampliamente reconocidos. Si esto sucede, puedes añadir el archivo a la lista de exclusiones de tu antivirus o marcarlo como seguro. Si el sistema operativo solicita permisos para ejecutar el archivo, asegúrate de concederlos.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA