



Universidad de Jaén

Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

REMAKE DEL VIDEOJUEGO MARIO BROS

Alumno

Alejandro Ramos Gallego

Tutor

D.Carlos Javier Ogayar Anguita
(Departamento de Informática)

[mes], [año]



Universidad de Jaén

Departamento de Informática

Don D.Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: **‘Remake del videojuego Mario Bros’**, que presenta Don Alejandro Ramos Gallego, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, [mes] de [año]

El alumno:

El tutor:

Alejandro Ramos Gallego

D.Carlos Javier Ogayar Anguita

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a los profesores que me han impartido conocimientos a lo largo de mi carrera, contribuyendo significativamente a mi formación académica y profesional. Un agradecimiento especial a mi familia, que ha sido mi soporte en los momentos más desafiantes, brindándome amor y apoyo incondicional en cada paso que he dado. Por último, mi gratitud a mi tutor, quien con su guía experta y dedicación ha hecho de este Trabajo de Fin de Grado una experiencia verdaderamente enriquecedora y exitosa.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad <small>(solo TFG)</small>	Sin especialidad
Mención <small>(solo TFG)</small>	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Alejandro Ramos Gallego
Fecha de asignación	Haga clic aquí o pulse para escribir una fecha.
Descripción corta	Desarrollo de un remake del clásico videojuego "Super Mario Bros" utilizando Unity, centrado en la implementación de gráficos mejorados en 2D

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
<p><i>Estas normas se han utilizado como base o referencia para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como proyecto la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.</i></p>	

Contenido

1	Introducción	13
1.1	Propósito del Proyecto	13
1.2	Alcance del Proyecto	14
2	Herramientas y tecnologías.....	16
2.1	Unity - Motor de Desarrollo	16
2.2	Herramientas Adicionales	17
3	Diseño y planificación	19
3.1	Metodología de Desarrollo	19
3.2	Planificación Temporal.....	21
3.3	Estimaciones de recursos y costes.....	22
4	Diseño inicial	24
4.1	Visión General del Juego	24
4.2	Mecánicas del Juego	25
4.3	Personajes	28
4.4	Power-Ups y Objetos	32
4.5	Interfaz de Usuario	33
5	Desarrollo del juego.....	37
5.1	Configuración Inicial.....	37
5.1.1	Configuración del Proyecto en Unity	37
5.2	Desarrollo de Características.....	38
5.2.1	Creación y Manipulación de Sprites	38
5.2.2	Implementación de Físicas y Colisiones	41
5.2.3	Animación de Personajes y Objetos.....	45
5.2.4	Creación y Gestión de Enemigos	49
5.2.5	Implementación de Transformaciones y Power-ups	55
5.2.6	Gestión de Items, Poderes Especiales y Bloques.....	57
5.2.7	Implementación del Sistema de Puntuación y Audio.	65
5.2.8	Desarrollo de la Interfaz de Usuario y Temporizador.....	74
5.3	Enriquecimiento y Expansión de Juego	78
5.3.1	Construcción de Niveles	78
5.3.2	Gestión de Vidas y Sistema de Respawn	92
5.3.3	Sistema de Guardado.....	96
5.3.4	Implementación y Gestión del Menú de Inicio.....	98
5.3.5	Conexión entre Zonas del nivel y Transición de Niveles	100
5.3.6	Configuración de la Cámara y Ajustes Visuales	105
6	Apéndices.....	110
6.1	Guía original del Trabajo Fin de Título.....	110
6.2	Manuales de usuario.....	110
7	Bibliografía	111

Índice de ilustraciones

Ilustración 2.1. Herramientas Adicionales.....	18
Ilustración 3.4. Goomba	29
Ilustración 3.7. Koopa Con Alas	30
Ilustración 4.9. HUD	33
Ilustración 4.10. Pantalla Inicio	34
Ilustración 4.11. Pantalla – Nivel Superado	34
Ilustración 4.12. Pantalla – Perder Vida.....	35
Ilustración 4.13. Pantalla – Game Over	35
Ilustración 4.14. Alertas Visuales.....	36
Ilustración 5.1. Versión Unity	37
Ilustración 5.2. Estructura del Proyecto	38
Ilustración 5.3. Texture Type	38
Ilustración 5.4. Sprite Editor	39
Ilustración 5.5. Filter Mode	40
Ilustración 5.6. Sprite Renderer	40
Ilustración 5.7. Rigidbody 2D.....	41
Ilustración 5.8. Tipo de Rigidbody 2D	41
Ilustración 5.9. Physics Material 2D.....	43
Ilustración 5.10. Control de Movimiento.....	43
Ilustración 5.11. Mecánica de Salto	44
Ilustración 5.12. Detección del suelo	44
Ilustración 5.13. Animation	46
Ilustración 5.14. Animator	46
Ilustración 5.15. Configuración Animación.....	46
Ilustración 5.16. Integración con el Sistema de Animación	47
Ilustración 5.17. Control Dinámico de Animaciones.....	47
Ilustración 5.18. Transición de Estados - Animator	47
Ilustración 5.19. Animación Agachado.....	48
Ilustración 5.20. Animación Agachado.....	48
Ilustración 5.21. Animation Enemy (Goomba).....	49
Ilustración 5.22. Animator Enemy (Goomba)	49
Ilustración 5.23. Componentes Físicos del Enemy (Goomba)	50
Ilustración 5.24. Clase 'Enemy'	50
Ilustración 5.25. Gestión de Colisiones Enemy.....	51
Ilustración 5.26. Clase Derivada de Enemy (Goomba)	51
Ilustración 5.27. Utilización de las paletas del Tilemap	53
Ilustración 5.28. Utilización de las paletas del Tilemap	53
Ilustración 5.29. Utilización de las paletas del Tilemap	54
Ilustración 5.30. Animator Mario	55
Ilustración 5.31. Gestión Power-Ups	56
Ilustración 5.32. Prefab Variant	57
Ilustración 5.33. AnimacionesSprites.cs	58

Ilustración 5.34. Parámetros Bloque.....	59
Ilustración 5.35. Parámetros Bola de Fuego.....	60
Ilustración 5.36. Prefabs Bola de Fuego.....	60
Ilustración 5.37. Modo Invencible	61
Ilustración 5.38. Animación de Invencibilidad	62
Ilustración 5.39. Detección de Enemigos – Items encima del Bloque	63
Ilustración 5.40. Interacción con la Planta Piraña	64
Ilustración 5.41. Colisiones cuando Mario es golpeado	64
Ilustración 5.42. Sprite de la bandera	65
Ilustración 5.43. Animator - Escalar	65
Ilustración 5.44. Condicion de la animación Escalar	66
Ilustración 5.45. Función para Bajar de la bandera.....	66
Ilustración 5.46. Función para Saltar de la bandera.....	67
Ilustración 5.47. Score Manager	68
Ilustración 5.48. Alturas a las que Mario puede tocar la bandera.....	68
Ilustración 5.49. Calculo de la altura y asignación de puntos de la bandera	69
Ilustración 5.50. Efecto puntos	70
Ilustración 5.51. Tipos de Carga de Audio	71
Ilustración 5.52. Tipos de compresión de Audio	72
Ilustración 5.53. Audio Source.....	72
Ilustración 5.54. Transiciones musicales	73
Ilustración 5.55. Musica – Modo invencible	73
Ilustración 5.56. Render Mode - Canvas	74
Ilustración 5.57. Canvas Scaler	74
Ilustración 5.58. Anchors Presets	75
Ilustración 5.59. Vista Previa de la UI	75
Ilustración 5.60. Vista previa monedas - UI	76
Ilustración 5.61. Función Actualizar Tiempo en el HUD	77
Ilustración 5.62. Función para convertir cada segundo restante en puntos.....	77
Ilustración 5.63. Utilización de las paletas del Tilemap	78
Ilustración 5.64. Utilización del Composite Collider 2D	79
Ilustración 5.65. Mapa nivel 1	80
Ilustración 5.66. Mapa nivel 1 - Tilemap	80
Ilustración 5.67. Código Bloques Invisibles.....	81
Ilustración 5.68. Cambio de Seta Mágica a Flor de Fuego	81
Ilustración 5.69. GameObject Brush	82
Ilustración 5.70. Mapa nivel 2	83
Ilustración 5.71. Mapa nivel 2 - Tilemap	83
Ilustración 5.72. Platform Efecto 2D	84
Ilustración 5.73. MovimientoPlataforma.cs	84
Ilustración 5.74. Mapa nivel 3	86
Ilustración 5.75. Mapa nivel 3 – Tilemap	86
Ilustración 5.76. Mapa nivel 4	87
Ilustración 5.77. Mapa nivel 4 - Tilemap	87
Ilustración 5.78. Daño causado por colisionar con Lava	87
Ilustración 5.79. Barras de Fuego.....	88

Ilustración 5.80. HachaFinal.cs.....	88
Ilustración 5.81. Finalización del Nivel – Interacción con Toad.....	89
Ilustración 5.82. Control de Bowser.....	90
Ilustración 5.83. Gestión muerte y salud de Bowser.....	91
Ilustración 5.84. Zona Muerte.....	92
Ilustración 5.85. GameManager.....	93
Ilustración 5.86. CheckPoint.....	93
Ilustración 5.87. Uso de 'DontDestoryOnLoad()'.....	94
Ilustración 5.88. Uso de 'SceneManager.LoadScene()'.....	94
Ilustración 5.89. Modo Invencible.....	95
Ilustración 5.90. Puntuación Máxima – ScpreManager.cs.....	96
Ilustración 5.91. Recuperación de los valores de nivel y mundo.....	97
Ilustración 5.92. Menu Inicial.....	98
Ilustración 5.93. MenuInicial.cs.....	99
Ilustración 5.94. BotonMenu.cs.....	99
Ilustración 5.95. Prefabs ConexionZonas_ y Zonas_.....	100
Ilustración 5.96. ConexionZonas.cs.....	101
Ilustración 5.97. Zona.cs.....	101
Ilustración 5.98. Castillo.cs.....	102
Ilustración 5.99. Manejo de la Transición.....	103
Ilustración 5.100. Escena de Transición.....	103
Ilustración 5.101. Carga del nuevo nivel.....	104
Ilustración 5.102. AspectRatioCamara.cs.....	105
Ilustración 5.103. SegumientoCamara.cs.....	106
Ilustración 5.104. Collider – Configuración Camara.....	107
Ilustración 5.105. Eliminación de Koopa de la Escena.....	108
Ilustración 5.106. Eliminación de Enemigos e items de la Escena.....	108
Ilustración 5.107. Activación de Enemigos en la Escena.....	109

Índice de tablas

Tabla 3.1. Planificación Temporal	21
Tabla 3.2. Costes Recursos Humano	22
Tabla 3.3. Costes Recursos Materiales y Software	22
Tabla 3.4. Coste total del proyecto	23
Ilustración 3.3. Bowser	29
Tabla 4.1 Puntos obtenidos por derrotar a cada enemigo	31
Tabla 4.2 Puntos obtenidos por recoger un power-up u objeto.....	32

1 INTRODUCCIÓN

1.1 Propósito del Proyecto

El propósito de este Trabajo de Fin de Grado es desarrollar un remake del icónico videojuego Super Mario Bros en 2D, utilizando el motor de juego Unity. Este proyecto tiene como objetivo principal aplicar y demostrar las competencias adquiridas en el campo de la Ingeniería Informática, específicamente en el desarrollo de software, diseño gráfico y programación orientada a objetos. Además, se busca explorar las capacidades de Unity como herramienta de desarrollo de videojuegos, enfocándose en la reproducción fiel de las mecánicas, estética y niveles del juego original, al tiempo que se incorpora mejoras de diseño que enriquezcan la experiencia del usuario.

Objetivos específicos del proyecto:

- **Recrear la Jugabilidad Clásica:** Implementar las mecánicas fundamentales de Mario Bros, asegurando que el juego no solo se sienta como una réplica fiel del original en términos de controles y respuesta, sino que también capture la esencia y el desafío que definieron el juego.
- **Modernizar Elementos Gráficos:** Aunque el juego mantendrá un estilo visual que respete el original, se utilizarán las herramientas modernas de Unity para mejorar los gráficos, animaciones y efectos visuales, adaptándolos a las resoluciones actuales sin perder el encanto retro.
- **Documentación Completa del Proceso de Desarrollo:** Detallar cada fase del desarrollo, desde la planificación y diseño hasta la implementación y pruebas, proporcionando una base documental sólida.

1.2 Alcance del Proyecto

El alcance de este Trabajo de Fin de Grado se centra en desarrollar un remake del videojuego Super Mario Bros, limitando el contenido al primer mundo, que comprende los cuatro primeros niveles.

El alcance de este proyecto incluye:

1. Desarrollo de Niveles:

Recreación de los Cuatro Primeros Niveles: El proyecto incluirá el diseño y desarrollo de los primeros cuatro niveles del mundo 1 de Super Mario Bros, cada uno con sus características únicas y desafíos, incluyendo el nivel subterráneo y el nivel del castillo final .

2. Mecánicas de Juego

Implementación de Mecánicas clásicas: Se implementarán las mecánicas básicas de movimiento de Mario, incluyendo correr, saltar y agacharse, así como las interacciones con enemigos, objetos y el entorno de juego.

Sistemas de Puntuación y Progreso: Se reproducirán los sistemas de puntuación basados en la recolección de monedas, eliminación de enemigos y finalización de niveles, junto con la gestión de vidas y el progreso del jugador a través de los niveles.

Interacciones con Enemigos y Objetivos: Se incluirán interacciones clásicas con enemigos como los Goombas, distintos tipos de Koopas, Planta Piraña y Bowser, así como con objetos como bloques de interrogación, bloques de ladrillos, y power-ups(setas, flor de fuego y estrella)

3. Elementos Audiovisuales

Gráficos y Animaciones Modernizadas: Aunque el estilo visual mantendrá un homenaje al arte pixelado original , se utilizará las herramientas de Unity para mejorar las animaciones y los gráficos, asgeurnado que sean adecuados para dispositivos de alta resolución.

Efectos de Sonido y Música: Se recrearán los efectos de sonido icónicos y la música de fondo característica de cada nivel, optimizados para una reproducción de alta calidad.

4. Documentación y Pruebas

Pruebas de Juego: Se realizarán pruebas para asegurar que el juego funciona correctmanete en términos de mecánica, rendimiento y bugs.

2 HERRAMIENTAS Y TECNOLOGÍAS

2.1 Unity - Motor de Desarrollo

Unity es un motor de desarrollo de videojuegos ampliamente conocido y utilizado en la industria por su versatilidad y robustez, lo que lo hace una elección ideal para la creación de juegos en 2D. Para este Trabajo Fin De Grado, he seleccionado Unity debido a varias de sus características clave que son particularmente adecuadas para este proyecto:

1. Facilidad de Uso:

Interfaz Intuitiva: Unity ofrece una interfaz de usuario gráfica que es intuitiva y amigable para desarrollares de todos los niveles, lo que facilita la creación, prueba y depuración de juegos.

Amplia Documentación: Unity cuenta con una extensa documentación en línea, incluyendo tutoriales y foros de discusión, lo que proporciona un excelente soporte durante el aprendizaje y desarrollo [1].

2. Herramientas Integradas

Sistema de Animación Avanzado: Las herramientas de animación de Unity permiten crear animaciones complejas y fluidas, lo que mejora la calidad y el dinamismo visual del juego.

Sistema de Físicas Incorporado: Unity incluye un motor de físicas que facilita la simulación de movimientos realistas y colisiones, esencial para replicar y mejorar las mecánicas de juego de Mario Bros.

3. Desarrollo de Scripts

Soporte para C#: Unity utiliza C# para scripting, un lenguaje de programación moderno y potente, que permite escribir scripts limpios y eficientes para controlar la lógica del juego, eventos, y comportamiento de los personajes.

4. Recursos y Assets:

Unity proporciona acceso a una extensa biblioteca de recursos en su Asset Store, donde se pueden encontrar desde scripts hasta modelos y texturas que pueden ser utilizados para acelerar el proceso de desarrollo.

2.2 Herramientas Adicionales

Además de Unity, para el desarrollo del videojuego he utilizado otras herramientas para diversas tareas del proceso de creación del juego. Estas herramientas adicionales incluyen Visual Studio para scripting, Paint para diseño gráfico y GIMP para edición más detallada de gráficos.

1. Visual Studio

Integración con Unity: Visual Studio se integra perfectamente con Unity, proporcionando un entorno robusto para escribir, depurar y gestionar scripts en C#.

Herramienta de Desarrollo Avanzada: Visual Studio es conocido por su potente editor de código, capacidad de depuración y características de autocompletado inteligente, lo que lo convierte en una herramienta indispensable para el desarrollo de scripts complejos necesarios para las mecánicas y la lógica del juego [2].

2. Paint

Edición gráfica sencilla: Paint lo he utilizado en este proyecto para tareas rápidas y simples de edición gráfica, para modificar y crear sprites y texturas de manera sencilla.

Accesibilidad y Facilidad de Uso: Paint es una herramienta accesible que forma parte de los sistemas operativos de Windows, conocido por su interfaz intuitiva y fácil manejo, lo que lo hace ideal para los ajustes rápidos y tareas de diseño gráfico sin complicaciones [3].

3. GIMP

Edición Gráfica Avanzada: Para la edición más detallada y refinada de sprites, he utilizado GIMP, un programa de manipulación de imágenes gratuito y de código abierto.

Herramientas de Diseño Profesional: GIMP proporciona herramientas avanzadas para la edición de imágenes, como capas, máscaras, filtro y pinceles, facilitando la modificación de los sprites para el juego [4].



Ilustración 2.1. Herramientas Adicionales

3 DISEÑO Y PLANIFICACIÓN

3.1 Metodología de Desarrollo

Para el desarrollo del remake de Super Mario Bros, he optado por una combinación de la metodología **Scrum**, que es una forma de metodología ágil, con un enfoque estructurado en el desarrollo incremental e iterativo. Scrum se caracteriza por su flexibilidad y capacidad para adaptarse a cambios rápidos y continuos en los requisitos del proyecto [5].

Implementación de la Metodología Scrum:

Scrum se ha utilizado para estructurar el desarrollo en sprints cortos, cada uno enfocado en distintos componentes del juego, como la configuración inicial, el desarrollo de características, y la expansión del juego. La organización en sprints permitió revisiones periódicas del progreso y ajustes continuos del plan de desarrollo.

Fases de Desarrollo Detalladas:

1. Configuración Inicial (Sprint 1):

Configuración del Proyecto en Unity: Se seleccionó Unity 2021.3.38f1 por su estabilidad y soporte. La estructura del proyecto se organizó según las recomendaciones estándar de Unity para facilitar el acceso y la gestión de recursos.

Selección del Template: Se optó por el template 2D de Unity, preconfigurando el entorno para optimizar el diseño y desarrollo en dos dimensiones.

2. Desarrollo de Características (Sprints 2-8):

Creación y Manipulación de Sprites: Diseño y ajuste de sprites usando herramientas como Paint y GIMP.

Implementación de Físicas y Colisiones: Uso de Rigidbody y Colliders para agregar realismo físico a las interacciones y movimientos.

Animación de Personajes y Objetos: Utilización de Unity Animator para animaciones dinámicas y atractivas.

Creación y Gestión de Enemigos: Desarrollo de comportamientos enemigos utilizando clases base y derivadas para diferentes tipos de enemigos.

Implementación de Transformaciones y Power-ups: Gestión de estados de Mario mediante Animator para activar cambios con power-ups.

Gestión de Items y Bloques: Uso de prefabs para la implementación eficiente y uniforme de elementos interactuables en los niveles.

3. Enriquecimiento y Expansión del Juego

Construcción de Niveles: Uso de Tilemap para el diseño eficiente y flexible de niveles.

Sistema de Guardado: Implementación de PlayerPrefs para la persistencia de datos esenciales como puntuaciones y progreso del jugador.

Gestión del Menú de Inicio: Diseño de la interfaz de usuario para facilitar la navegación inicial y la selección de opciones de juego.

Conexión entre Zonas del Nivel y Transición de Niveles: Implementación de tuberías como transiciones entre diferentes áreas y manejo de cambios de nivel.

4. Evaluación y Adaptación:

Al final de cada sprint, se llevaba a cabo una revisión del progreso y una retrospectiva para identificar mejoras en los procesos y técnicas utilizadas.

Este enfoque iterativo permitió ajustes continuos en el plan de desarrollo, asegurando que el juego cumplía con las expectativas de calidad y jugabilidad.

3.2 Planificación Temporal

Fase de Desarrollo	Tareas Principales	Inicio	Fin	Duración
Configuración Inicial	Configuración del Proyecto, Selección del Template	2024-03-15	2024-03-16	10h
Desarrollo de Características				
Creación y Manipulación de Sprites	Diseño y ajuste de sprites	2024-03-17	2024-03-23	35h
Implementación de Físicas y Colisiones	Configuración de Rigidbody y Colliders	2024-03-24	2024-03-28	25h
Animación de Personajes y Objetos	Creación de clips de animación y configuración en Animator	2024-03-29	2024-04-05	40h
Creación y Gestión de Enemigos	Desarrollo y ajuste de comportamientos enemigos	2024-04-06	2024-04-12	35h
Implementación de Transformaciones	Gestión de power-ups y transformaciones de Mario	2024-04-13	2024-04-18	30h
Gestión de Items y Bloques	Implementación y configuración de items y bloques interactivos	2024-04-19	2024-04-22	20h
Sistema de Puntuación y Audio	Configuración del audio y sistema de puntuación	2024-04-23	2024-04-26	20h
Interfaz de Usuario y Temporizador	Desarrollo de HUD y manejo del temporizador	2024-04-27	2024-04-30	20h
Enriquecimiento y Expansión de Juego				
Construcción de Niveles	Uso de Tilemap para diseñar niveles	2024-05-01	2024-05-06	30h
Sistema de Guardado	Implementación del sistema de guardado de progresos	2024-05-07	2024-05-08	10h
Menú de Inicio	Diseño e implementación del menú de inicio	2024-05-09	2024-05-10	10h
Transiciones entre Niveles	Gestión de conexiones entre zonas y cambios de nivel	2024-05-11	2024-05-13	15h
Configuración de la Cámara	Ajustes en la cámara y seguimiento dinámico de Mario	2024-05-14	2024-05-17	20h

Tabla 3.1. Planificación Temporal

3.3 Estimaciones de recursos y costes

A continuación, se presenta una estimación detallada de los recursos y costes asociados al desarrollo del remake de Super Mario Bros utilizando Unity. Esta estimación incluye tanto los recursos humanos como materiales.

Recursos Humanos:

Descripción	Sueldo mensual	Coste (3 meses)
Desarrollador de videojuegos	1160 EUR	3480 EUR

Tabla 3.2. Costes Recursos Humano

Recursos Materiales y Software:

El ordenador utilizado tiene un coste de 800 euros, si asumo una amortización de 4 años, obtengo un coste de 16.67 EUR al mes.

Recurso	Descripción	Precio (mes)	Coste (3 meses)
Unity Personal	Student Plan	0 EUR	0 EUR
Visual Studio Community	Entorno de desarrollo integrado gratuito	0 EUR	0 EUR
Paint	Herramienta de diseño básico	0 EUR	0 EUR
GIMP	Programa de edición de imágenes avanzada	0 EUR	0 EUR
Ordenador Personal	Necesario para la realización del proyecto	16.67 EUR	50 EUR
Microsoft Office Word	Para documentación	5.75 EUR	17.25 EUR

Tabla 3.3. Costes Recursos Materiales y Software

Resumen de Costes Estimados:

Categoría	Coste Estimado Total
Recursos Humanos	3480 EUR
Recursos Materiales y Software	67.25 EUR
Total Estimado	3547.25 EUR

Tabla 3.4. Coste total del proyecto

4 DISEÑO INICIAL

4.1 Visión General del Juego

Título del Juego

- **Super Mario Bros Remake**

Género

- **Plataforma 2D:** El juego mantiene el género de plataformas en dos dimensiones, característico del original, enfocado en el control del personaje principal, Mario, que debe superar diversos obstáculos y enemigos para salvar a la Princesa Peach.

Estilo Visual

- **Pixel Art Mejorado:** Aunque el juego original es conocido por su estilo de pixel art, con este remake propongo una versión mejorada con sprites más detallados y escenarios más ricos visualmente, sin perder el encanto del arte pixelado original.

Temática

- **Aventura y Rescate:** La temática central gira en torno a la aventura de Mario a través de diferentes mundos, cada uno con su propia temática y desafíos, para rescatar a la Princesa Peach de las garras del villano Bowser.

Audiencia Objetivo

- **Jugadores de Todas las Edades:** Si bien el juego apela a la nostalgia de los jugadores que crecieron con el original en los años 80 y 90, también está diseñado para ser accesible y disfrutable para los jóvenes jugadores. La sencillez de las mecánicas y la universalidad de su historia hacen de este remake un juego adecuado para toda la familia.

Objetivo del Juego

- **Progresión Lineal a través de Niveles Desafiantes:** Los jugadores deben guiar a Mario a través de varios niveles, cada uno ofreciendo un incremento en dificultad, con el objetivo final de derrotar a Bowser y rescatar a la Princesa. Cada nivel está diseñado para poner a prueba las habilidades de salto, timing y estrategia del jugador, con varios secretos y atajos por descubrir.

4.2 Mecánicas del Juego

Para este remake de Super Mario Bros, he conservado numerosas mecánicas clásicas mientras he añadido varias innovaciones que enriquecen la experiencia de juego. El objetivo ha sido mantener la autenticidad del juego original al tiempo que se introduce una dimensión moderna a través de mejoras técnicas y funcionales.

Controles Básicos

- **Movimiento:** Mario puede moverse hacia la izquierda o derecha usando los controles direccionales, facilitando la exploración e interacción con el entorno y los enemigos.
- **Salto:** El salto es la mecánica central en este videojuego. Mario puede realizar saltos de diferentes alturas dependiendo de cuánto tiempo el jugador presione el botón de salto. Esto permite una variedad de maniobras y estrategias al enfrentar obstáculos o enemigos.
- **Correr:** Manteniendo presionado, Mario puede correr, lo que aumenta su velocidad y la distancia de sus saltos. Correr también es crucial para superar ciertos obstáculos y enemigos más rápidamente.
- **Agacharse:** Mario puede agacharse para evitar ataques o para entrar en tuberías.
- **Disparo de Bolas de Fuego:** Al obtener la Flor de Fuego, Mario puede lanzar bolas de fuego, lo que permite atacar enemigos desde una distancia.

- **Romper Bloques desde Arriba:** Cuando Mario es Super Mario o posee la capacidad de fuego, puede romper ciertos bloques golpeándolos desde abajo.

Interracciones con el Entorno

- **Bloques Interactivos:** Golpearlos puede revelar monedas o power-ups, siendo parte fundamental de la exploración y progresión. Algunos bloques son rompibles, mientras que otros pueden ser usados múltiples veces.
- **Tuberías:** Sirven como acceso a áreas secretas y como obstáculos estratégicos. Algunas tuberías permiten a Mario desplazarse entre diferentes partes del nivel o acceder a áreas completamente nuevas.
- **Bandera de Final de Nivel:** Situada al final de los tres primeros niveles, al tocarla se termina el nivel. La altura a la que Mario toca la bandera determina la cantidad de puntos que recibe.
- **Castillo:** Al final de los tres primeros niveles, el castillo actúa como la puerta de entrada para el siguiente nivel. Cuando Mario entra en el castillo, se inicia una secuencia que lleva a Mario al próximo nivel, marcando la progresión del juego y ofreciendo una pausa visual entre los niveles con una breve animación de Mario entrando al castillo.
- **Plataformas Móviles:** Estas plataformas se mueven según patrones predefinidos y requieren precisión para ser alcanzadas, añadiendo una capa de desafío al movimiento y posicionamiento.
- **Bloques Invisibles:** Revelan ítems cuando Mario salta y los golpea desde abajo.
- **Barras de Fuego:** Giran alrededor de un punto fijo y representan un obstáculo que requiere timing para ser superado sin recibir daño.
- **Lava:** Presente en el nivel del castillo (nivel 4), que resulta en la pérdida de vida si Mario es pequeño o como un golpe si Mario es Super o Modo Fuego si cae en ella.

- **Bowser:** El antagonista principal al final del nivel del castillo. La interacción con Bowser implica esquivar sus ataques de llamas y derrotarlo o bien lanzando bolas de fuego o derrumbando el puente.
- **Hacha:** Ubicada al final del puente del nivel 4, tiene un papel crucial en la confrontación con Bowser. Al saltar sobre el hacha, Mario corta la cuerda del puente donde Bowser está parado, haciendo que caiga a la lava.
- **Toad:** Al terminar el nivel 4, Toad se encarga de dar por finalizado el juego.

Mécanicas de Enemigos

- **Patrones de Movimiento:** Cada tipo de enemigo tiene patrones de movimiento distintos. Por ejemplo, los Goombas caminan hacia adelante hasta que chocan con un obstáculo y se voltean, mientras que los Koopas se esconden en sus caparazones cuando son golpeados.
- **Interracciones:** Los enemigos pueden ser derrotados saltando sobre ellos o usando power-ups como las bolas de fuego.

4.3 Personajes

Cada personaje, tanto protagonistas como antagonistas, ha sido cuidadosamente diseñado para mantener la esencia del original mientras le he añadido detalle visual. Aquí se detalla la información sobre los personajes principales y los enemigos que los jugadores encontrarán.

Personaje Principal:

- **Mario:** Mario es el protagonista del juego. Su misión es rescatar a la Princesa Peach, superando una serie de obstáculos y enemigos a lo largo de varios niveles. Mario puede transformarse y obtener nuevas habilidades mediante el uso de power-ups.



Ilustración 4.1. Mario

Personajes Secundarios:

- **Toad:** Toad aparece en el último nivel después de que Mario derrota a Bowser. Actúa como un personaje con quien Mario interactúa para finalizar el juego, ofreciendo el diálogo de clausura.

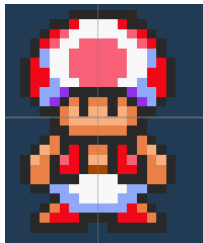


Ilustración 4.2. Toad

Antagonistas

- **Bowser:** El principal antagonista del juego, Bowser es el rey de los Koopas y ha secuestrado a la Princesa Peach. Es el jefe final que Mario debe enfrentar en el último nivel de cada mundo.



Ilustración 4.1. Bowser

- **Goombas:** Pequeñas criaturas con forma de seta que son los enemigos más comunes en los niveles. Se mueven lentamente hacia Mario y pueden ser derrotados fácilmente saltando sobre ellos.

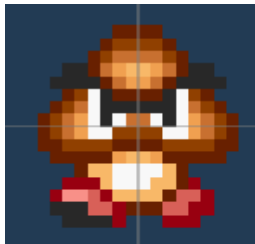


Ilustración 4.1. Goomba

- **Koopa Troopas:** Tortugas que caminan de un lado a otro. Al ser golpeados, se retraen dentro de sus caparazones, que luego pueden ser utilizados por Mario como proyectiles contra otros enemigos.

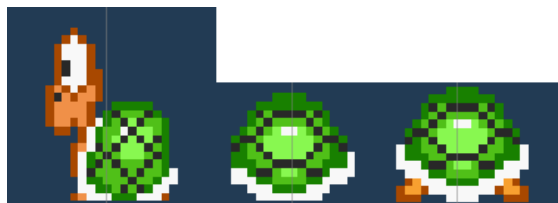


Ilustración 4.5. Koopa Troopa

- **Koopa Troopas Rojo:** El Koopa Rojo es una variante del Koopa tradicional que se distingue por su habilidad para no caer de las plataformas automáticamente, y tener una lógica especial de patrullaje.

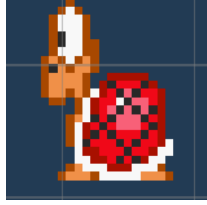


Ilustración 4.6. Koopa Rojo

- **Koopa Troopas Con Alas:** El Koopa con alas añade una capa de complejidad con su capacidad de volar y perder las alas.

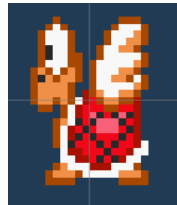


Ilustración 4.2. Koopa Con Alas

- **Planta Piraña:** Plantas carnívoras que salen de las tuberías intentando morder a Mario. No pueden ser derrotadas fácilmente y requieren que Mario las esquive hábilmente.



Ilustración 4.8. Planta Piraña

Tabla de Puntos Obtenidos por derrotar a cada enemigo:

Enemigo	Puntos Obtenidos
Goomba	100 puntos
Koopa Troopa	200 puntos
Koopa Rojo	200 puntos
Koopa con Alas	400 puntos
Planta Piraña	300 puntos
Bowser	5000 puntos

Tabla 4.2 Puntos obtenidos por derrotar a cada enemigo

4.4 Power-Ups y Objetos

El juego incluye varios power-ups y objetos que ayudan a Mario en su aventura, cada uno con características únicas que enriquecen la jugabilidad:

- **Super Champiñón:** Este power-up aumenta el tamaño de Mario, permitiéndole resistir un golpe adicional de cualquier enemigo. Además, al estar en este estado potenciado, Mario puede romper bloques de ladrillo al saltar bajo ellos.
- **Flor de Fuego:** Otorga a Mario la capacidad de lanzar bolas de fuego. Estas bolas pueden derrotar a la mayoría de los enemigos desde una distancia, añadiendo una estrategia ofensiva distinta al gameplay.
- **Estrella de Invencibilidad:** Proporciona a Mario un corto periodo de invulnerabilidad, durante el cual puede derrotar a cualquier enemigo con solo tocarlo.
- **Seta de Vida (One-Up):** Aumenta en una las vidas disponibles de Mario. Este ítem es raro y a menudo se encuentra oculto o en áreas de difícil acceso, premiando la exploración y la habilidad en el juego.
- **Monedas:** Sirven como la moneda del juego y son abundantes en todos los niveles. Al recolectar 100 monedas, Mario recibe una vida extra, lo que incentiva la recolección meticulosa a lo largo de cada nivel.

Tabla de Puntos que Mario recibe al interactuar con cada power-up y objeto:

Enemigo	Puntos Obtenidos
Super Champiñón	1000 puntos
Flor de Fuego	1000 puntos
Estrella de Invencibilidad	1000 puntos
Seta de Vida (One-Up)	1000 puntos
Monedas	200 puntos

Tabla 4.3 Puntos obtenidos por recoger un power-up u objeto

4.5 Interfaz de Usuario

El diseño de la UI en este remake de Super Mario Bros está centrado en la claridad y funcionalidad, asegurando que los jugadores tengan acceso fácil a la información esencial mientras juegan.

1. HUD

Indicador de Puntos: Muestra la puntuación actual del jugador, actualizando en tiempo real conforme Mario recolecta monedas, derrota enemigos o alcanza hitos importantes dentro del nivel, como capturar la bandera.

Contador de Monedas: Informa al jugador sobre el número de monedas recolectadas, lo que es crucial para el seguimiento de progreso hacia obtener vidas extras.

Indicador del Nivel: Muestra el nivel actual en el que se encuentra Mario, proporcionando a los jugadores una referencia clara de su progreso en el juego

Temporizador de Nivel: Un reloj regresivo que añade urgencia y desafío al juego, empujando a los jugadores a completar cada nivel antes de que se agote el tiempo.



Ilustración 4.3. HUD

2. Menús y Pantallas

Pantalla - Inicio: La primera interacción del jugador con el juego. Muestra el título del juego y ofrece opciones para comenzar un juego nuevo o cargar la partida en el nivel donde lo dejó.

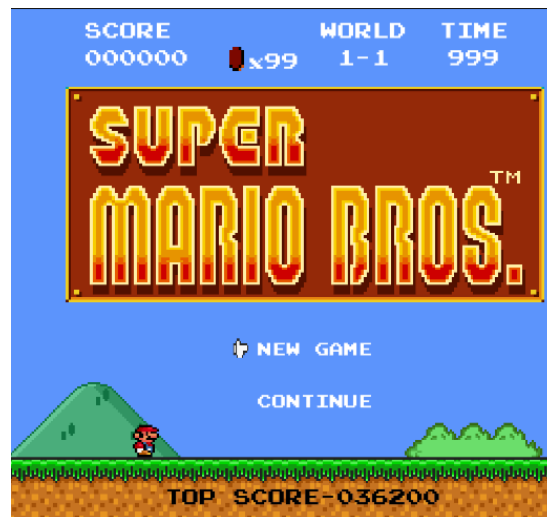


Ilustración 4.4. Pantalla Inicio

Pantalla - Nivel Superado: Se muestra al completar un nivel, resumiendo el desempeño del jugador con detalles como el número de monedas recogidas, el nivel siguiente del juego y las vidas restantes.



Ilustración 4.5. Pantalla – Nivel Superado

Pantalla - Perder Vida: Aparece cuando Mario pierde una vida pero aún le quedan vidas adicionales.



Ilustración 4.6. Pantalla – Perder Vida

Pantalla - Game Over: Se muestra cuando Mario pierde todas sus vidas, devolviéndolo a la pantalla de Inicio.



Ilustración 4.7. Pantalla – Game Over

3. Feedback Visual y Auditivo

Alertas Visuales: Información como la obtención de un nuevo power-ups o la pérdida de una vida se destacan mediante animaciones y cambios visuales en la HUD. Esto incluye la visualización de la puntuación al obtener un ítem, eliminar a un enemigo, y captura la bandera, así como efectos visuales al obtener un power-up o recibir un golpe.

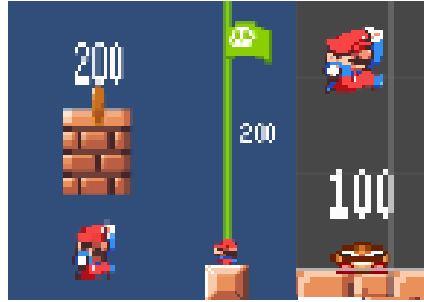


Ilustración 4.8. Alertas Visuales

Efectos Sonoros: Cada acción importante, como saltar, recoger monedas, o activar un power-up, está acompañada de efectos de sonido distintivos que mejoran la retroalimentación inmediata al jugador.

5 DESARROLLO DEL JUEGO

5.1 Configuración Inicial

5.1.1 Configuración del Proyecto en Unity

Para el desarrollo del remake, he seleccionado la versión 2021.3.38f1, conocida por su estabilidad y soporte extenso.

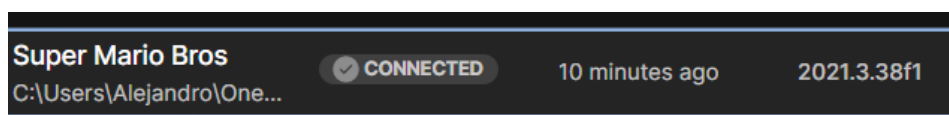


Ilustración 5.1. Versión Unity

Selección del Template:

He optado por el template 2D de Unity, que proporciona una configuración óptima del entorno para desarrollar juegos en dos dimensiones. Este template preconfigura el editor con las herramientas y paneles adecuados para el diseño en 2D, asegurando que los recursos como sprites y animaciones sean fácilmente gestionables y optimizados para este tipo de proyectos.

Estructura del proyecto:

El proyecto está organizado en una estructura de carpetas estándar que Unity recomienda y utiliza por defecto, lo que facilita la gestión y el acceso a los diversos recursos y configuraciones del proyecto.

- **Assets:** en esta carpeta se guardan todos los recursos del juego, scripts, sprites, fuentes de texto, animaciones, prefabs, archivos de audio y demás.
- **Packages:** esta carpeta contiene la lista de paquetes agregados al proyecto.
- **ProjectSettings:** esta carpeta contiene los archivos con las configuraciones de distintas partes del proyecto como la física, capas, etiquetas y configuraciones de tiempo.

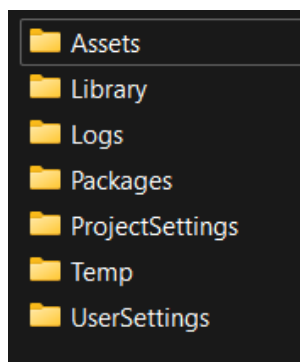


Ilustración 5.2. Estructura del Proyecto

5.2 Desarrollo de Características

5.2.1 Creación y Manipulación de Sprites

Para cargar y manipular un Sprite en Unity hay que seguir una serie de pasos específicos que permiten utilizar estos componentes gráficos de manera efectiva en el proyecto. Antes de nada, voy a definir los conceptos básicos [6]:

Sprite: Es una imagen en dos dimensiones que se utiliza para representar objetos gráficos como personajes, enemigos, y elementos del entorno en un videojuego. Los Sprites son importados en formatos PNG o JPG, y se transforman en componentes manejables dentro del motor a través de la configuración de su Sprite Renderer.

Sprite Renderer: Componente de Unity que se añade a un GameObject para poder mostrar el Sprite en la escena. Controla cómo se visualiza el Sprite, incluyendo la transparencia, el color y la capa.

Pasos para Cargar y Manipular Sprites

1. Importar la Imagen

Paso 1: Arrastrar y soltar la imagen en la carpeta “Sprites” dentro de “Assets” del proyecto en Unity.

Paso 2: En el panel “Inspector”, cambiar el “Texture Type” a “Sprite (2D and UI)”.

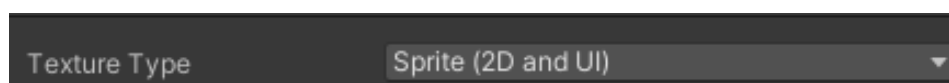


Ilustración 5.3. Texture Type

2. Configurar el Sprite

Paso 3: Ajustar las propiedades del Sprite en el “Inspector”

Las opciones incluyen:

- **Sprite Mode:** Define si la imagen se maneja como un “Simple” Sprite Individual o “Multiple”, que es útil para spritesheets donde múltiples sprites están contenidos en una sola imagen. En el modo “Multiple”, se necesitará usar el Sprite Editor para definir los límites de cada Sprite individual dentro de la hoja.

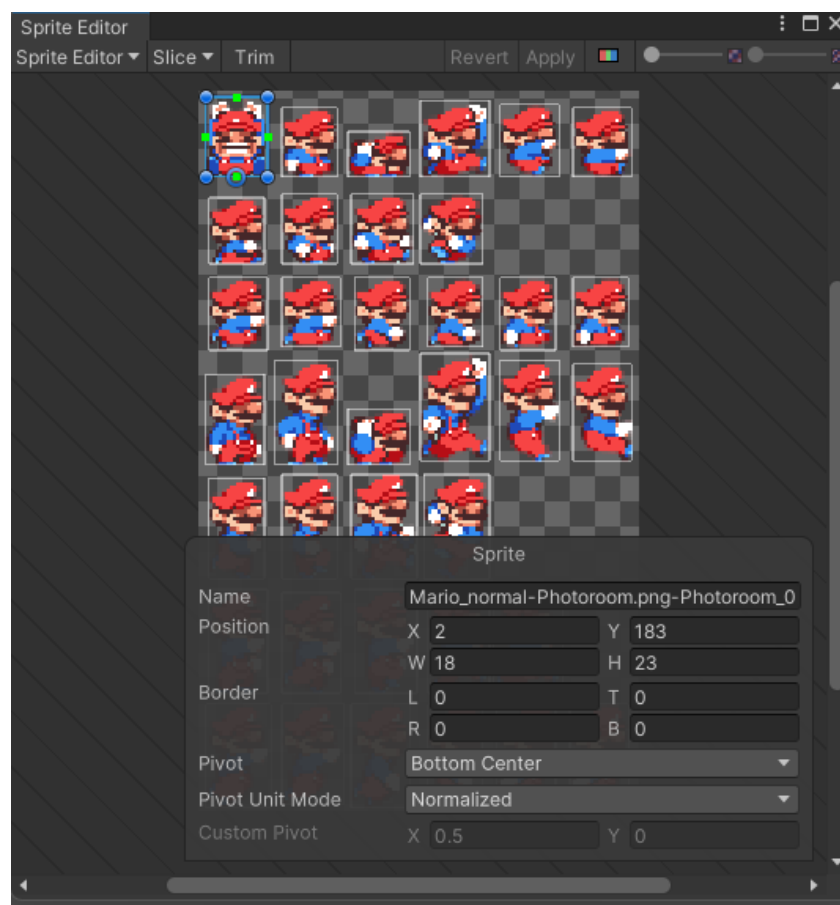


Ilustración 5.4. Sprite Editor

- **Pixel per Unit:** Esta propiedad establece cuántos píxeles de la imagen corresponden a una unidad en Unity.
- **Pivot:** El pivot es el punto alrededor del cual se realizarán todas las rotaciones y escalas del Sprite.

- **Filter Mode:** Esta propiedad define cómo Unity interpola los píxeles del Sprite cuando es escalado o transformado:
 - *Point:* No hay interpolación; los píxeles permanecen bloqueados y nítidos, ideal para un estilo visual pixelado.
 - *Bilinear:* Los píxeles se suavizan, lo que hace que los Sprites se vean menos pixelados y más suaves al ser escalados.

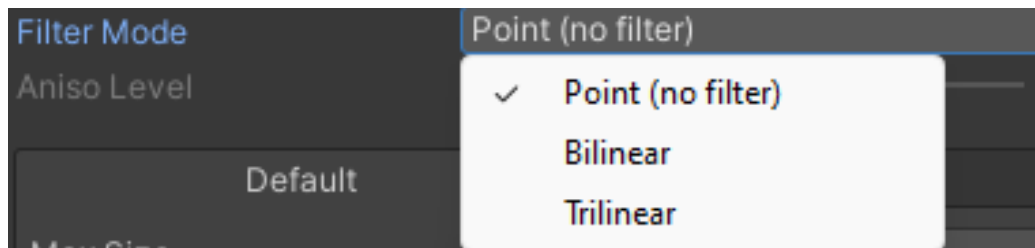


Ilustración 5.5. Filter Mode

3. Añadir Sprite a la Escena

Paso 4: Se crea un nuevo GameObject en la escena y se añade el componente “Sprite Renderer”

En “Sprite Renderer” existe la opción de “**Sorting Layers**”, que permite controlar el orden de los objetos en la escena, es decir esta propiedad sirve para que algunos objetos se dibujen sobre otros.

Paso 5: Se asigna el Sprite al componente “Sprite Renderer” arrastrando el Sprite desde el panel “Project” al campo “Sprite” en el Inspector del GameObject.

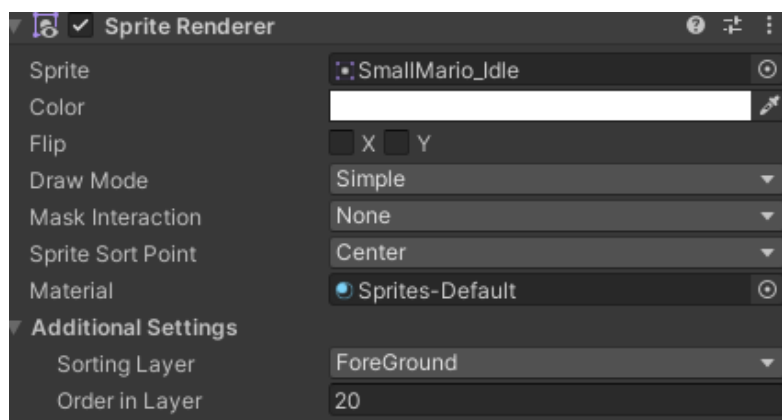


Ilustración 5.6. Sprite Renderer

5.2.2 Implementación de Físicas y Colisiones

Para implementar las Físicas y las Colisiones a los objetos he seguido los siguientes pasos, lo que garantiza que los objetos interactúen de manera realista con su entorno y otros objetos del juego [7].

1. Añadir un Rigidbody al GameObject:

Selección del GameObject: primero se escoge el objeto dentro de la jerarquía del proyecto que se desea aplicar las físicas.

Agregar el componente Rigidbody: esto permite que el objeto pueda responder a la gravedad y a otras fuerzas, como impulsos y colisiones.

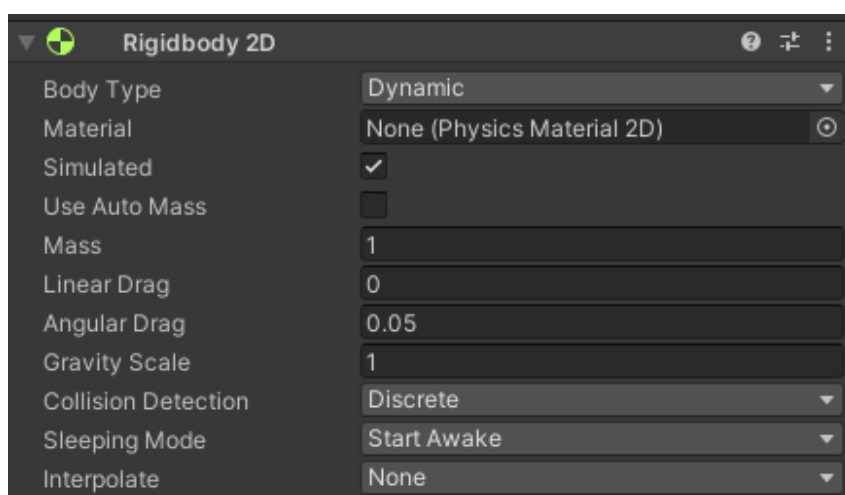


Ilustración 5.7. Rigidbody 2D

2. Configuración del Rigidbody:

Tipo de Rigidbody: Se elige entre “**Dynamic**”, para objetos que reaccionarán a fuerzas y colisiones; “**Kinematic**”, que no reaccionan a la gravedad, pero pueden ser movidos mediante scripts; o “**Static**”, para objetos que no se mueven, pero pueden ser colisionados.

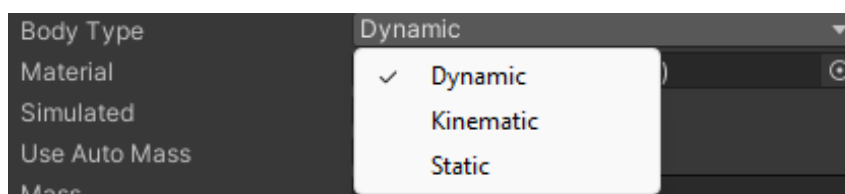


Ilustración 5.8. Tipo de Rigidbody 2D

3. Añadir Colliders:

Selección del tipo de Collider: Se tiene que escoger el Collider que se ajuste a la forma del objeto. Esto puede ser un **‘BoxCollider’**, **‘SphereCollider’**, o **‘MeshCollider’**.

Ajuste del Collider: Además se debe modificar las dimensiones y la posición del Collider para asegurar de que cubre el objeto de manera precisa, proporcionando una representación exacta del espacio que ocupa el objeto para la detección de colisiones.

4. Configuración de las Colisiones:

Uso de Capas (Layers): Hay que organizar los objetos en capas para controlar con qué otros objetos pueden interactuar.

	Default	TransparentFX	Ignore Raycast	Water	UI	Player	Ground
Default							
TransparentFX							
Ignore Raycast							
Water							
UI							
Player						✓	
Ground							

Ilustración 5.2. Layer Collision Matrix

Aquí se puede ver como se configura las capas, y en este caso, se muestra que solo va a existir colisión entre la capa “Player” y la capa “Ground”.

5. Implementación de Materiales Físicos:

Además, se puede crear y ajustar propiedades como la **fricción** y el **rebote** para influir en cómo los objetos rebotan o resbalan al colisionar. Esto se aplica al **‘Collider’** para cambiar la manera en que el objeto interactúa al entrar en contacto con otros objetos.

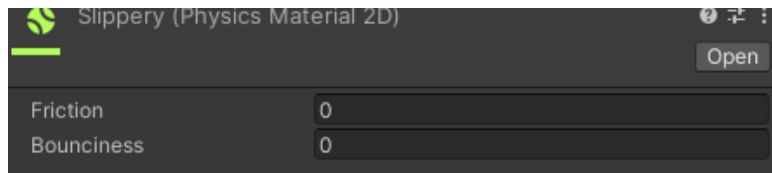


Ilustración 5.9. Physics Material 2D

6. Script de Movimiento (Mover.cs)

Control de Movimiento: He implementado un sistema que permite a Mario moverse hacia izquierda o derecha y ajustar su velocidad basándose en la aceleración y la fricción, simulando un movimiento realista dentro del entorno del juego.

```
private void FixedUpdate()
{
    //Se aplica la física de movimiento horizontal basada en la dirección actual
    currentVelocity = rb2D.velocity.x;
    if(currentDirection > 0)//Movimiento hacia la derecha
    {
        if(currentVelocity < 0)// Si está moviéndose hacia la izquierda, se añade fricción para detenerlo.
        {
            currentVelocity += (acceleration + friction) * Time.deltaTime;
        }
        else if (currentVelocity < maxVelocity)// Se acelera hacia la derecha.
        {
            currentVelocity += acceleration * Time.deltaTime;
        }
    }
    else if(currentDirection < 0)// Movimiento hacia la izquierda.
    {
        if(currentVelocity > 0)// Si está moviéndose hacia la derecha, añade fricción para detenerlo.
        {
            currentVelocity -= (acceleration + friction) * Time.deltaTime;
        }
        else if (currentVelocity > -maxVelocity)// Acelera hacia la izquierda
        {
            currentVelocity -= acceleration * Time.deltaTime;
        }
    }
}
```

Ilustración 5.10. Control de Movimiento

Mecánica de Salto: Mario puede iniciar un salto solo si está en contacto con el suelo, verificado por el script de colisiones. Durante el salto, la gravedad de Mario se ajusta para ofrecer una respuesta dinámica basada en la duración que el jugador mantiene presionado el botón de salto.

```

void Update()
{
    //LÓGICA DEL SALTO
    if (isJumping)
    {
        if (rb2D.velocity.y < 0f)//Cuando el personaje está cayendo
        {
            //Se restablece la gravedad al valor predeterminado al caer
            rb2D.gravityScale = defaultGravity;
            if (colisiones.Grounded())//Se mira si el personaje ha tocado el suelo
            {
                //Finaliza el salto cuando el personaje toca el suelo
                isJumping = false;
                jumpTimer = 0;
            }
        }
        else if (rb2D.velocity.y > 0f)//Mientras el personaje esta subiendo
        {
            //se permite mantener el botón de salto para un salto más alto
            if (Input.GetKey(KeyCode.Space))
            {
                jumpTimer += Time.deltaTime;
            }
            if (Input.GetKeyUp(KeyCode.Space))
            {
                //al soltar el espacio, aumenta la gravedad para caer más rápido si el tiempo de salto no se ha completado
                if (jumpTimer < maxJumpingTime)
                {
                    rb2D.gravityScale = defaultGravity * 3f;
                }
            }
        }
    }
}

```

Ilustración 5.11. Mecánica de Salto

7. Colisiones (Colisiones.cs)

Para la detección del suelo, utilizo un script de colisiones que verifica constantemente si Mario está en contacto con el suelo mediante técnicas de raycasting [8].

```

public bool Grounded()
{
    // Determina la posición de los pies del personaje para el Raycast.
    Vector2 footLeft = new Vector2(collider2D.bounds.center.x - collider2D.bounds.extents.x, collider2D.bounds.center.y);
    Vector2 footRight = new Vector2(collider2D.bounds.center.x + collider2D.bounds.extents.x, collider2D.bounds.center.y);

    // Realiza un Raycast hacia abajo desde el borde izquierdo y derecho del collider para detectar el suelo.
    if (Physics2D.Raycast(footLeft, Vector2.down, collider2D.bounds.extents.y * 1.5f, groundLayer))
    {
        isGrounded = true;
    }
    else if (Physics2D.Raycast(footRight, Vector2.down, collider2D.bounds.extents.y * 1.5f, groundLayer))
    {
        isGrounded = true;
    }
    else
    {
        isGrounded = false;
    }
    return isGrounded; // Retorna si el personaje está o no en el suelo.
}

```

Ilustración 5.12. Detección del suelo

Uso del 'Time.deltaTime': es un valor crucial que representa el tiempo en segundo que ha transcurrido desde que se dibujó el último frame. Este valor es esencial para garantizar que la mecánica de juego se ejecuta de manera suave y consistente en diferentes dispositivos. Es tan importante porque sin el uso de este, un juego correría más rápido en sistemas más poderosos y más lento en sistemas con menos recursos. Al multiplicar las velocidades y fuerzas aplicadas por 'Time.deltaTime', se normaliza el movimiento a una tasa consistente, independiente del hardware.

5.2.3 Animación de Personajes y Objetos

En este apartado, describo el proceso de animación, el cual permite a los objetos y personajes del juego mostrar un comportamiento dinámico y atractivo visualmente. Los pasos para implementar las animaciones son los siguientes:

- 1. Concepto de Animaciones:** Las animaciones son secuencias de imágenes estáticas (sprites) que, al ser reproducidas a una velocidad adecuada, crean la ilusión de movimiento [9].
- 2. Uso de Unity Animator:** Unity emplea un sistema llamado "Animator Controller", que funciona como una máquina de estados finitos. Cada estado representa una animación diferente, y el sistema puede cambiar de un estado a otro bajo ciertas condiciones, permitiendo una transición entre animaciones [10].
- 3. Creación de Clips de Animaciones:** Se inicia seleccionando los sprites que formarán parte del clip de animación y arrastrándolos directamente a la escena de Unity. Esto automáticamente crea un GameObject con un componente Sprite Renderer y, si se trata de múltiples sprites, añade un Animator y un clip de animación [11].

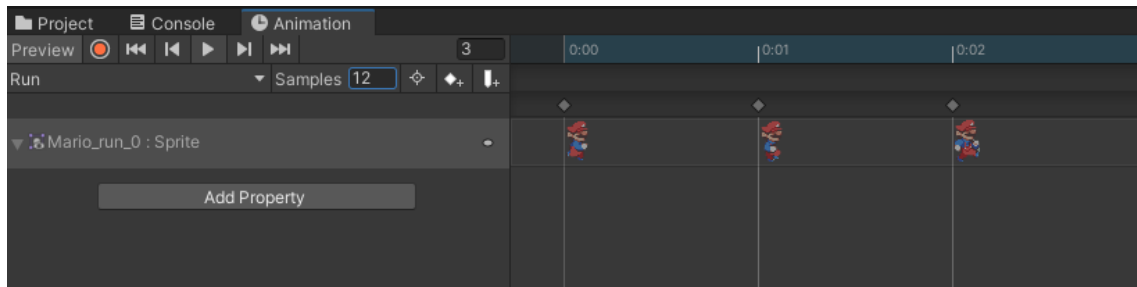


Ilustración 5.13. Animation

- 4. Configuración del Animator Controller:** Dentro del “Animator Controller”, se configuran los diferentes estados de animación y las transiciones entre ellos.

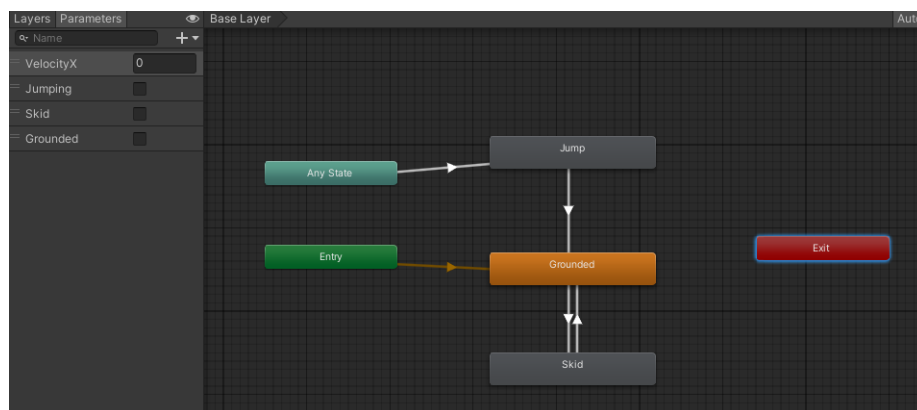


Ilustración 5.14. Animator

- 5. Animaciones en Bucle y No Bucle:** En esencial determinar si una animación debe reproducirse en bucle (como correr) o si debe ejecutarse una sola vez (como un salto).

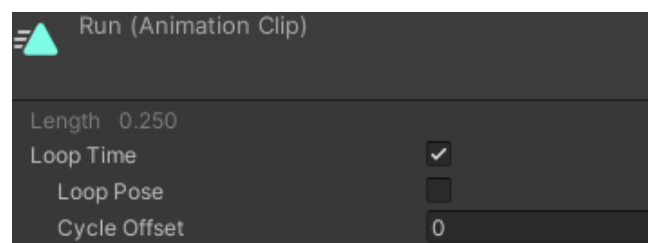


Ilustración 5.15. Configuración Animación

Para implementar correctamente las animaciones y garantizar que se sincronizan con las acciones y movimientos del personaje en el juego, se realizan ajustes que aseguran que las animaciones se activen en los momentos precisos, de acuerdo con la lógica del Juego:

- **Integración con el Sistema de Animación:** Se añade una referencia al componente “Animator”. Esto permite activar o cambiar animaciones basadas en el estado y acciones del personaje, como saltar, correr o detenerse.

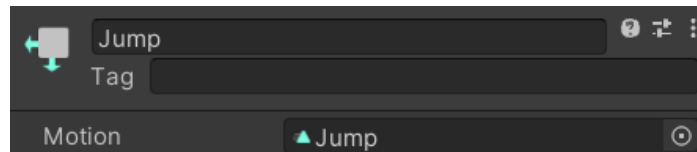


Ilustración 5.16. Integración con el Sistema de Animación

- **Control Dinámico de Animaciones:** Se utiliza los métodos “animator.SetBool” y animator.SetFloat” para ajustar las animaciones basadas en el estado de movimiento del personaje.

```
// Método para actualizar el estado de "Jumping" en el Animator.  
3 referencias  
public void Jumping(bool isJumping)  
{  
    animator.SetBool("Jumping", isJumping);  
}
```

Ilustración 5.17. Control Dinámico de Animaciones

- **Sincronización con la Física:** Las animaciones se sincronizan con los estados físicos del personaje, verificando condiciones como si no se está en el suelo y saltando a través del script de colisiones.



Ilustración 5.18. Transición de Estados - Animator

Además, se ha añadido la capacidad de agacharse a Mario, limitada a sus formas Super Mario y Mario Fuego. La implementación involucra:

- **Animaciones específicas:** Se han creado animaciones dedicadas para Mario agachado tanto en su forma Super como en la forma Fuego.

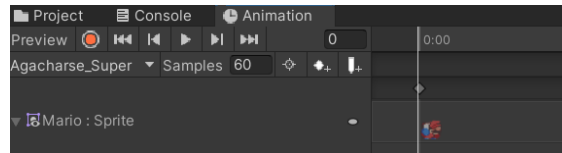


Ilustración 5.19. Animación Agachado

- **Restricciones de movimiento:** Mientras está agachado, Mario no puede moverse.

```
if (Input.GetKey(KeyCode.DownArrow))
{
    if (colisiones.Grounded()) //si esta colisionando con el suelo puede agacharse
    {
        estaAgachado = true;
    }
}
```

Ilustración 5.20. Animación Agachado

- **Bloqueo de acciones:** En forma de Fuego, no puede lanzar bolas de fuego mientras está agachado.
- **NUEVA FUNCIONALIDAD – Romper bloques con un Pisitón** Al estar agachado, Mario puede romper bloques presionando la tecla 'z'.

5.2.4 Creación y Gestión de Enemigos

En esta sección, describo la implementación de enemigos en el juego, enfocándome en cómo interactúan con el jugador y el entorno. La creación de enemigos sigue un proceso detallado que asegura su comportamiento dinámico y realista dentro del juego.

1. Preparación de Sprites:

Organización: Los sprites de los enemigos lo he organizado en carpetas específicas dentro del proyecto para fácil acceso y modificación.

Configuración: Ajusto las propiedades de tamaño y el punto de pivote de los sprites para asegurar animaciones precisas.

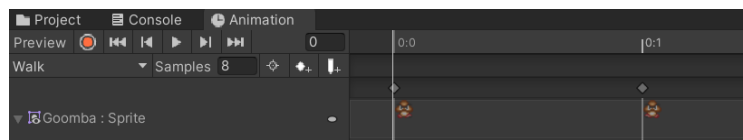


Ilustración 5.21. Animation Enemigo (Goomba)

2. Animación: Utilizo el “Animator Controller” para crear las animaciones específicas de cada enemigo.

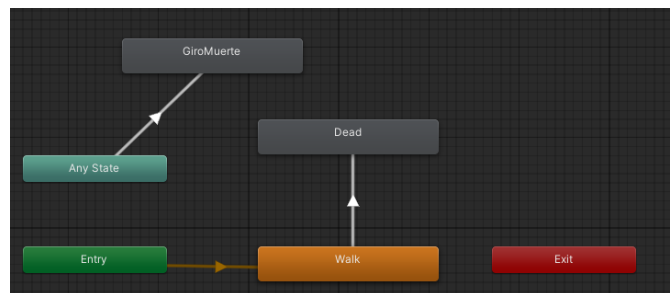


Ilustración 5.22. Animator Enemigo (Goomba)

3. Componentes Físicos: Añado un “Rigidbody2D” para permitir que el enemigo interactúe con la física del juego y un “Collider” para definir el área de interacción física.

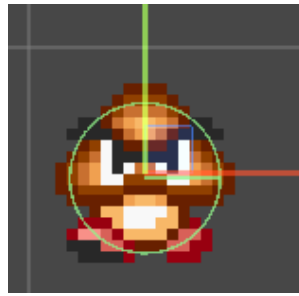


Ilustración 5.23. Componentes Físicos del Enemigo (Goomba)

- 4. Script de Movimiento:** Utilizo una estructura de clases con una clase base 'Enemigo' que define propiedades y métodos comunes. De esta clase derivan otras como 'Goomba', 'Koopa' y 'Planta Piraña', que personalizan el movimiento y otros comportamientos.

Clase Base: Enemigo: Incluye componentes básicos como 'Animator', 'Automovimiento', y 'Rigidbody2D'. Este método facilita la reutilización y expansión de funcionalidades.

```
public class Enemigo : MonoBehaviour
{
    protected Animator animator; // Componente para controlar las animaciones del enemigo
    protected AutoMovimiento autoMovimiento; // Script que maneja el movimiento automático del enemigo
    protected Rigidbody2D rb2D; // Componente Rigidbody2D para manejar la física del enemigo
}
```

Ilustración 5.24. Clase 'Enemigo'

Movimiento Autónomo: 'AutoMovimiento' gestiona el movimiento automático y la dirección de los enemigos, adaptándose a obstáculos mediante cambio de dirección y detención.

Clases Derivadas: Cada clase derivada ajusta y expande el comportamiento de la clase base.

- 5. Gestión de Colisiones:** Configuro las colisiones para que el enemigo pueda interactuar de manera adecuada con elementos del juego.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    // Detecta colisiones con objetos definidos como enemigos y llama al método Hit de Mario
    if (collision.gameObject.layer == LayerMask.NameToLayer("Enemigo"))
    {
        if (mario.esInvencible)
        {
            // Si Mario es invencible, ejecuta la muerte por invencibilidad del enemigo
            Enemigo enemigo = collision.gameObject.GetComponent<Enemigo>();
            enemigo.golpeDeModoInvencible();
        }
        else
        {
            mario.Hit(); // Si Mario no es invencible, ejecuta el método de recibir daño
        }
    }
}
```

Ilustración 5.25. Gestión de Colisiones Enemigo

- 6. Interacciones con el Jugador:** Implemento lógicas específicas para que cada tipo de enemigo responda de manera única cuando interactúa con el jugador.

```
public class Goomba : Enemigo // Se hereda de la clase Enemigo
{
    // Se sobrescribe el método Pisoton definido en la clase base Enemigo
    2 referencias
    public override void Pisoton(Transform player)
    {
        animator.SetTrigger("Hit"); // Se activa la animación de ser golpeado
        //Se cambia la capa del objeto a "OnlyGround" para evitar más colisiones con enemigos
        gameObject.layer = LayerMask.NameToLayer("OnlyGround");
        //Se destruye el objeto después de un breve tiempo, dando tiempo para que se muestre la animación
        Destroy(gameObject, 0.5f);
        autoMovimiento.PausarMovimiento(); //Se pausa el movimiento del objeto, deteniendo su auto-movimiento
    }
}
```

Ilustración 5.26. Clase Derivada de Enemigo (Goomba)

Para cada tipo de enemigo, se ha desarrollado animaciones específicas que reflejan sus comportamientos únicos:

1. Goomba

Andar: Animación básica de movimiento.

Muerte: Al ser pisoteado por el jugador, se aplasta y desaparece.

2. Koopa

Andar: Movimiento habitual caminando

Escondarse en el caparazón: Cuando el jugador salta encima, se esconde en su caparazón.

Movimiento en el caparazón: Puede ser empujado para atacar a otros enemigos o dañar a Mario.

Salida del caparazón: Después de un tiempo o cuando el jugador salta de nuevo sobre él mientras está en su caparazón.

3. Planta Piraña

Mostar y esconderse: La planta sale de la tubería y se retira alternativamente.

Muerte por bola de fuego: Cuando es atacada por una bola de fuego, desaparece sin animación específica.

4. Koopa Rojo

El Koopa Rojo se distingue de otros Koopas por su habilidad para no caer de las plataformas automáticamente, lo que implica una lógica especial de patrullaje:

- **Detección de Bordes:** Se implementó un sistema de detección de bordes para el Koopa Rojo usando raycast. Este sistema permite a Koopa cambiar de dirección al alcanzar el final de una plataforma, evitando que se caiga de ella.

```

// Control de movimiento estándar con manejo de caídas y colisiones laterales
if (estaCayendo)
{
    if (CheckGrounded())
    {
        estaCayendo = false;
    }
}
else
{
    if (CheckCollisionLateral())
    {
        cambiarDireccion();
    }
    else if (evitarCaída && !CheckGrounded())
    {
        cambiarDireccion();
    }
    else
    {
        CheckTiempoParado();
    }
    rb2D.velocity = new Vector2(speed, rb2D.velocity.y);
}

bool CheckGrounded()//Comprueba si el enemigo está en contacto con el suelo
{
    // Comprueba si el objeto está en el proceso de caída
    if (estaCayendo)
    {
        // Centro: Ubicación central del collider del objeto
        Vector2 centro = new Vector2(col2D.bounds.center.x, col2D.bounds.center.y);
        // Realiza un raycast hacia abajo desde el centro del objeto
        if (Physics2D.Raycast(centro, Vector2.down, col2D.bounds.extents.y * 1.25f, groundLayer))
        {
            return true; // Si el rayo detecta el suelo, devuelve verdadero indicando que el objeto está en el suelo
        }
        else
        {
            return false; // Si no hay suelo debajo, devuelve falso
        }
    }
}

```

Ilustración 5.27. Utilización de las paletas del Tilemap

- **Movimiento Automatizado:** Utilizando el script 'AutoMovimiento', el Koopa Rojo sigue un patrón de movimiento horizontal hasta que detecta un borde o una colisión lateral, momento en el cual invierte su dirección.

5. Koopa Con Alas

El Koopa con alas añade una capa de complejidad con su capacidad de volar y perder las alas:

- **Capacidad de Vuelo:** Al inicio, el Koopa con Alas es capaz de volar, lo que se gestiona a través de animaciones específicas y cambios en la lógica de movimiento para simular el vuelo.

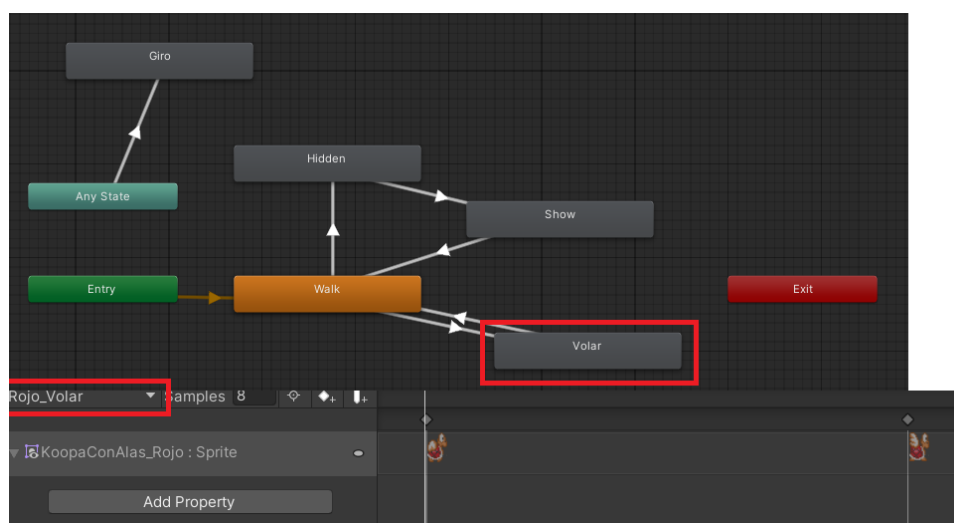


Ilustración 5.28. Utilización de las paletas del Tilemap

- **Pérdida de Alas:** Al ser pisoteado por primera vez, el Koopa con Alas pierde sus alas. Esto se manejó mediante una transición en la que el Koopa pasa de un estado de vuelo a un movimiento normal, modificando dinámicamente sus propiedades físicas. Tras perder las alas, el Koopa con Alas adopta comportamientos de un Koopa normal.

```
// Sobrescribe el método Pisoton para incorporar la lógica de perder alas
5 referencias
public override void Pisoton(Transform player)
{
    if (estaVolando)
    {
        AudioManager.Instancia.PlayPisoton(); // Reproduce el sonido de pisotón
        PerderAlas(); // Llama a la función que maneja la pérdida de alas
    }
    else
    {
        base.Pisoton(player); // Si ya no está volando, aplica la lógica de pisotón de la clase base
    }
}
```

Ilustración 5.29. Utilización de las paletas del Tilemap

5.2.5 Implementación de Transformaciones y Power-ups

Para la implementación de Transformaciones y Power-ups, he utilizado el componente 'Animator', que permite gestionar las animaciones mediante un sistema de estados y transiciones. Los estados de Mario los he configurado con animaciones específicas que se activan bajo ciertas circunstancias, como obtener un objeto o interactuar con enemigos.

1. Animaciones y Estados en Animator:

Se duplican las animaciones existentes de Mario pequeño para los estados de Super Mario y Mario Fuego, ajustando los sprites correspondientes.

2. Substates para Organizar Animaciones

Utilizo substates dentro de 'Animator' para agrupar animaciones similares (Small, Super, Fuego), facilitando la gestión y la visualización.

3. Gestión de Transiciones y Condiciones

Ajusto las transiciones entre los estados de animación en función de las acciones del jugador.

Se establece condiciones en el 'Animator' para controlar cuándo debe pasar de un estado a otro.

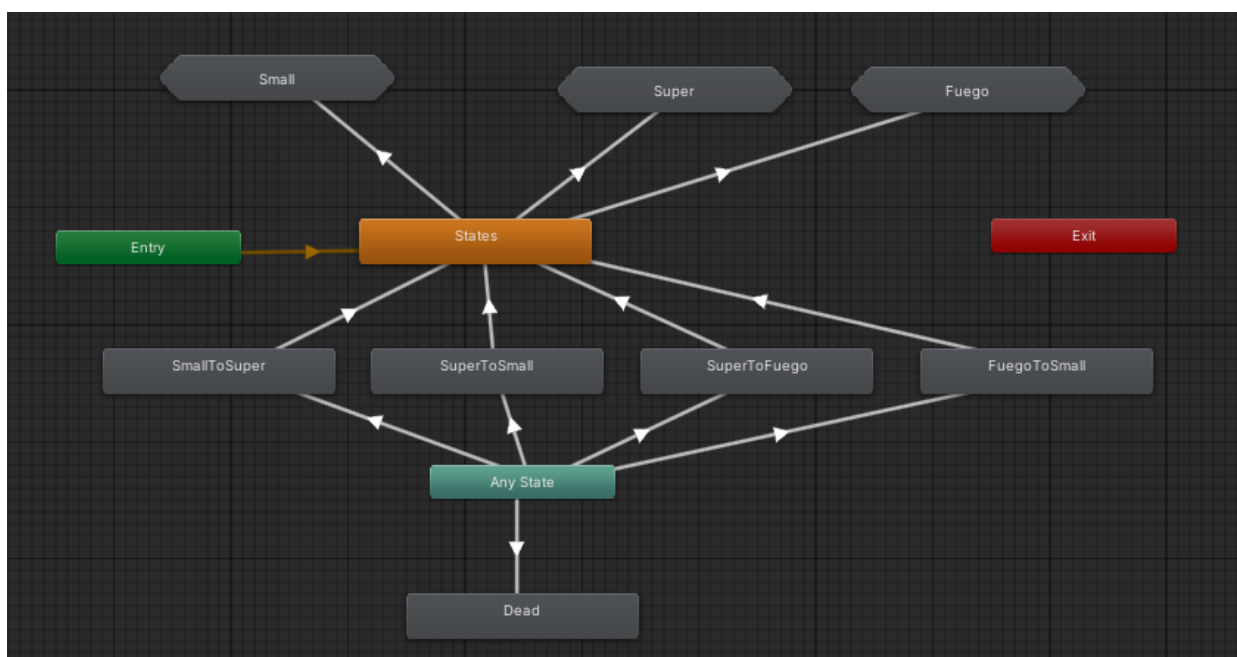


Ilustración 5.30. Animator Mario

4. Implementación de Power-Ups

Se añaden eventos a las animaciones para manejar los cambios de estado cuando Mario obtiene power-ups, es decir, añadiendo triggers y variables en el 'Animator' para activar las transformaciones adecuadas.

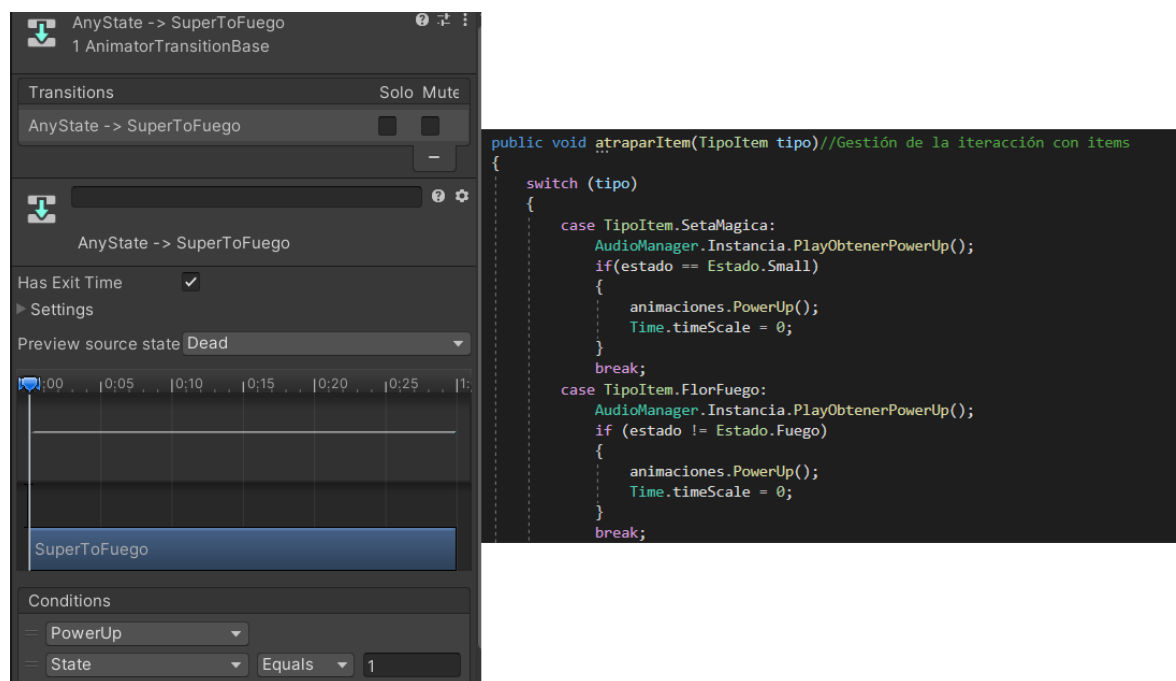


Ilustración 5.31. Gestión Power-Ups

Optimización y Efectos Adiciones

Además, he añadido efectos visuales como la transparencia durante las transformaciones para mejorar la experiencia visual, y además se maneja la escala de tiempo del juego para pausar otros elementos mientras Mario se transforma, asegurando que las animaciones de transformación se ejecuten sin interrupciones.

5.2.6 Gestión de Items, Poderes Especiales y Bloques

Para poder gestionar los Items como power-ups y monedas, he utilizado la funcionalidad de prefabs, que es un asset que encapsula un GameObject con sus componentes y propiedades, permitiendo su uso repetido, lo que permite una implementación eficiente y uniforme a lo largo de los diferentes niveles del juego. Al realizar cambios en un prefab, estos se aplican a todas las instancias de ese prefab en el juego [12].

Para enemigos, ítems y bloques que requieren variaciones, como diferencias en comportamiento y apariencia, utilizo la funcionalidad de 'Prefab Variant. Esto permite que las variantes hereden propiedades del prefab base, pero también mantengan sus características únicas.

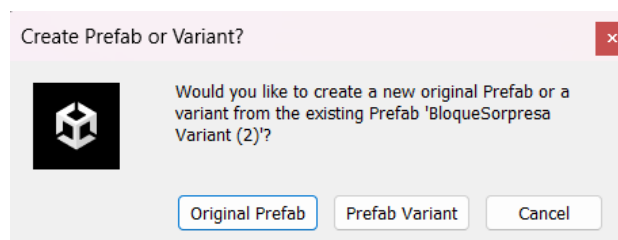


Ilustración 5.32. Prefab Variant

Al cambiar una propiedad en el prefab base, como la velocidad de un enemigo, todas las variantes que no hayan modificado esa propiedad específica se actualizarán automáticamente.

Para la animación de los ítems y bloques, he optado por utilizar scripts directos en lugar de los sistemas ‘Animator’ y ‘Animation’. Este enfoque simplifica el proceso al evitar la sobrecarga que implica manejar numerosos controladores de animación, especialmente útil para los ítems que solo requieren animaciones básicas de cambio de Sprite.

En la implementación, cada ítem tiene asignado el script que gestiona un array de sprites. Mediante una corutina, el script cicla continuamente a través de este array, actualizando el Sprite visible en intervalos definidos.

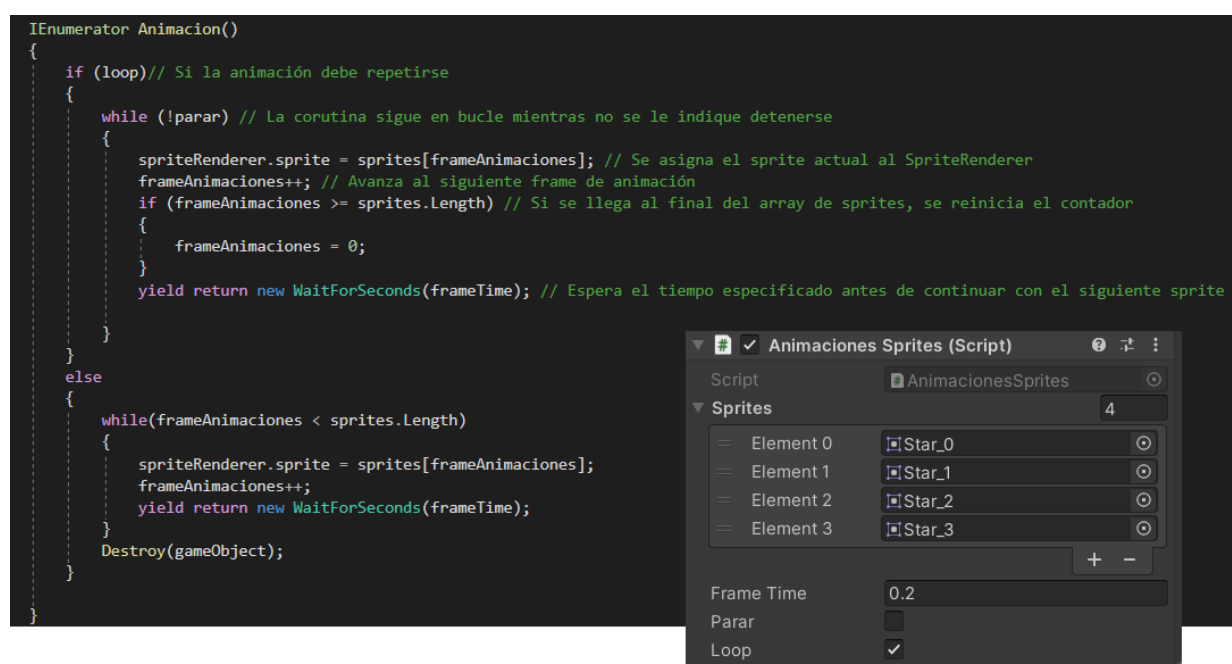


Ilustración 5.33. AnimacionesSprites.cs

Además, he implementado un sistema de gestión de bloques similar al de ítems, que permite interacciones dinámicas con Mario. Este sistema se basa en varios scripts que permiten a los bloques responder a diferentes acciones de Mario.

1. **Bloque:** Controla las interacciones con los bloques, detectando golpes de Mario desde abajo. Dependiendo de su configuración, puede romperse, emitir monedas o ítems, además de iniciar una animación de rebote y cambiar su Sprite a una versión vacía una vez agotados los recursos.

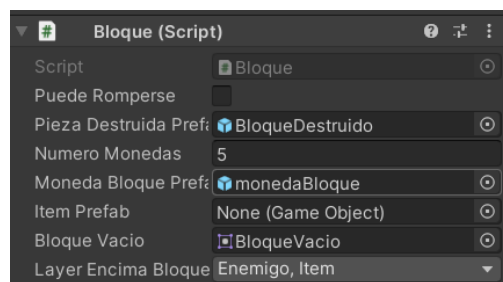


Ilustración 5.34. Parámetros Bloque

2. **MonedaBloque:** Gestiona las monedas que aparecen al golpear ciertos bloques, animando su trayectoria hacia arriba y su desaparición.
3. **AnimacionesSprite:** Utilizado en bloques que cambian visualmente, como pasaba en los ítems, este script anima secuencias de Sprite para visualizar cambios sin la necesidad de sistemas de animación más complejos.
4. **AutoMovimiento:** Estos se aplican a los ítems, que se mueven independientemente tras ser libreados de un bloque.

Para el **lanzamiento de Fuego**, he configurado un sistema que permite a Mario disparar Bolas de Fuego cuando se encuentra en el estado “Fuego”. Este proceso se inicia con la creación de un prefab de bola de fuego, que lo he configurado con propiedades específicas como la dirección, velocidad y la fuerza de rebote cuando la bola colisione con superficies horizontales.

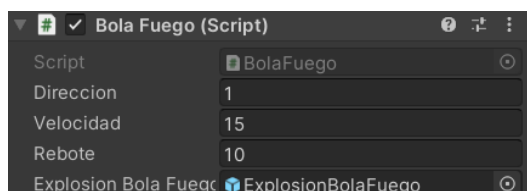


Ilustración 5.35. Parámetros Bola de Fuego

El disparo se realiza cuando se presiona la tecla ‘Z’, que genera una bola de fuego que se desplaza en la dirección que Mario enfrenta. Esta dirección es determinada por la orientación actual de Mario.

Cuando la bola de fuego impacta con un enemigo, se activa una animación de rotación del enemigo, controlada por el ‘Animator’. Este impacto también desactiva el movimiento automático del enemigo y, después de ejecutar la animación, el enemigo es destruido.

Si la bola impacta en un objeto sólido, se genera una animación de explosión utilizando un prefab de explosión creado, que simula el efecto de la bola de fuego desintegrándose.

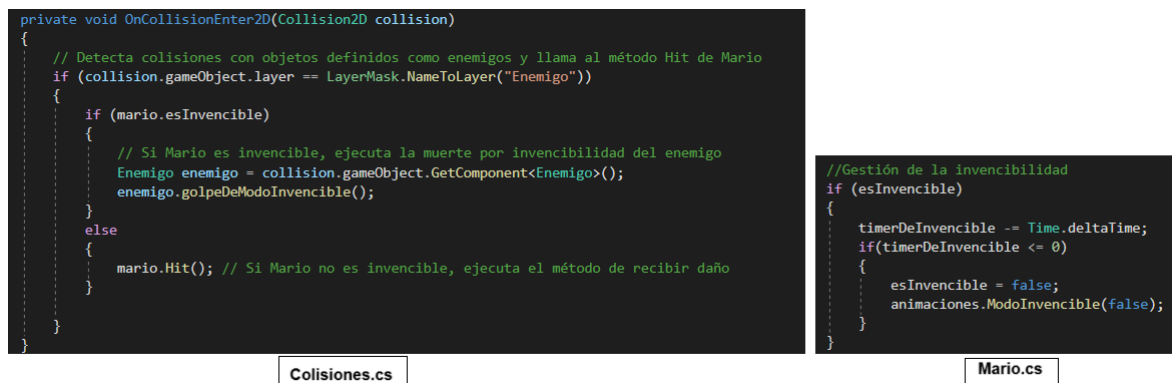
El comportamiento de los enemigos frente a la bola de fuego está configurado para reaccionar de manera específica según el tipo de enemigo. La planta, simplemente se destruye, mientras que el Goomba y Koopa tienen una animación especial de rotación al ser golpeados.



Ilustración 5.36. Prefabs Bola de Fuego

Para la implementación del **efecto de invencibilidad de Mario**, he desarrollado un sistema que activa un estado especial cuando Mario obtiene la Estrella de Invencibilidad. Este estado permite a Mario ser invulnerable a los ataques y daños durante un período de tiempo determinado. El proceso ha incluido varios pasos clave:

1. **Creación del Prefab de la Estrella:** se ha desarrollado un prefab para la Estrella que incluye animaciones específicas para representar su estado activo. Esto implica una secuencia de arsprites que reflejan el brillo y movimiento característico de la Estrella.
2. **Gestión de la Interacción:** Al recoger la Estrella, Mario activa un modo especial donde se cambian sus propiedades físicas y de interacción. Esto se ha hecho mediante scripts que modifican los componentes de movimiento y colisión, permitiendo que Mario no reciba daño.



```
private void OnCollisionEnter2D(Collision2D collision)
{
    // Detecta colisiones con objetos definidos como enemigos y llama al método Hit de Mario
    if (collision.gameObject.layer == LayerMask.NameToLayer("Enemigo"))
    {
        if (mario.esInvencible)
        {
            // Si Mario es invencible, ejecuta la muerte por invencibilidad del enemigo
            Enemigo enemigo = collision.gameObject.GetComponent<Enemigo>();
            enemigo.golpeDeModoInvencible();
        }
        else
        {
            mario.Hit(); // Si Mario no es invencible, ejecuta el método de recibir daño
        }
    }
}
```

Colisiones.cs

```
//Gestión de la invencibilidad
if (esInvencible)
{
    timerDeInvencible -= Time.deltaTime;
    if (timerDeInvencible <= 0)
    {
        esInvencible = false;
        animaciones.ModoInvencible(false);
    }
}
```

Mario.cs

Ilustración 5.37. Modo Invencible

3. **Animación de Invencibilidad:** Se implementó una capa adicional en el sistema de animación de Mario para cambiar los colores del Sprite durante el efecto de invencibilidad, creando una apariencia visual que refleja el estado temporal de invulnerabilidad.

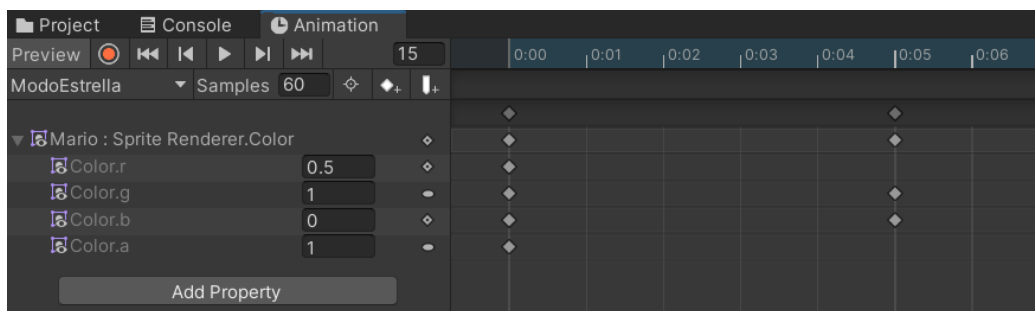


Ilustración 5.38. Animación de Invencibilidad

4. **Control de Tiempo:** El efecto de la Estrella tiene una duración limitada. Se establece un temporizador que, una vez finalizado, revierte a Mario a su estado normal.
5. **Interacciones con Enemigos:** Durante el estado de invencibilidad, cualquier contacto de Mario con los enemigos resulta en la eliminación de estos, igual a cómo afectan las bolas de fuego.

Además, he resuelto algunos **puntos pendientes sobre el estado ‘Herido’ de Mario, otras muertes de los enemigos y varias interacciones y estados específicos del juego** que voy a detallar a continuación:

1. Detección de Enemigos Encima de Bloques:

Para matar a los enemigos situados encima de un bloque cuando Mario golpea ese bloque desde abajo, se ha utilizado un área de colisión configurada justo encima del bloque. Esta caja se genera usando la función **‘Physics2D.OverlapBoxAll’**, que detecta todos los colisionadores dentro de una caja definida en la ubicación específica. Si un enemigo está dentro de esta caja en el momento del golpe, se le aplica la acción de ser eliminado.



Ilustración 5.39. Detección de Enemigos – Items encima del Bloque

2. Capacidad de Romper Bloques:

Solo 'Super Mario' puede romper bloques. Esto se maneja verificando el estado actual de Mario antes de permitir que el bloque se rompa. Si Mario no está en su forma potenciada, el bloque no se romperá y solo se mostrará una animación de rebote.

3. Cambiar la Dirección de los Ítem Encima de los Bloques:

Cuando Mario da un 'cabezado' a un bloque y hay ítems encima de este, la dirección de estos ítems cambia. Esto se implementa detectando ítems sobre el bloque (usando la misma técnica a la detección de enemigos) y aplicando un cambio en su dirección de movimiento.

4. Muerte por 'caparazon' de Koopa a Otros Enemigos:

Cuando un 'caparazón' de Koopa es lanzado por Mario y colisiona con otros enemigos, estos son eliminados. Esto se gestiona detectando colisiones entre el caparazón en movimiento y cualquier enemigo, y luego aplicando la lógica de 'muerte' a esos enemigos. Si colisiona con otro caparazón estos cambiarán de dirección.

5. Interacción con la Planta Piraña:

A diferencia de otros enemigos, Mario no puede 'pisotear' a la Planta Piraña para matarla. Si intenta hacerlo, Mario es el que recibe daño. Esto se controla verificando si el colisionador de Mario entra en contacto con una Planta Piraña y, de ser así, aplicando daño a Mario en lugar de al enemigo.

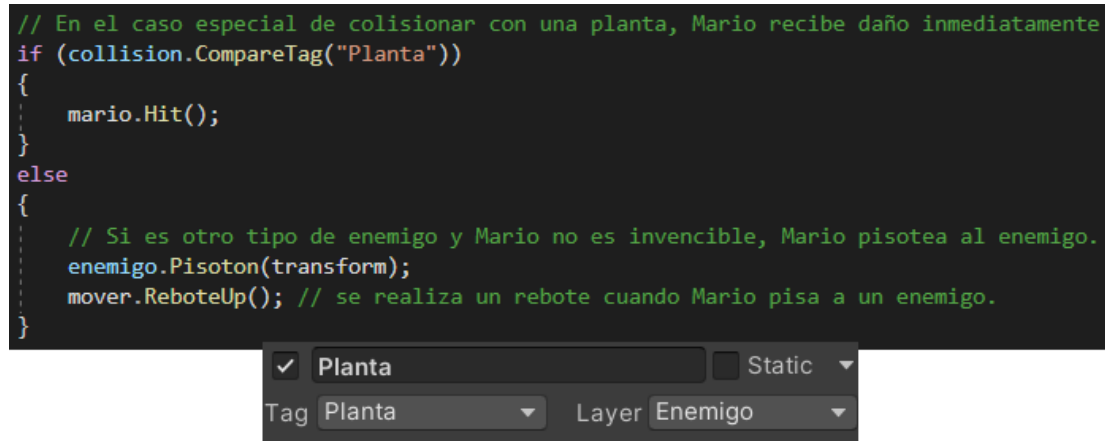


Ilustración 5.40. Interacción con la Planta Piraña

6. Invencibilidad Temporal de Mario tras ser Herido:

Cuando ‘Super Mario’ o ‘Mario Fuego’ recibe un golpe, entra en un estado temporal de invencibilidad, donde parpadea y no puede ser dañado. Durante este tiempo, cualquier colisión con enemigos no resultará en más daño. Este estado se maneja con un temporizador que cuenta la duración de la invencibilidad y restablece la capacidad de Mario para recibir daño una vez que el temporizador expira.

Además, en el estado de invencibilidad, se ajusta las capas de colisión para que Mario pueda atravesar enemigos sin interactuar físicamente con ellos, evitando así que se acumulen daños adicionales durante este período vulnerable.

```
public void ColisionesHerido(bool activado)
{
    // Se cambia las capas de colisión cuando Mario está herido para evitar más daño o interacción
    //No se cambia en todos, ya que mario puede seguir ejecutando pisoton y cabezado cuando esta en estado herido
    if (activado)
    {
        gameObject.layer = LayerMask.NameToLayer("OnlyGround");
        transform.GetChild(1).gameObject.layer = LayerMask.NameToLayer("OnlyGround");
    }
    else
    {
        gameObject.layer = LayerMask.NameToLayer("Player");
        transform.GetChild(1).gameObject.layer = LayerMask.NameToLayer("Player");
    }
}
```

Ilustración 5.41. Colisiones cuando Mario es golpeado

5.2.7 Implementación del Sistema de Puntuación y Audio.

Antes de detallar los puntos mencionados en este apartado, voy a explicar cómo he realizado la implementación y gestión de la bandera en el juego, es decir, el proceso realizado para que Mario interactúe correctamente con este elemento al final de cada nivel:

1. **Creación de Elementos Gráficos:** Como en todos los apartados anteriores, inicio configurando los sprites necesarios para la bandera, sus movimientos y la asta de la bandera.

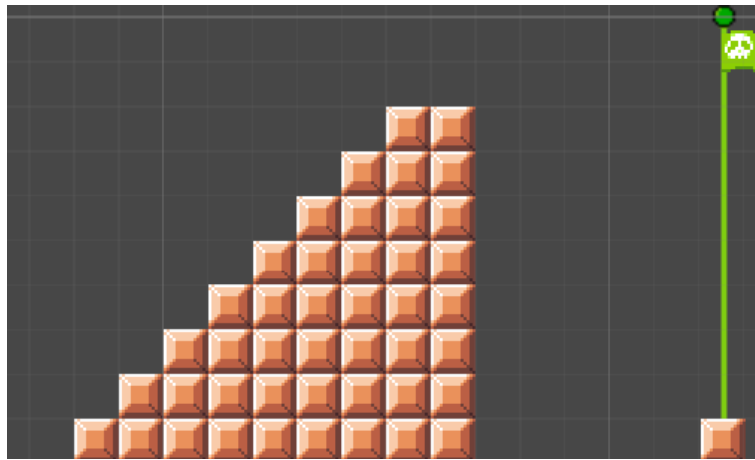


Ilustración 5.42. Sprite de la bandera

2. **Animaciones de Mario para Bajar la Bandera:** Se ha implementado una animación específica donde Mario se desliza hacia abajo por el poste.
3. **Interacción de Mario con la Bandera:**

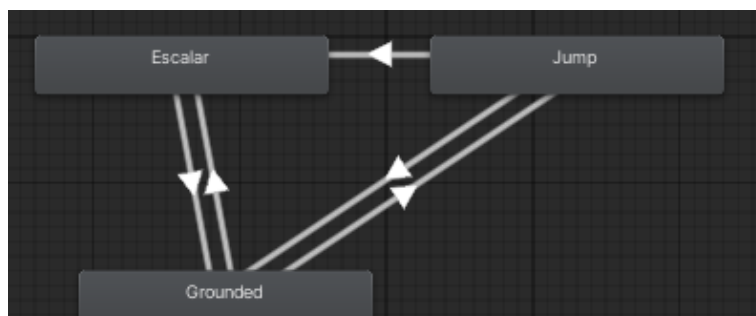


Ilustración 5.43. Animator - Escalar

Inicio del Descenso: Al interactuar con la bandera, se detecta la colisión de Mario con ella y activa el proceso de descenso.

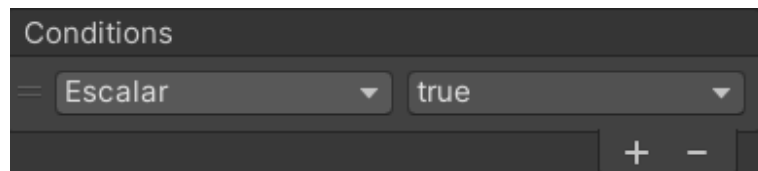


Ilustración 5.44. Condición de la animación Escalar

Control de Descenso: Durante el descenso, Mario se mueve hacia abajo a una velocidad constante que está programada.

Bajar la Bandera: Simultáneamente con el descenso de Mario, la bandera misma, que inicialmente está en la parte superior del poste, también desciende. Esto se logra moviendo la bandera hacia abajo en el poste a la misma velocidad que Mario, creando la ilusión de que Mario la está bajando realmente. Esto se gestiona en el script de VictoriaBandera.

Acciones al Finalizar el Descenso: Una vez que Mario llega al final del poste, se activan varias acciones:

- **Termina la Animación de Descenso:** La animación de descenso se detiene y se activa una nueva animación donde Mario se desmonta del poste, esto se hace trasladando la posición de Mario un poco en el eje x.

```
public void BajarBandera()//función que inicia el proceso de descenso de Mario por el poste de la bandera
{
    inputMoveEnabled = false; // Desactiva el control de entrada del jugador para mover a Mario
    rb2D.isKinematic = true; // Cambia el Rigidbody2D a kinematic para controlar manualmente la posición sin influencia de la física
    rb2D.velocity = new Vector2(0, 0f); //Se detiene el movimiento actual al establecer la velocidad a cero
    escalandoBandera = true;
    isJumping = false;
    animaciones.Jumping(false);
    animaciones.Escalar(true);
    // Se ajusta ligeramente la posición de Mario para alinearla con la posición de la bandera.
    transform.position = new Vector2(transform.position.x + 0.1f, transform.position.y);
}
```

Ilustración 5.45. Función para Bajar de la bandera

- **Caminata Hacia el Castillo:** Después de bajarse del poste, Mario entra automáticamente en una animación de caminata que lo lleva hacia el siguiente nivel, simbolizando la conclusión del nivel actual.

```
IEnumerator SaltoBajarBandera()//Corrutina que gestiona el proceso después de que Mario acabe de descender el poste de la bandera
{
    escalandoBandera = false; // Indica que Mario ha terminado de escalar la bandera
    rb2D.velocity = Vector2.zero;
    animaciones.Pausar();
    yield return new WaitForSeconds(0.25f); // Pequeña pausa antes de iniciar la siguiente parte de la animación

    while (!banderaAbajo) // Espera hasta que la bandera haya llegado completamente abajo
    {
        yield return null; // Continúa esperando cada frame hasta que la condición sea verdadera
    }

    // Se ajusta la posición de Mario para alinearla correctamente después de bajar la bandera
    yield return new WaitForSeconds(0.25f);
    transform.position = new Vector2(transform.position.x + 0.4f, transform.position.y);

    animaciones.Escalar(false);
    rb2D.isKinematic = false;

    //se activa el movimiento automático para que Mario camine hacia el final del nivel
    animaciones.ContinuarAnimacion();
    caminandoAuto = true;
    currentVelocity = velocidadCaminandoAuto;
}
```

Ilustración 5.46. Función para Saltar de la bandera

SISTEMA DE PUNTUACIÓN

Para la gestión del sistema de puntos he utilizado un patrón de diseño llamado **‘Singleton’** [13]. Este patrón asegura que una clase tiene una única instancia y proporciona un punto de acceso global a ella. Esta técnica es útil para manejar componentes como el Score Manager, que necesita ser accesible de manera global y persistente a través de diferentes escenas del juego.

Para implementar el patrón Singleton en el Score Manager, he seguido los siguientes pasos:

1. **Creación de la Instancia Singleton:** He definido una variable privada estática de la misma clase que se encarga de almacenar la instancia, además creo un método público que permite el acceso a esta instancia.

```

public class ScoreManager : MonoBehaviour
{
    public static ScoreManager Instancia;
    public int puntos;
    Ⓜ Mensaje de Unity | 0 referencias
    private void Awake()
    {
        if(Instancia == null)
        {
            Instancia = this;
        }
    }
    // Start is called before the first frame update
    Ⓜ Mensaje de Unity | 0 referencias
    void Start()
    {
        puntos = 0;
    }

    9 referencias
    public void SumarPuntos(int cantidad)
    {
        puntos += cantidad;
    }
}

```

Ilustración 5.47. Score Manager

- 2. Acceso y Gestión de Puntos:** Se puede modificar la puntuación desde cualquier parte del juego.

Al final de cada nivel, cuando Mario baja por el asta de la bandera, los puntos se otorgan según la altura a la que Mario toca la bandera. El asta de la bandera se divide en segmentos, y cada segmento otorga diferentes cantidades de puntos:

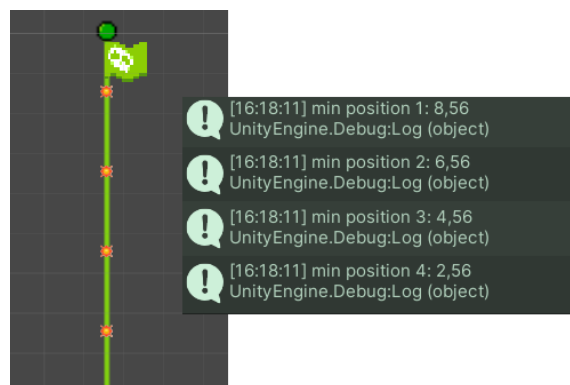


Ilustración 5.48. Alturas a las que Mario puede tocar la bandera

- Zona Superior del Asta: 5000 puntos.
- Segunda Zona desde arriba: 2000 puntos.
- Tercera Zona desde arriba: 800 puntos.
- Cuarta Zona desde arriba: 400 puntos.
- Base del Asta: 200 puntos.

El cálculo de la altura y la asignación de puntos se realizan mediante la posición de Mario en relación con el asta de la bandera al momento de colisionar con ella. Se verifica la posición de Mario en relación con dividir la altura total del asta en cinco segmentos, asignando puntos según la sección alcanzada.

Estos puntos se manejan a través del 'ScoreManager', que utiliza el patrón Singleton para asegurarse de que la gestión de puntos sea accesible globalmente y persista entre diferentes escenas y niveles.

```
void CalcularAltura(float posicionMario)//Puntos conseguidos según la posición de Mario al saltar a la bandera
{
    float size = GetComponent<BoxCollider2D>().bounds.size.y;
    //Debug.Log("tamaño" + size);

    float minimaPosicion1 = transform.position.y + (size - size / 5); //5000 puntos
    //Debug.Log("min position 1: " + minimaPosicion1);

    float minimaPosicion2 = transform.position.y + (size - 2*size / 5); //2000 puntos
    //Debug.Log("min position 2: " + minimaPosicion2);

    float minimaPosicion3 = transform.position.y + (size - 3*size / 5); //800 puntos
    //Debug.Log("min position 3: " + minimaPosicion3);

    float minimaPosicion4 = transform.position.y + (size - 4*size / 5); //400 puntos
    //Debug.Log("min position 4: " + minimaPosicion4);

    int numeroPuntos = 0;
    if(posicionMario >= minimaPosicion1)
    {
        numeroPuntos = 5000;
    }
    else if (posicionMario >= minimaPosicion2)
    {
        numeroPuntos = 2000;
    }
    else if(posicionMario >= minimaPosicion3)
    {
        numeroPuntos = 800;
    }
    else if(posicionMario >= minimaPosicion4)
    {
        numeroPuntos = 400;
    }
    else
    {
        numeroPuntos = 200;
    }
    ScoreManager.Instancia.SumarPuntos(numeroPuntos);
}
```

Ilustración 5.49. *Calculo de la altura y asignación de puntos de la bandera*

Para enriquecer la retroalimentación visual para los jugadores, he implementado efectos visuales que muestran los puntos obtenidos por diversas acciones dentro del juego, ya sea matando a los enemigos o recogiendo ítem.

1. Primero se ha diseñado y utilizado sprites individuales para diferentes cantidades de puntos (100,200,400,800,1000,2000,5000).
2. Para la implementación técnica, se ha creado un prefab que incluye estos sprites y mediante un script se controla la dinámica de aparición y ocultamiento de los sprites.

Estos sprites aparecen y se mueven ligeramente hacia arriba antes de desvanecerse, creado un efecto visual que simula el acto de ganar puntos.

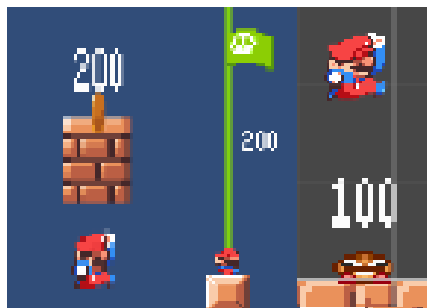


Ilustración 5.50. Efecto puntos

AUDIO

En el desarrollo del juego, se ha dado especial atención a la implementación del sistema de audio. El tratamiento del audio en Unity se ha estructurado en tres categorías principales: música, efectos de sonido y sonidos de ambiente. Cada tipo de audio se ha gestionado de manera específica para optimizar tanto la calidad como el rendimiento del juego [14][15].

Configuración de Audio:

El audio del juego se ha configurado con diferentes métodos de carga para optimizar el rendimiento y la calidad:

- **Descomprimir al Cargar:** Esta técnica se ha utilizado para efectos de sonido cortos y frecuentes, como los sonidos de saltos y recolección de monedas, proporcionando una reproducción sin retardos.
- **Comprimido en Memoria:** Ideal para sonidos que ofrecen un balance entre tamaño y frecuencia de uso.
- **Streaming:** Reservado para archivos de audio de gran tamaño y uso poco frecuente, como largas pistas de música o diálogos, minimizando el uso de la memoria RAM.

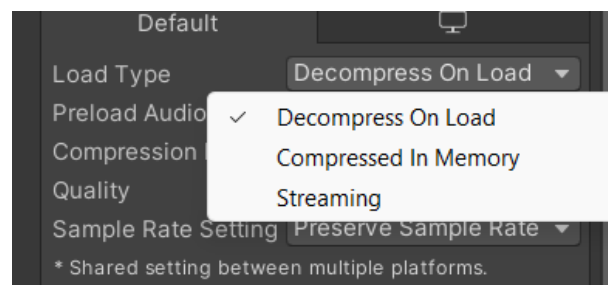


Ilustración 5.51. Tipos de Carga de Audio

Compresión de Audio: Se han seleccionado formatos de compresión adecuados para cada tipo de sonido, considerando la calidad necesaria y el impacto en el rendimiento. He utilizado PCM para efectos donde la inmediatez es crucial, y formatos comprimidos como Vorbis para otros sonidos donde la fidelidad es menos crítica pero deseable.

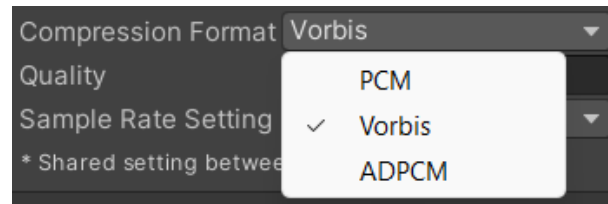


Ilustración 5.52. Tipos de compresión de Audio

Integración de Audio

La integración y gestión del audio se han centralizado a través del componente 'AudioManager', implementado siguiendo el patrón Singleton. Esto asegura que solo existe una instancia de este gestor en todo el juego, facilitando el acceso y control de los sonidos desde cualquier parte del código.

- **AudioSource y AudioClips:** Se han utilizado múltiples AudioSource para separar los sonidos de efectos, música y ambiente.

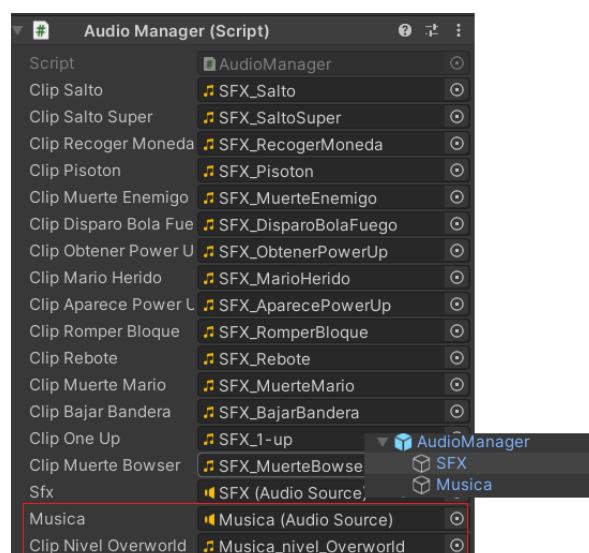


Ilustración 5.53. Audio Source

El AudioManager no solo gestiona la reproducción de sonidos efectivos, sino que también maneja la transición y adaptación de la música ambiental a través de diferentes escenarios del juego:

- **Transiciones Musicales:** Se manejan cambios dinámicos en la música de fondo según el contexto del juego utilizando métodos específicos que ajustan la música a la atmósfera deseada.

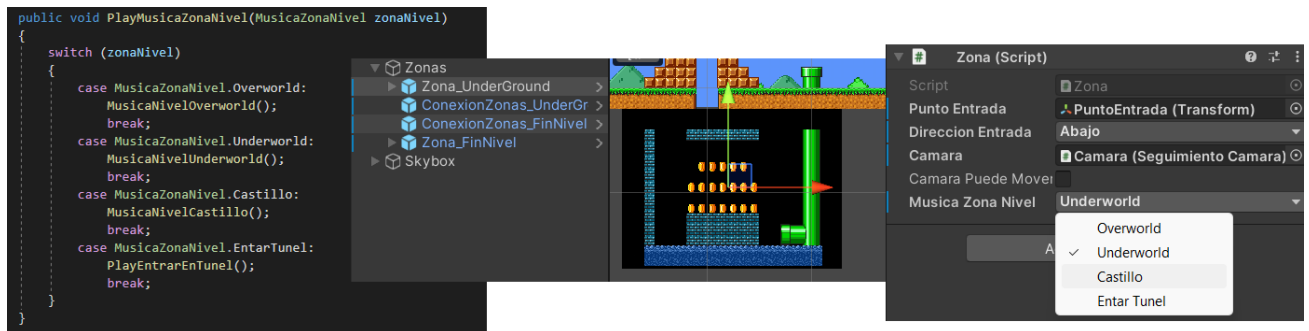


Ilustración 5.54. Transiciones musicales

- **Modo Invencible:** Durante este modo, se reproduce una pista especial, y el sistema está diseñado para volver automáticamente a la música de fondo del nivel una vez que el modo invencible termina.

```
public void MusicaMarioInvencible()
{
    modoInvencible = true;
    musica.clip = clipMarioInvencible;
    musica.loop = true;
    musica.Play();
}

2 referencias
public void PararMusicaMarioInvencible(bool playZonaNivelMusica)
{
    if (modoInvencible)
    {
        modoInvencible = false;
        if (playZonaNivelMusica)
        {
            PlayMusicaZonaNivel(musicaActual);
        }
    }
}
```

Ilustración 5.55. Musica – Modo invencible

5.2.8 Desarrollo de la Interfaz de Usuario y Temporizador

Para abordar la gestión e implementación de la interfaz de usuario en el desarrollo del juego, he creado un sistema que permite interactuar con Mario a través de diversos elementos visuales y configuraciones que reflejan dinámicamente el estado del juego y las acciones del jugador.

1. Componentes Básicos de la UI

Canvas: Es el elemento principal que sirve como contenedor para todos los componentes de la UI. Se utiliza para organizar y renderizar visualmente todos los elementos de la interfaz en la pantalla [16].

EventSystem: Es el sistema que gestiona los eventos de la UI, como clics de botón o entradas táctiles, asegurando que los inputs del usuario sean procesados correctamente [17].

Render Mode: El canvas permite varias configuraciones de renderizado, pero he elegido la opción de 'Screen Space – Overlay', que dibuja la UI por encima de todo lo demás de la pantalla [18].

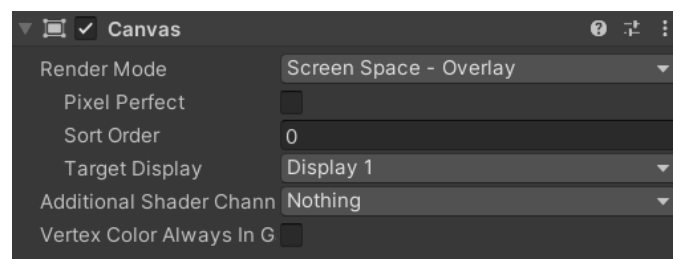


Ilustración 5.56. Render Mode - Canvas

2. Adaptabilidad y Escalabilidad:

Se ha utilizado el componente 'Canvas Scaler' para asegurar que nuestra UI se vea consistentemente en diferentes dispositivos y resoluciones [19].

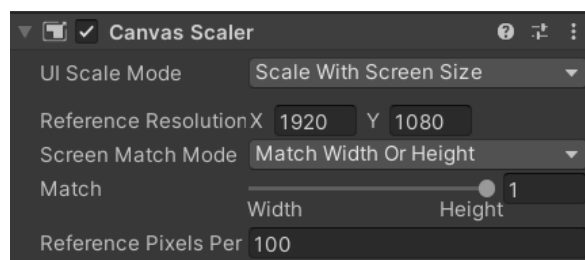


Ilustración 5.57. Canvas Scaler

3. Elementos Interactivos

Botones, Textos e Imágenes: Estos son los elementos fundamentales para mostrar información como puntuación, niveles, estado del jugador y demás. Además, permite interacciones simples como presionar botones para navegar por el juego o cambiar configuraciones.

Text Mesh Pro: Se utiliza para una mejor calidad visual del texto, ofreciendo claridad independiente de la resolución, lo cual es esencial para interfaces como la puntuación y los diálogos dentro del juego [20].

4. Anclajes y Ajustes Dinámicos

Además, he implementado anclajes dinámicos ('Anchors') para que los elementos de UI mantengan su posición relativa y tamaño proporcional respecto al Canvas.

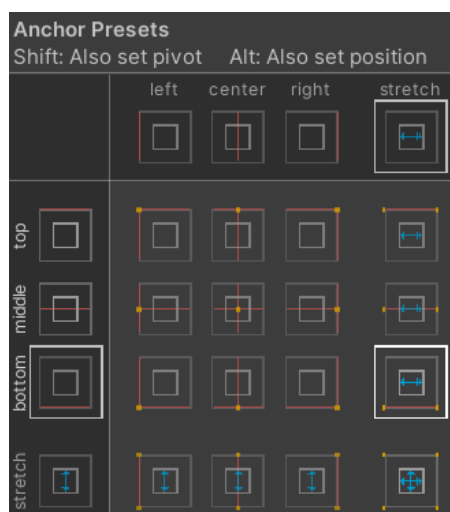


Ilustración 5.58. Anchors Presets

Para la **Creación de la UI**, primero se ha utilizado elementos de la UI de Unity como TextMeshPro para los elementos textuales que muestran la información de puntuación, monedas, nivel en el que nos encontramos y el tiempo. Estos elementos se organizan dentro de un canvas para mantener una jerarquía clara y manejable.

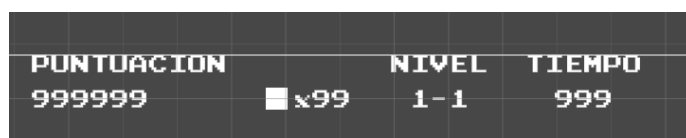


Ilustración 5.59. Vista Previa de la UI

La implementación se ha llevado a cabo mediante los siguientes scripts principales que interactúan con la lógica del juego:

- **ScoreManager:** Gestiona todas las modificaciones de la puntuación.
- **HUD:** Controla la visualización de la información de puntuación, conteo de moneda, nivel actual y temporizador. Este script actualiza dinámicamente los datos en pantalla en respuesta a cambios notificados por el 'ScoreManager' y otros componentes del juego.
- **NivelManager:** Coordina el incremento de monedas y gestiona otros elementos del nivel, como el seguimiento del tiempo y la actualización del HUD correspondiente. A través de este manager, se comunica cualquier cambio necesario en la representación visual de la interfaz.

Además, para la animación de imágenes en la HUD, como la de la moneda, se ha utilizado el script 'AnimacionImagen', que cicla a través de un array de sprites, mostrándolos secuencialmente en un componente 'Image' para crear una animación fluida.



Ilustración 5.60. Vista previa monedas - UI

TEMPORIZADOR

Para gestionar el temporizador en el juego, se ha configurado el HUD para mostrar el tiempo restante, que se maneja mediante 'NivelManager'. El temporizador decrementa cada segundo en tiempo de juego para mantener la sensación de desafío.

Implementación del Temporizador

1. **HUD:** Se ha modificado el HUD para incluir un display del tiempo, que se actualiza dinámicamente. Se emplea una conversión de los valores float del temporizador a enteros para que la visualización sea en números enteros sin decimales, manteniendo la estética clásica de los videojuegos.

```
// Método utilizado para actualizar la cuenta regresiva del tiempo en la interfaz de usuario
3 referencias
public void ActualizarTiempo(float tiempoRestante)
{
    // Se convierte el tiempo a un entero, para no manejar decimales en la cuenta regresiva
    int tiempoRestanteInt = Mathf.RoundToInt(tiempoRestante);
    tiempo.text = tiempoRestanteInt.ToString("D3");
}
```

Ilustración 5.61. Función Actualizar Tiempo en el HUD

2. **NivelManager:** Gestiona el temporizador de cada nivel. Al iniciar, se establece un tiempo inicial que decrece cada segundo de juego. Se usa la función 'Time.deltaTime' para asegurarse que el descuento es independiente de la tasa de frames, lo que proporciona una experiencia uniforme independientemente del hardware.
3. **Finalización del Tiempo:** Si el temporizador llega a cero, causa la muerte de Mario si no se ha completado el nivel.

Cuando Mario completa un nivel antes de que el tiempo asignado expire, el tiempo restante se convierte en puntos adicionales para el jugador (cada segundo son 50 puntos).

Para implementarlo, una vez que Mario alcanza el punto final del nivel, se invoca un método que detiene el temporizador y calcula los puntos finales a partir del tiempo restante. Estos puntos se añaden al puntaje total del jugador mediante el 'ScoreManager'.

```
IEnumerator SegundosToPuntos() // Corrutina que convierte cada segundo restante en 50 puntos
{
    yield return new WaitForSeconds(1f); // Espera un segundo antes de comenzar la conversión
    int tiempoRestante = Mathf.RoundToInt(temporizador);
    while (tiempoRestante > 0) // Mientras queden segundos
    {
        tiempoRestante--;
        hud.ActualizarTiempo(tiempoRestante); // Actualiza la cuenta regresiva en la interfaz de usuario
        ScoreManager.Instancia.SumarPuntos(50); // Suma puntos por cada segundo
        AudioManager.Instancia.PlayRecogerMoneda(); // Reproduce el sonido de recoger moneda por cada segundo convertido
        yield return new WaitForSeconds(0.05f); // Espera 0.05 segundos entre cada adición de puntos
    }
}
```

Ilustración 5.62. Función para convertir cada segundo restante en puntos

5.3 Enriquecimiento y Expansión de Juego

5.3.1 Construcción de Niveles

Para gestionar la construcción de niveles en el juego, se ha utilizado la herramienta Tilemap de Unity, que proporciona un método eficiente y flexible para diseñar entornos de juego de manera rápida y a escala. La principal ventaja de Tilemap es su capacidad para trabajar con cuadrículas, donde cada celda puede ser llenada con "tiles" que son sprites individuales. Esto permite una colocación precisa y fácil de texturas o elementos de juego [21].

El proceso comienza con la creación de un objeto "Grid" que sirve como contenedor para uno o más objetos "Tilemap". Este objeto grid gestiona propiedades comunes como el tamaño de la celda y la separación entre ellas, proporcionando una base uniforme para el diseño del nivel.

Para cada Tilemap, se utilizan "paletas" que son colecciones de tiles. Estas paletas permiten seleccionar y colocar tiles dentro de Tilemap con facilidad.

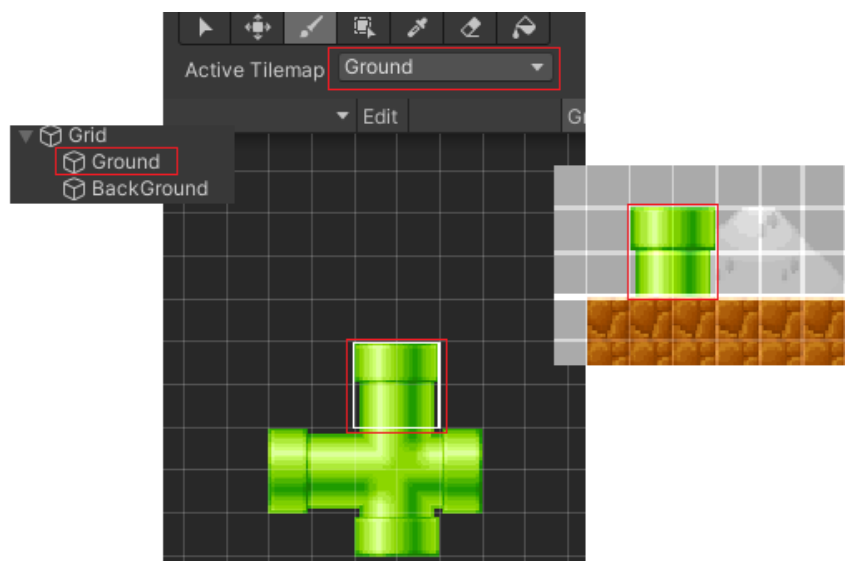


Ilustración 5.63. Utilización de las paletas del Tilemap

Como en el diseño de niveles es importante configurar adecuadamente los layers y colliders, Unity permite añadir automáticamente colliders a los tiles, facilitando la configuración de interacciones físicas sin necesidad de ajustar manualmente cada elemento. Para optimizar el rendimiento y evitar problemas en las interacciones de los colliders, he utilizado el componente Composite Collider, que combina múltiples colliders en uno solo, reduciendo la carga computacional y simplificando el manejo de las colisiones.

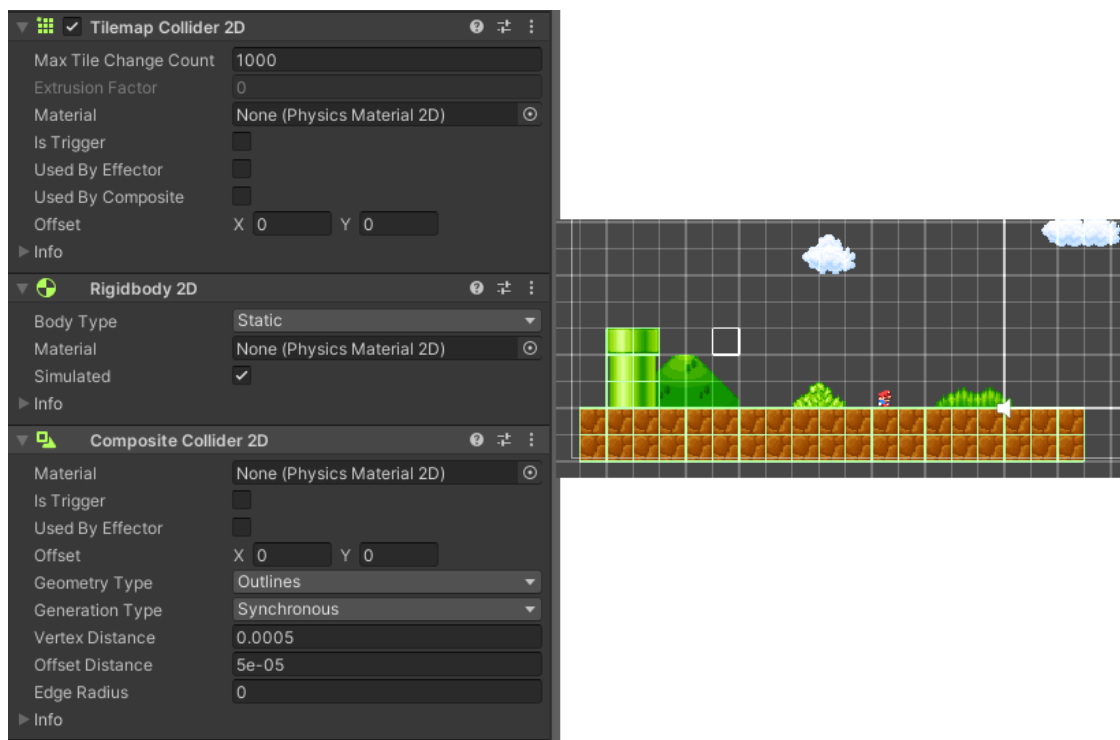


Ilustración 5.64. Utilización del Composite Collider 2D

En todos los niveles, he utilizado una imagen de referencia del juego original, permitiéndome dibujar los elementos del nivel directamente sobre una plantilla visual en Tilemap. Este método asegura que todos los componentes del nivel son colocados de manera adecuado y fiel al original, manteniendo la autenticidad del estilo y la jugabilidad.

PRIMER NIVEL

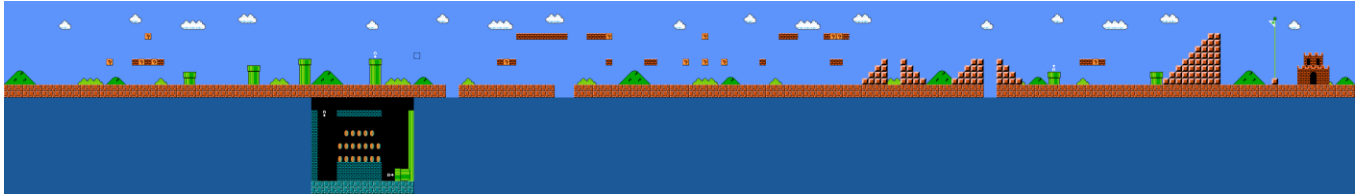


Ilustración 5.65. Mapa nivel 1

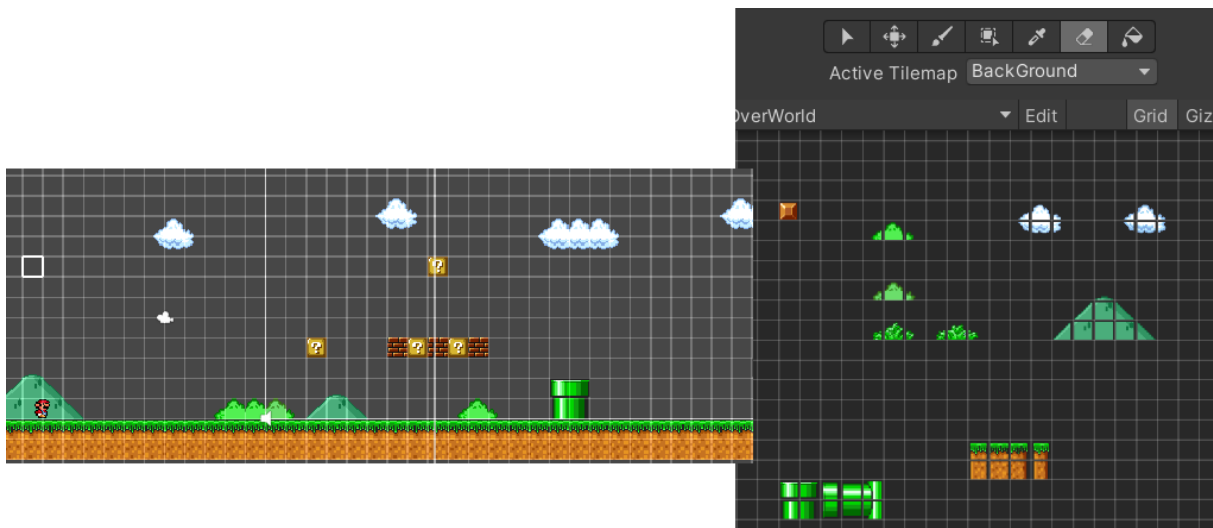


Ilustración 5.66. Mapa nivel 1 - Tilemap

Además, en este primer nivel he ajustado lo siguiente:

- **Bloques invisibles**

He implementado bloques invisibles que se activan cuando Mario los golpea desde abajo. Estos bloques, inicialmente configurados para ser transparentes, mediante el script 'Bloque' cambia su propiedad de visibilidad y colisión en respuesta a la interacción del jugador.


```
private void Start()
{
    //Para el bloque invisible, si el bloque no tiene sprite asociado, desactivar su collider
    if (spriteRenderer.sprite == null)
    {
        boxCollider2D.enabled = false;
    }
}

public void ColisionCabeza(bool marioSuper)
{
    if(spriteRenderer.sprite == null) //Cuando Mario colisiona con la cabeza con el bloque invisible, se activa su collider
    {
        boxCollider2D.enabled = true;
        spriteRenderer.sprite = bloqueVacio;
    }
}
```

Ilustración 5.67. Código Bloques Invisibles

- **Interacción de Bloques de ítems**

Si Mario está en el estado 'Super' y golpea un bloque que liberaría una 'Seta Mágica', el bloque en cambio ofrecerá una flor de fuego.

```
IEnumerator mostrarItem()// función que anima la aparición de un ítem desde el bloque cuando este es golpeado por Mario
{
    AudioManager.Instancia.PlayAparecePowerUp();
    GameObject itemNuevo;
    if(itemFlorPrefab != null && Mario.Instancia.esSuper())//Se controla si Mario es Super para que aparezca un ítem u otro
    {
        //Si es Super, el ítem será una Flor de Fuego
        itemNuevo = Instantiate(itemFlorPrefab, transform.position, Quaternion.identity);// Se crea el ítem en la misma posición del bloque
    }
    else
    {
        //Si no es Super, el ítem será una seta mágica
        itemNuevo = Instantiate(itemPrefab, transform.position, Quaternion.identity);// Se crea el ítem en la misma posición del bloque
    }
}
```

Ilustración 5.68. Cambio de Seta Mágica a Flor de Fuego

- **Uso del GameObject 'Brush' para colocación de Objetos**

Para la colocación de objetos como monedas y bloques dentro del nivel, se ha utilizado el GameObject 'Brush'. Esta herramienta permite 'pintar' objetos directamente en el Tilemap, lo cual facilita la distribución uniforme y precisa de los elementos a lo largo de nivel.

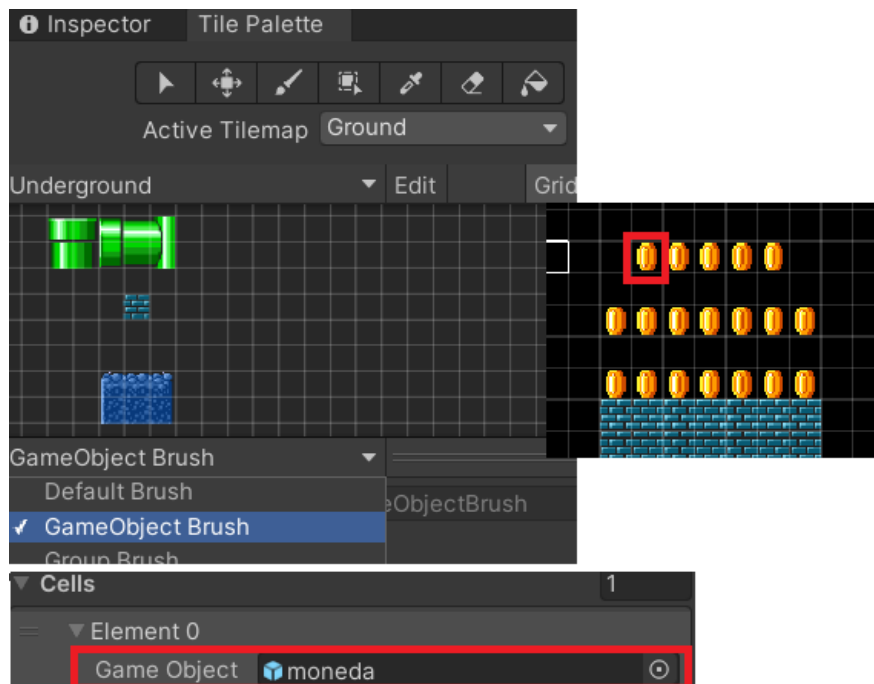


Ilustración 5.69. GameObject Brush

Al seleccionar el tipo de objeto, se puede fácilmente colocar estos elementos en la escena movimiento el cursor y haciendo clic en las ubicaciones deseadas dentro de la cuadrícula de Tilemap

SEGUNDO NIVEL

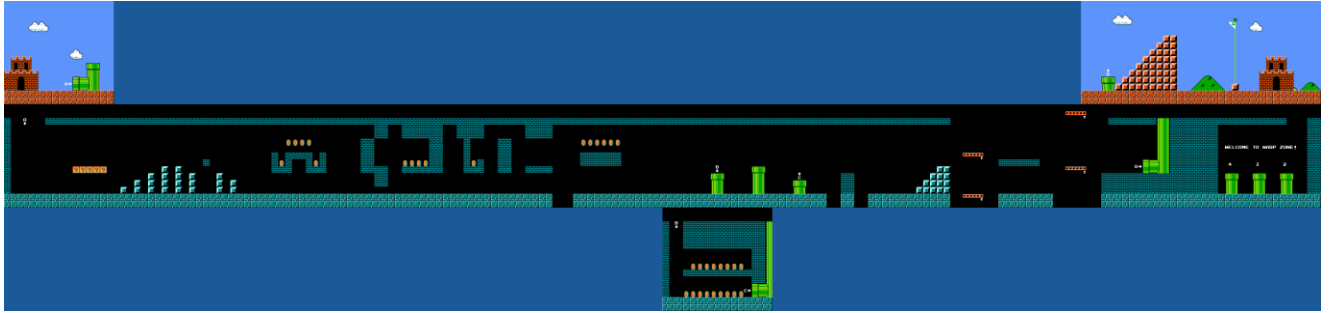


Ilustración 5.70. Mapa nivel 2

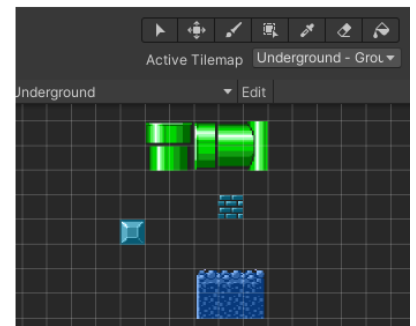
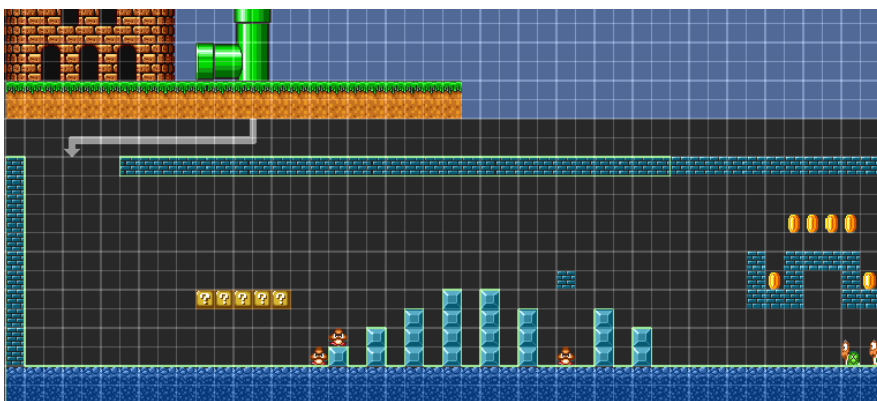


Ilustración 5.71. Mapa nivel 2 - Tilemap

Además, en este segundo nivel he ajustado lo siguiente:

- **Plataformas**

Para implementar y gestionar las plataformas móviles, he seguido los siguientes pasos:

1. **Configuración de la Plataforma:** Como se ha hecho con todos los objetos, se ha añadido el Sprite de plataforma a la escena y se ha agregado un 'BoxCollider2D' para que Mario pueda interactuar físicamente con ella.

2. Implementación del Comportamiento de Plataforma

He utilizado el 'PlatformEffector2D', que permite crear un comportamiento de plataforma donde Mario puede saltar desde abajo a la plataforma.

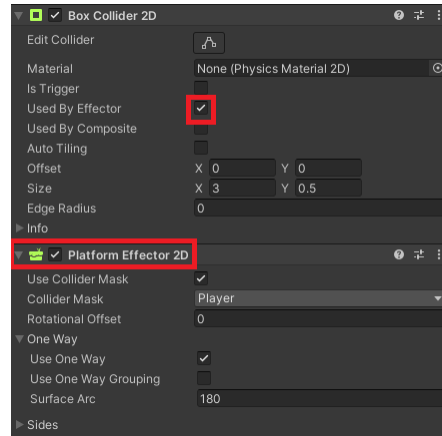


Ilustración 5.72. Platform Efecto 2D

3. Scripting del Movimiento

He creado un script que maneja el movimiento de la plataforma entre dos puntos definidos (punto de inicio y punto final). Este script permite configurar la velocidad del movimiento, la dirección (horizontal, vertical o libre) y si el movimiento es continuo o de solo ida y vuelta.

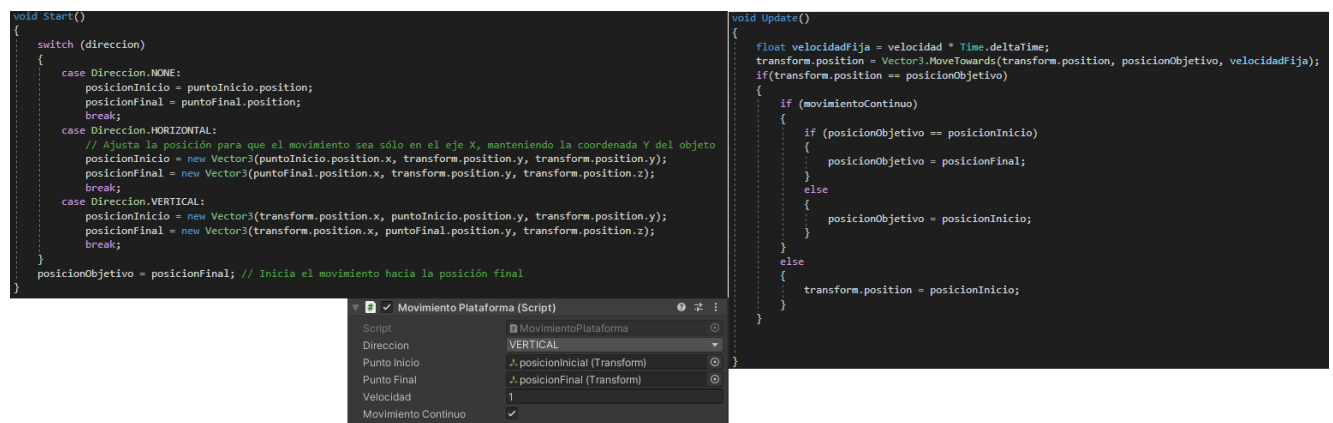


Ilustración 5.73. MovimientoPlataforma.cs

4. Integración con Mario

Para la correcta interacción con Mario, se asegura que cuando Mario salta sobre la plataforma en movimiento, él se mueve junto con ella. Esto se consigue haciendo que Mario sea hijo de la plataforma mientras está en contacto con la misma.

Cuando Mario deja la plataforma, se revierte el parentesco para que Mario vuelva a ser independiente del movimiento de la plataforma.

TERCER NIVEL



Ilustración 5.74. Mapa nivel 3

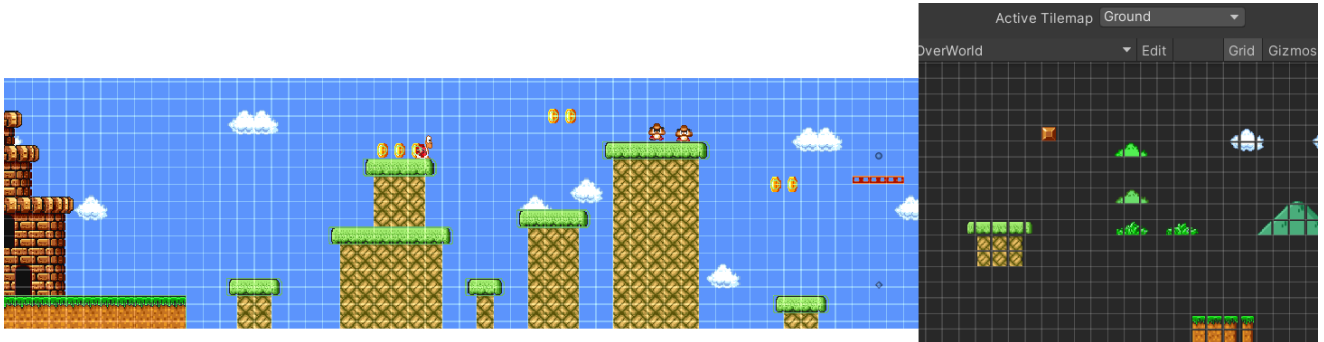


Ilustración 5.75. Mapa nivel 3 – Tilemap

CUARTO NIVEL

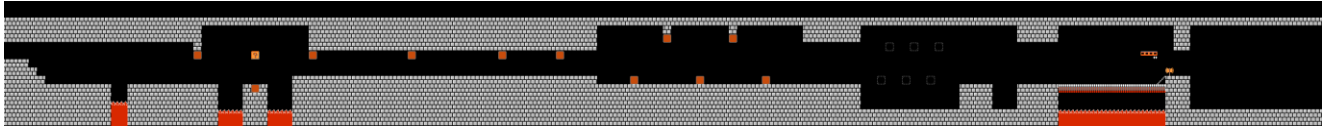


Ilustración 5.76. Mapa nivel 4

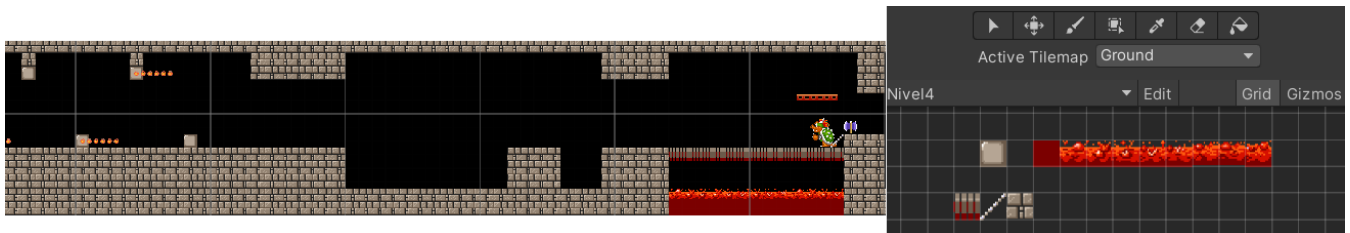


Ilustración 5.77. Mapa nivel 4 - Tilemap

Además, en este cuarto nivel he añadido lo siguiente:

- **Lava**

Se ha creado una nueva capa llamada 'Lava' para manejar específicamente las interacciones con este elemento.

He programado la lava para que cause daño o la muerte a Mario al contacto, dependiendo de su estado. Esto lo he implementado modificando el script de colisiones en el método OnCollisionEnter2D.

```
if(collision.gameObject.layer == LayerMask.NameToLayer("Lava"))
{
    if (!mario.esInvencible)
    {
        mario.Hit(); // Ejecuta el método Hit de Mario, indicando que ha sufrido daño por la lava
    }
}
```

Ilustración 5.78. Daño causado por colisionar con Lava

- **Barra de Fuego**

Las barras de Fuego se han diseñado como objetos rotativos que son colocados a lo largo de los niveles.

Se ha creado un script específico (BarraFuego) para controlar la rotación continua de las barras de fuego, usando la propiedad 'velocidadRotacion' para ajustar su rapidez.

Las barras de Fuego se configuraron para interactuar con Mario, utilizando colliders y ajustando el script de colisiones para gestionar los efectos de contacto.

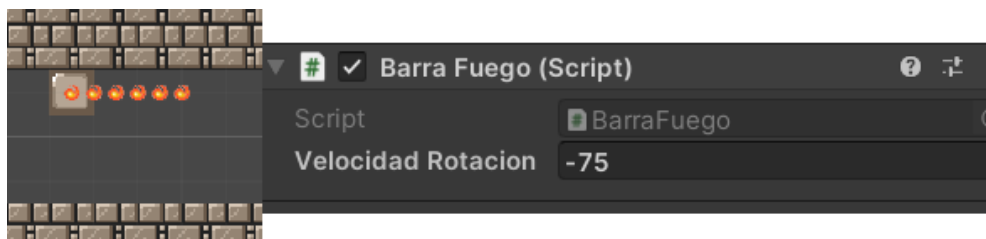


Ilustración 5.79. Barras de Fuego

PUENTE Y FINAL:

- **Creación del puente:** Se creó un objeto vacío 'puente' que sirve como contenedor para todas las piezas del puente.
- **Interacción con el Hacha:** Se creó un Sprite para el hacha al final del puente. Al detectar la colisión con Mario, mediante el script 'HachaFinal', se inicia una corrutina que destruye las piezas del puente una a una con un pequeño retardo entre cada acción, para dar tiempo a visualizar cómo se desmorona el puente.

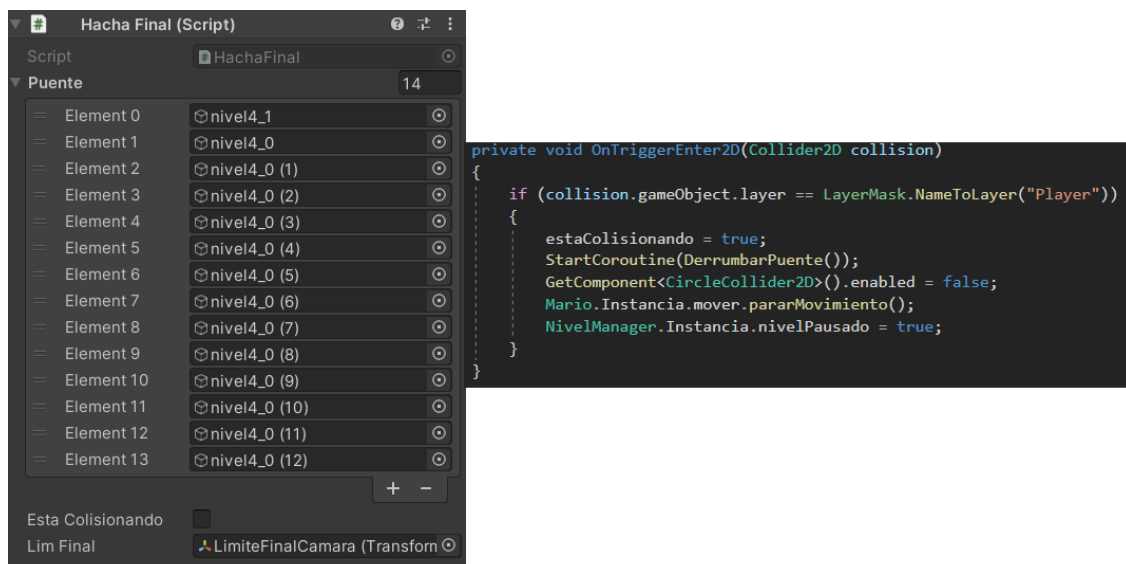


Ilustración 5.80. HachaFinal.cs

- **Finalización del Nivel:** Tras la destrucción del puente, se gestionó la transición hacia el encuentro final con Toad mediante la activación de un canvas que contiene los mensajes finales del nivel.

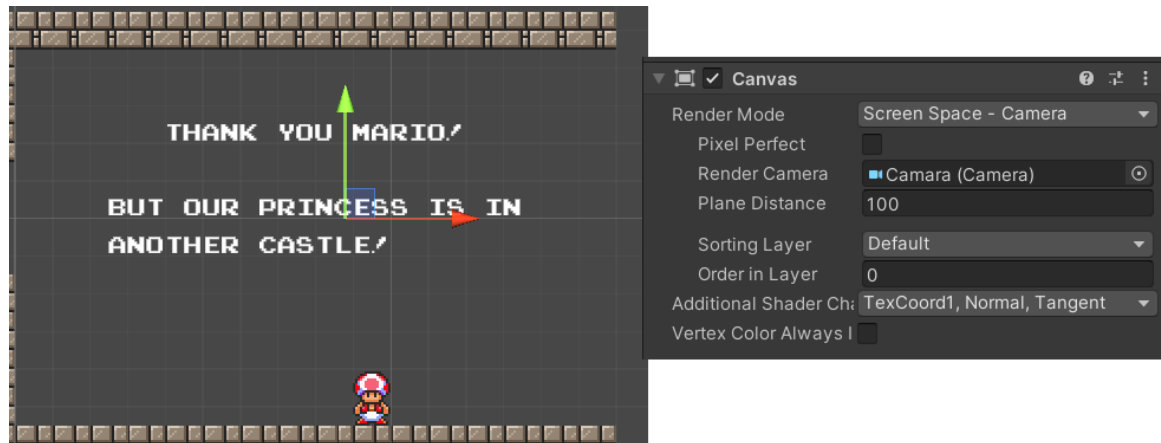


Ilustración 5.81. Finalización del Nivel – Interacción con Toad

Se añadió un collider alrededor de Toad para detectar cuando Mario lo alcanza después de que el puente se destruye. Al tocar a Toad, se detiene a Mario y se activa el canvas con los textos, los cuales se revelan de forma secuencial.

El canvas utilizado para los mensajes finales fue configurado para operar en el espacio de la cámara, asegurando que los textos se mostraran correctamente en relación con la vista del jugador, sin importar la resolución o la configuración de la pantalla.

BOWSER:

Bowser es implementado como una clase derivada de Enemigo, lo que le permite heredar y extender funcionalidades comunes a todos los enemigos, como la gestión de animaciones y movimientos básicos. Sin embargo, he añadido características únicas para adaptarlo a los desafíos específicos que presenta este jefe final:

- **Control de Movimientos y Ataques:**

Movimiento: Bowser tiene la capacidad de moverse hacia adelante y hacia atrás en relación a la posición de Mario, lo cual se controla mediante una variable 'puedeMoverse' que se activa cuando Mario entra en un rango mínimo definido. Esto hace que Bowser inicie su aproximación sólo cuando Mario está suficientemente cerca, aumentando la tensión y la dificultad de la confrontación.

Salto: Se implementa un temporizador que gestiona los intervalos de salto, añadiendo una capa de imprevisibilidad al comportamiento de Bowser. Los saltos se utilizan tanto para esquivar ataques como para intentar aplastar a Mario si está debajo.

Disparo de Fuego: Similar a los saltos, un temporizador controla los intervalos entre disparos. Bowser dispara bolas de fuego que se dirigen hacia Mario.

```
protected override void Update()
{
    // Este bloque verifica si Bowser debería moverse o disparar basado en la distancia a Mario
    if (!puenteDerrumbandose)
    {
        // Control de movimiento basado en la proximidad de Mario
        if (!puedeMoverse && Mathf.Abs(Mario.Instancia.transform.position.x - transform.position.x) <= minDistanciaParaMover)
        {
            puedeMoverse = true;
        }
        if (puedeMoverse)
        {
            // Ajuste de dirección de Bowser basado en la posición de Mario
            if (transform.position.x >= (Mario.Instancia.transform.position.x + 2f) && direccion == 1)
            {
                direccion = -1;
                transform.localScale = Vector3.one;
            }
            else if (transform.position.x <= (Mario.Instancia.transform.position.x - 2f) && direccion == -1)
            {
                direccion = 1;
                transform.localScale = new Vector3(-1, 1, 1);
            }

            // Movimiento de Bowser
            rb2D.velocity = new Vector2(direccion * velocidad, rb2D.velocity.y);

            // Control de saltos
            jumpTimer -= Time.deltaTime;
            if (jumpTimer <= 0)
            {
                Saltar();
            }
        }

        // Control de disparos
        if (!puedeDisparar &&
            Mathf.Abs(Mario.Instancia.transform.position.x - transform.position.x) <= minDistanciaParaDisparar)
        {
            puedeDisparar = true;
        }
        if (puedeDisparar)
        {
            timerDisparar -= Time.deltaTime;
            if (timerDisparar <= 0)
            {
                Disparar();
            }
        }
    }
}
```

Ilustración 5.82. Control de Bowser

- **Gestión de Salud y Respuestas a Ataques:**

Bowser cuenta con un sistema de salud que decrece cada vez que es golpeado por las bolas de fuego de Mario. Una vez que la salud llega a cero, se activa una secuencia de muerte que implica animaciones específicas para mostrar su derrota, lo que incluye cambios en la interacción con el entorno al convertirse en una parte no colisionable.

```
public override void golpeBolaFuego()
{
    rb2D.velocity = Vector2.zero;
    saludBowser--;
    if(saludBowser <= 0)
    {
        giroMuerte();
        estaMuerto = true;
        gameObject.layer = LayerMask.NameToLayer("OnlyGround");
    }
}
```

Ilustración 5.83. Gestión muerte y salud de Bowser

- **Interacción con el Entorno:**

Derrumbe del Puente: Un elemento crucial en la batalla contra Bowser es el puente que se colapsa cuando Mario toca el hacha al final de la plataforma. Esto inicia una secuencia donde las piezas del puente se destruyen progresivamente, con Bowser cayendo al vacío si aún está vivo.

PlayerPrefs permite que los datos persistan incluso después de cerrar el juego, lo que es esencial para mantener la integridad del sistema de puntuaciones.

5.3.2 Gestión de Vidas y Sistema de Respawn

La implementación de vidas de Mario la he realizado de la siguiente manera:

Sistemas de Vidas:

- **Inicialización de Vidas:** Al comenzar una partida, el 'GameManager' inicializa las vidas de Mario a tres.
- **Pérdidas de Vidas:**

Las vidas se pierden cuando Mario cae en zonas de muerte o cuando el tiempo se agota. Estas zonas están gestionadas en el juego mediante un collider, que detecta cuando Mario entra en contacto con ella y llama a 'GameManager' para manejar la pérdida de vida.

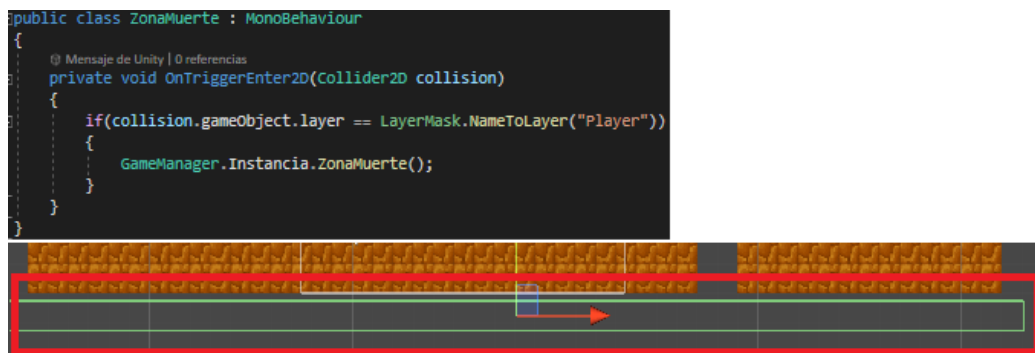


Ilustración 5.84. Zona Muerte

- **Game Over:** Si las vidas llegan a cero, el juego termina.

Recuperación de Vidas:

- **Por Recolección de Monedas:** Cada vez que Mario alcanza un total de 100 monedas, no solo se resetea el contador de monedas, sino que también se otorga una vida extra.
- **Por ítems:** Cuando Mario recoge el ítem 'Seta de Vida', le otorga una vida adicional directamente.

SISTEMA DE RESPAWN

El proceso de Respawn comienza cuando Mario muere, ya sea por caer al vacío (Zona de muerte) o al agotarse el tiempo. En estos eventos, el script '*GameManager*' es el encargado de iniciar el proceso de respawn.

Detección de la muerte de Mario y activación del respawn

- **GameManager:** Gestiona la muerte de Mario controlando las vidas restantes y ejecutando la corrutina de respawn cuando es necesario.

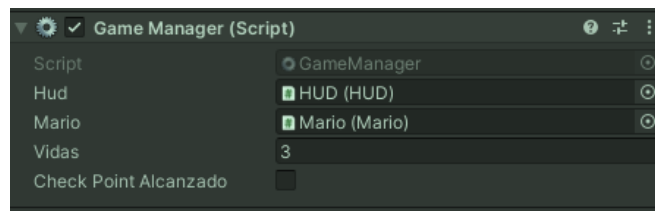


Ilustración 5.85. GameManager

- **CheckPointAlcanzado:** Esta variable es utilizada para determinar si Mario debe respawnear en el checkpoint o en el inicio del nivel.

Implementación de CheckPoints

- Se ha colocado un GameObject con un collider configurado como Trigger en los puntos clave del nivel. El script asociado '*CheckPoint*' detecta la interacción de Mario con ese punto, como se ha hecho con las Zonas de Muerte.



Ilustración 5.86. CheckPoint

Cada checkpoint tiene asignado un 'id' y una posición inicial del jugador que determina dónde debe reaparecer Mario. Este sistema es manejado además por el script 'GameManager' que registra el último checkpoint activado y maneja la lógica de respawn desde ese punto.

Preservación de estados entre la carga de las escenas

- **DontDestroyOnLoad:** Se aplica a los objetos 'GameManager', 'AudioManager', 'ScoreManager' y 'Mario' para mantener su estado después de recargar la escena. Esto asegura que, a pesar de la recarga de la escena para el respawn, la información como las puntuaciones y la configuración de audio permanezcan intactas. [22]

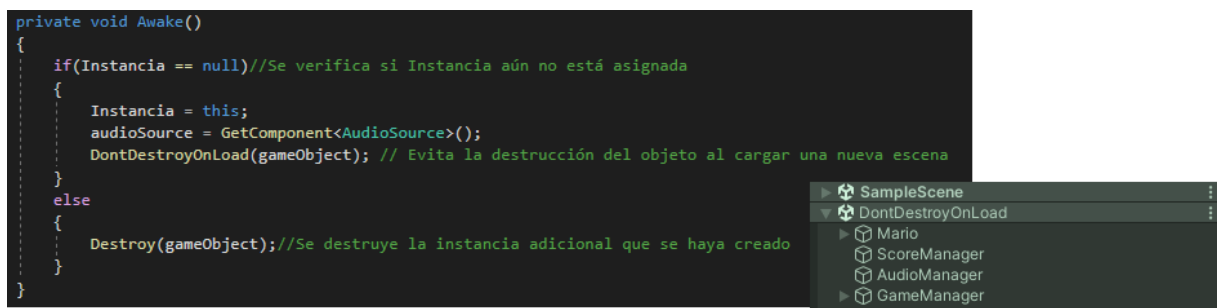


Ilustración 5.87. Uso de 'DontDestroyOnLoad()'

Recarga de la escena

- En lugar de reposicionar manualmente a todos los objetos y enemigos del nivel, la escena completa se recarga utilizando 'SceneManager.LoadScene()'.

```
IEnumerator Reaparicion()//corrutina que maneja la reaparición tras una muerte
{
    yield return new WaitForSeconds(1f); //Espera 1 segundo antes de reaparecer
    estaReapareciendo = false;
    SceneManager.LoadScene(0); //Recarga la escena actual para reiniciar el nivel
}
```

Ilustración 5.88. Uso de 'SceneManager.LoadScene()'

Ajuste del sistema de cámara y otros elementos después del respawn

- Después de un respawn, para que la cámara se inicialice en la posición nueva de Mario he implementado la función 'EmpezarSeguir' en 'NivelManager' para reajustar la posición de la cámara según la nueva posición de Mario.

```
public void EmpezarSeguir(Transform t) //método para reorientar la cámara
{
    objetivoSeguir = t;
    float newPosicionX = objetivoSeguir.position.x + unidadesPorDelanteObjetivo;
    newPosicionX = Mathf.Clamp(newPosicionX, ultimaPosicion, maxPosicionX);
    transform.position = new Vector3(newPosicionX, transform.position.y, transform.position.z);
    ultimaPosicion = newPosicionX;
}
```

Ilustración 5.89. Modo Invencible

5.3.3 Sistema de Guardado

Para implementar y gestionar el sistema de guardado en el videojuego, se desarrolló una estrategia eficiente que asegura la persistencia de los datos esenciales del jugador entre sesiones. Este sistema se basa en el uso de la clase 'PlayerPrefs' de Unity para almacenar y recuperar la información del progreso del jugador. [23]

Implementación del Sistema de Guardado

- **Gestión de Puntuaciones (ScoreManager)**

Al final de cada partida, se verifica si la puntuación actual supera la máxima puntuación previamente guardada. Si es así, se actualiza el valor y se guarda en 'PlayerPrefs' bajo la clave "Puntos". Esto asegura que la puntuación máxima siempre esté actualizada y disponible para sesiones futuras.

```
public void GameOver()
{
    // Comprueba si la puntuación actual es mayor que la máxima puntuación guardada
    if (puntos > maxPuntos)
    {
        maxPuntos = puntos; // Actualiza la máxima puntuación
        PlayerPrefs.SetInt("Puntos", maxPuntos); // Guarda la nueva máxima puntuación en las preferencias del jugador
    }
}
```

Ilustración 5.90. Puntuación Máxima – ScoreManager.cs

- **Gestión del Progreso del Juego (GameManager)**

Cuando el jugador pierde o el juego termina, se guardan el mundo y nivel actual en 'PlayerPrefs'. Esto no solo incluye puntuaciones, sino también el progreso exacto en términos de ubicación en el juego.

Además, al iniciar el juego, se recuperan los valores de mundo y nivel de 'PlayerPrefs'. Si no hay valores previos, se establecen valores predeterminados. Este sistema permite que el jugador retome el juego exactamente donde lo dejó.

```
void Start()
{
    vidas = 3;
    monedas = 0;
    hud.ActualizarMonedas(monedas);
    // Se carga las preferencias guardadas para mundo y nivel o se establece valores por defecto si no existen
    mundoActual = PlayerPrefs.GetInt("Mundo", 1); // Se carga el último mundo guardado, por defecto es 1
    nivelActual = PlayerPrefs.GetInt("Nivel", 1); // Se carga el último nivel guardado, por defecto es 1
}

void GameOver()//Método para manejar el estado de juego terminado
{
    Debug.Log("Game Over");
    ScoreManager.Instancia.GameOver();
    gameOver = true;

    // Se guarda los índices del mundo y nivel actuales en PlayerPrefs para preservar el progreso
    PlayerPrefs.SetInt("Mundo", mundoActual); // Guarda el mundo actual
    PlayerPrefs.SetInt("Nivel", nivelActual); // Guarda el nivel actual

    StartCoroutine(Reaparicion()); // Inicia el proceso de reaparición incluso después de un game over
}
```

Ilustración 5.91. Recuperación de los valores de nivel y mundo

5.3.4 Implementación y Gestión del Menú de Inicio

El menú inicial sirve como puerta de entrada al juego, ofreciendo opciones para comenzar una nueva partida o continuar una existente, además de mostrar el puntaje máximo alcanzado.

Componentes del Menú:

- **Interfaz de Usuario:**

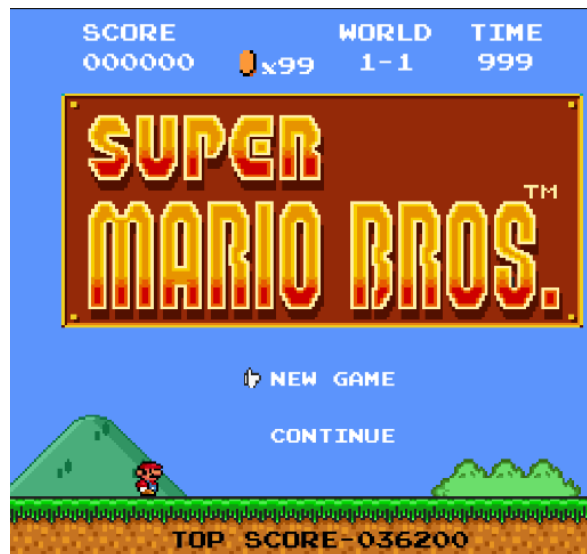


Ilustración 5.92. Menu Inicial

Top Score: Utilizando PlayerPrefs, el sistema recupera y muestra el puntaje más alto alcanzado, presentado mediante TextMeshProUGUI. Esto no solo informa al jugador de su mejor rendimiento anterior, sino que también añade un elemento competitivo al menú.

Botones de Interacción: El menú incluye dos botones principales gestionados a través de EventSystem de Unity:

- **Nuevo Juego:** Al seleccionar esta opción, se reinicia el progreso del juego, estableciendo el nivel y el mundo a su estado inicial y comenzando una nueva partida.
- **Continuar Juego:** Esta opción permite al jugador retomar su última partida guardada, cargando los niveles y puntos desde donde lo dejó.

- **Gestión de Estados:**

Script ‘MenuInicial’: Encargado de activar las configuraciones iniciales como mostrar los puntos y posicionar a Mario. Además, define el comportamiento de los botones del menú, asignando las acciones correspondientes a cada uno para iniciar o continuar el juego.

```
void Start()
{
    // Se obtiene la puntuación máxima guardada de PlayerPrefs y se muestra en el menú
    int puntos = PlayerPrefs.GetInt("Puntos");
    topScore.text = "TOP SCORE-" + puntos.ToString("D6");

    // Se reposiciona a Mario en la posición inicial definida para el menú
    Mario.Instancia.Reaparecer(marioPosicionInicial);

    // Establece el botón de nuevo juego como el objeto seleccionado por defecto en el sistema de evento
    EventSystem.current.SetSelectedGameObject(botonNuevoJuego);
}
```

Ilustración 5.93. MenuInicial.cs

Script ‘BotonMenu’: Maneja la visualización del icono de un cursor al seleccionar cada botón, mejorando la interacción visual del usuario con el menú mediante feedback visual directo cuando un botón es seleccionado o deseleccionado. [24]

```
//Método que se llama cuando el botón asociado es seleccionado
0 referencias
public void OnSelect(BaseEventData eventData)
{
    iconoCursor.SetActive(true); // Se activa el icono del cursor, haciéndolo visible
}
0 referencias
public void OnDeselect(BaseEventData eventData) // Método que se llama cuando el botón pierde la selección
{
    iconoCursor.SetActive(false);
}
```

Ilustración 5.94. BotonMenu.cs

5.3.5 Conexión entre Zonas del nivel y Transición de Niveles

CONEXIÓN DE LAS DISTINTAS ZONAS DEL NIVEL MEDIANTE TUBERÍAS

Las tuberías son elementos que sirven como conexiones entre diferentes zonas del nivel. Para implementar la conexión entre las diferentes zonas del nivel he seguido el siguiente proceso:

1. Configuración de Colliders

Primero he creado dos prefabs:

ConexionZonas: Este prefab incluye un collider que detecta la presencia de Mario y activa el proceso de transición. El script 'ConexionZonas' está asociado con este prefab para gestionar la detección y la acción de transición.

Zona: Este prefab representa la nueva zona a la que Mario se moverá. Incluye el punto de entrada, que especifica dónde Mario aparecerá en la nueva zona.

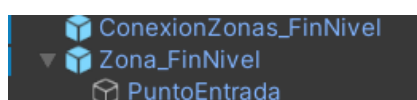


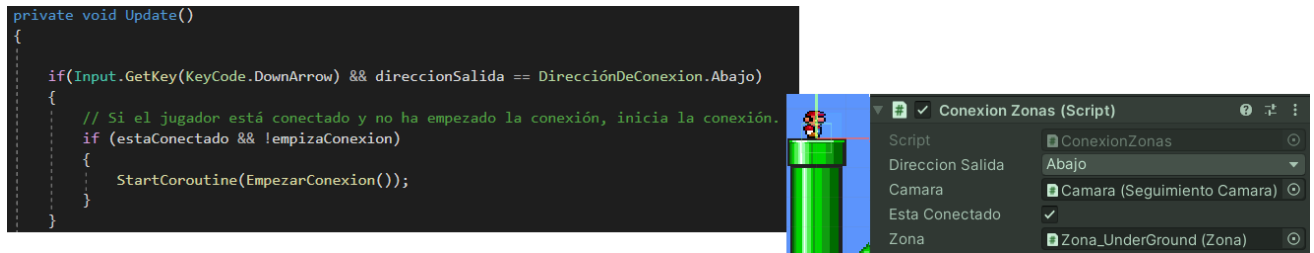
Ilustración 5.95. Prefabs ConexionZonas_ y Zonas_

Además, en el prefab 'ConexionZonas' se configura un collider que funciona como trigger, el cual se activa cuando Mario colisiona, y el script 'ConexionZonas' prepara y ejecuta la transición.

2. Script 'ConexiónZonas' para Detección y Transición

Este script detecta cuando Mario está en la posición correcta para la transición, es decir, esta sobre la tubería y presionar la tecla adecuada.

Cuando se cumplen las condiciones, el script inicia la corrutina que maneja la transición de Mario a la nueva zona. Esto implica deshabilitar temporalmente el control del jugador, ajustar la cámara y mover a Mario hacia el punto de entrada a la nueva zona.

*Ilustración 5.96. ConexionZonas.cs*

3. Script 'Zona' para gestionar la llegada

El script 'Zona' está configurado para recibir a Mario. Una vez que Mario es trasladado a la nueva zona, este script lo coloca en el punto de entrada y reactiva cualquier control o ajuste necesario, es decir, la cámara y los controles del jugador.

```
void LlegarZona()//Método que se llama cuando Mario llega a la zona
{
    Mario.Instancia.mover.ResetearMovimiento(); // Resetear los controles de movimiento de Mario
    NivelManager.Instancia.nivelPausado = false; // Reanuda el nivel para continuar con el juego
    if (CamaraPuedeMoverse)
    {
        camara.EmpezarSeguir(Mario.Instancia.transform); // Si la cámara esta márcada que puede moverse, empieza a seguir a Mario
    }
}
```

Ilustración 5.97. Zona.cs

TRANSICIÓN DE NIVELES

Para implementar las transiciones de niveles, el proceso lo he estructurado en los siguientes pasos:

1. Castillo

Cuando Mario entra en colisión con el castillo, el script lo detecta y desplaza a Mario a una posición fuera de la pantalla para simular que ha entrado al castillo, y además mira si se ha completado la transición de segundos a puntos para pasar de nivel.

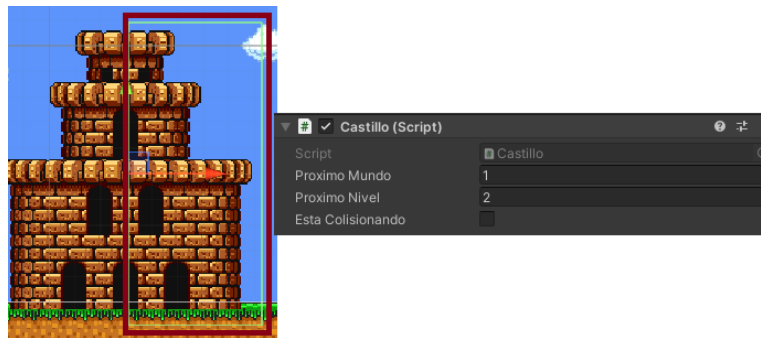


Ilustración 5.98. Castillo.cs

2. Manejo de la Transición con ‘GameManager’

Si se cumplen las condiciones adecuadas (Mario ha colisionado con el castillo y ha finalizado la transición de segundos a puntos), el ‘GameManager’ gestiona la transición al próximo nivel.

El ‘GameManager’ carga la escena de transición, esta escena se utiliza para mostrar información sobre el progreso del juego, número de vidas restantes, mundo y nivel a los que Mario se dirige.

```
void CargarTransicion()// Se carga la pantalla de transición
{
    SceneManager.LoadScene("Transicion");
    Invoke("CargarNivel", 5f);
}

0 referencias
void CargarNivel()
{
    foreach(Mundo m in mundos)
    {
        if(m.id == mundoActual)
        {
            foreach(Nivel n in m.niveles)
            {
                if(n.id == nivelActual)
                {
                    SceneManager.LoadScene(n.nombreEscena);
                    return;
                }
            }
        }
    }
}
```

Ilustración 5.99. Manejo de la Transición

3. Escena de Transición

En la escena de transición, el script 'Transición' activa o desactiva paneles basado en el estado de juego. Si es 'Game Over', se muestra el panel correspondiente, y si el juego continúa, se muestra el panel con información del del próximo nivel.



Ilustración 5.100. Escena de Transición

4. Carga del Nuevo Nivel

Una vez que pasa 5 segundos en la escena de Transición, el 'GameManager' configura las variables para el nuevo mundo y nivel, basándose en la información almacenada en las estructuras de 'Mundo' y 'Nivel'.

Finalmente, se carga el nuevo nivel utilizando 'SceneManager' para traer la escena correspondiente al mundo y nivel seleccionados.

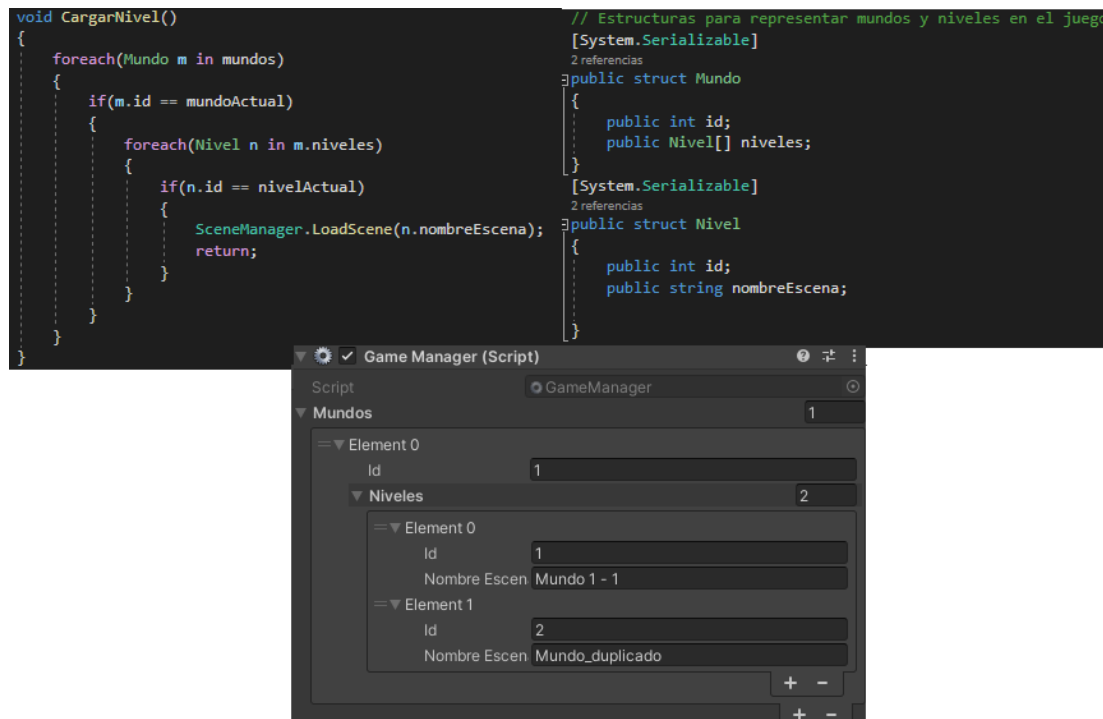


Ilustración 5.101. Carga del nuevo nivel

5.3.6 Configuración de la Cámara y Ajustes Visuales

En el desarrollo del videojuego, la gestión, implementación y configuración de la cámara es una parte crucial para mantener la jugabilidad y la estética fiel al juego original, mientras se adapta a las nuevas resoluciones.

A continuación, detallo los pasos y consideraciones específicas sobre la configuración de la cámara:

1. Configuración de la Proporción de Aspecto de la Cámara

Para mantener la proporción de aspecto visual del juego original, he desarrollado el script '**AspectRatioCamara**' que se encarga de:

- Calcular la proporción de aspecto del juego original, 16:15.
- Ajustar el tamaño y posición del viewport de la cámara para asegurar que el juego se muestre correctamente en cualquier resolución de pantalla, utilizando una comparación entre la proporción de aspecto objetivo y la actual de la pantalla.
- Configurar el 'Viewport Rect' de la cámara para ajustar su anchura y centrarlo, garantizando que la visualización no se estire ni comprima.

```
public class AspectRatioCamara : MonoBehaviour
{
    public Vector2 aspectObjetivo = new Vector2(16f, 15f);

    // Mensaje de Unity | 0 referencias
    private void Awake()
    {
        float targetAspect = aspectObjetivo.x / aspectObjetivo.y; // Se calcula la relación de aspecto objetivo
        float screenAspect = (float)Screen.width / (float)Screen.height; // Se obtiene la relación de aspecto actual de la pantalla

        float escalaAltura = screenAspect / targetAspect; // altura necesaria para ajustar la cámara
        float escalaAncho = 1f / escalaAltura; // ancho basado en la escala de altura

        Camera camara = GetComponent<Camera>();

        Rect rect = camara.rect; // rectángulo actual de la cámara
        rect.width = escalaAncho; // Se ajusta el ancho del rectángulo de la cámara
        rect.height = 1.0f; // Se establece la altura del rectángulo de la cámara al máximo posible
        rect.x = (1.0f - escalaAncho) / 2.0f; // Se centra la cámara en la pantalla ajustando el eje X
        rect.y = 0f; // Se establece el eje Y del rectángulo de la cámara

        camara.rect = rect; // Se aplican los cambios al rectángulo de la cámara
    }
}
```

Ilustración 5.102. AspectRatioCamara.cs

2. Seguimiento Dinámico de Mario

Para hacer que la cámara siga a Mario he desarrollado el script 'SeguimientoCamara' que se encarga de:

- Definir un offset para que la cámara se sitúe ligeramente delante de Mario, mejorando la visibilidad del área hacia la que se dirige.
- Establecer límites horizontales (mínimos y máximos) para el movimiento de la cámara, basados en posiciones específicas en el nivel (usando GameObjects 'limitelzq' y 'limiteDer'), que previenen que la cámara muestre áreas fuera de los límites del nivel.
- Ajustar dinámicamente la posición de la cámara en el eje X para que nunca retroceda, manteniendo la última posición máxima alcanzada como límite inferior para la cámara. Esto evita que el jugador pueda hacer que Mario vuelva hacia áreas previamente exploradas, lo cual es fiel a la mecánica original del juego.

```
public class SeguimientoCamara : MonoBehaviour
{
    public Transform objetivoSeguir; //Mario
    public float unidadesPorDelanteObjetivo = 2.5f; //Distancia por delante del objetivo (Mario)
    public float minPosicionX; // Posición mínima X que la cámara puede alcanzar
    public float maxPosicionX; // Posición máxima X que la cámara puede alcanzar

    public Transform limiteIzq; // Límite izquierdo del nivel
    public Transform limiteDer; // Límite derecho del nivel

    float anchoCamara; // Almacena la anchura de la cámara
    float ultimaPosicion; // Última posición X guardada de la cámara

    public Transform colliderIzq;
    public Transform colliderDer;

    // Start is called before the first frame update
    [SerializeField]
    void Start()
    {
        // Se calcula la anchura de la cámara basada en el tamaño ortográfico y el aspecto
        anchoCamara = Camera.main.orthographicSize * Camera.main.aspect;
        minPosicionX = limiteIzq.position.x + anchoCamara;
        maxPosicionX = limiteDer.position.x - anchoCamara;

        ultimaPosicion = minPosicionX;

        // Se donfigura la posición inicial de los colliders que limitan el movimiento de la cámara
        colliderIzq.position = new Vector2(transform.position.x - anchoCamara - 0.5f, colliderIzq.position.y);
        colliderDer.position = new Vector2(transform.position.x + anchoCamara + 0.5f, colliderDer.position.y);
    }

    // Update is called once per frame
    [SerializeField]
    void Update()
    {
        // nueva posición X de la cámara basándose en la posición del objetivo y la distancia definida por delante del mismo
        float nuevaPosicionX = objetivoSeguir.position.x + unidadesPorDelanteObjetivo;
        //nueva posición X dentro de los márgenes establecidos
        nuevaPosicionX = Mathf.Clamp(nuevaPosicionX, ultimaPosicion, maxPosicionX);

        // Se actualiza la posición de la cámara
        transform.position = new Vector3(nuevaPosicionX, transform.position.y, transform.position.z);
        // Guarda la última posición X actualizada
        ultimaPosicion = nuevaPosicionX;
    }
}
```

Ilustración 5.103. SeguimientoCamara.cs

Además, he colocado colliders en los bordes de la cámara para evitar que Mario salga de la vista, asegurando que siempre esté visible en pantalla.

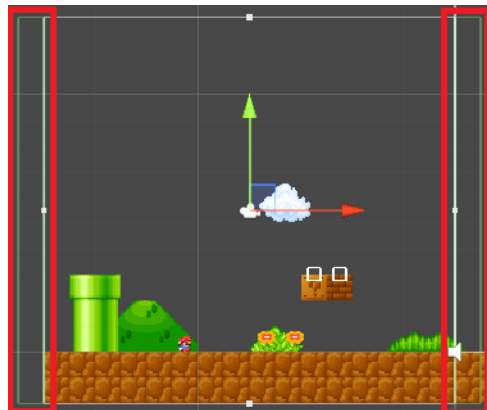


Ilustración 5.104. Collider – Configuración Cámara

3. Adaptación de la Cámara a Diferentes Resoluciones

Utilizando el ajuste dinámico en el script 'AspectRatioCamara' se garantiza que la experiencia visual se mantenga consistente en diferentes dispositivos.

FUERA DE CÁMARA

En el desarrollo del juego, una de las optimizaciones que he implementado ha sido la **gestión de los elementos fuera de cámara** para asegurar el rendimiento óptimo y la experiencia de juego fluida.

Para evitar el consumo innecesario de recursos, se ha implementado un sistema para eliminar automáticamente enemigos, ítems y objetos que salen de la visibilidad de la cámara. Esto lo he llevado a cabo mediante el script '**EliminarFueraCamara**', una vez que un objeto deja de ser visible y supera cierta distancia del centro de la cámara, determinada por '*minDistanciaEliminado*', se elimina del juego. Esto es crucial para mantener el rendimiento.

La **eliminación de enemigos en el juego** se maneja de diferente forma según el tipo de enemigo y su estado específico:

- **Koopa, Koopa Rojo y Koopa con Alas**

Cuando Koopa es golpeado y se refugia dentro de su caparazón, el enfoque de eliminación cambia. Si el Koopa escondido no está en movimiento (estático dentro del caparazón), puede eliminarse si está fuera del alcance visible y además ha superado la '*minDistanciaEliminado*' establecida.

Sin embargo, si el Koopa está activo y moviéndose dentro del caparazón (lanzado en modo caparazón), puede eliminarse también si esta por delante de la cámara.

```
EliminarFueraCamara eliminarFueraCamara = GetComponent<EliminarFueraCamara>();
if (lanzadoEnCaparazon) //Si Koopa ha sido lanzado y está en modo caparazon
{
    eliminarFueraCamara.destruirSoloPorDetras = false; //Puede destruirse también por delante de la cámara
}
else
{
    eliminarFueraCamara.destruirSoloPorDetras = true; //Solo puede destruirse si está por detrás de la cámara
}
```

Ilustración 5.105. Eliminación de Koopa de la Escena

- **Goomba**

Los Goombas se eliminan directamente cuando salen de la pantalla por detrás y superan la distancia mínima configurada.

- **Planta Piraña**

La eliminación de las Plantas Piraña se maneja eliminando el objeto padre.

```
void Update()
{
    if (spriteRenderer.isVisible)
    {
        esVisible = true; // Marca el objeto como visible si está dentro del campo de visión de la cámara
    }
    else
    {
        if (esVisible) //Verifica si el objeto ha sido visible anteriormente
        {
            // Comprueba si el objeto está suficientemente lejos de la cámara para ser eliminado
            if (Mathf.Abs(transform.position.x - Camera.main.transform.position.x) > minDistanciaEliminado)
            {
                // Si sólo debe ser destruido por detrás y se encuentra delante de la cámara, no se destruye
                if (destruirSoloPorDetras)
                {
                    if (transform.position.x > Camera.main.transform.position.x)
                    {
                        return;
                    }
                }
                // Destruye el objeto o su objeto padre dependiendo de la configuración (esto es para la planta Piraña, ya que se tiene que eliminar el objeto padre)
                if (parent == null)
                {
                    Destroy(gameObject);
                }
                else
                {
                    Destroy(parent);
                }
            }
        }
    }
}
```

Ilustración 5.106. Eliminación de Enemigos e items de la Escena

Además, los enemigos se activan no solo cuando entran en el campo visual de la cámara sino también en respuesta a la proximidad de otros enemigos activos.

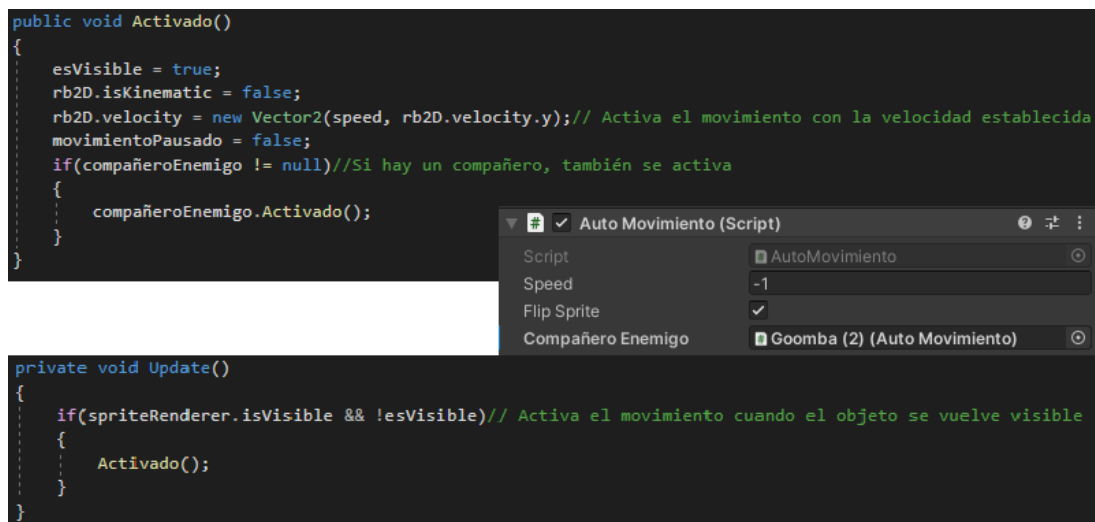


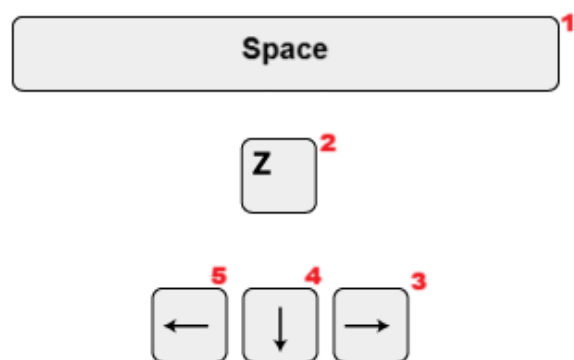
Ilustración 5.107. Activación de Enemigos en la Escena

6 APÉNDICES

6.1 Guía original del Trabajo Fin de Título

CONTENIDO: incluir la propuesta o **guía original del TFG/TFM** (publicada en la web de la EPS en el momento de la convocatoria), así como el histórico de modificaciones que haya podido tener dicha propuesta (título, objetivos, etc.).

6.2 Manuales de usuario



1. Saltar
2. Lanzar bolas de fuego (disponible cuando Mario tiene el poder de la Flor de Fuego), Su Mario está en estado Super y se encuentra sobre un bloque destructible, al presionar esta tecla mientras está agachado romperá el bloque.
3. Mover a Mario hacia la derecha
4. Agacharse.
5. Mover a Mario hacia la izquierda

7 BIBLIOGRAFÍA

- [1] Unity Technologies, 2024: <https://unity.com/es>
- [2] Microsoft, 2024. Visual Studio: <https://visualstudio.microsoft.com/>
- [3] Microsoft, 2024. Paint: <https://www.microsoft.com/es-es/windows/paint>
- [4] GIMP, 2024. GIMP: <https://www.gimp.org/>
- [5] Schwaber, K. y Sutherland, J., 2020. The Scrum Guide: <https://scrumguides.org/scrum-guide.html>
- [6] Unity Technologies, 2024. Unity - Manual: Sprites: <https://docs.unity3d.com/Manual/Sprites.html>
- [7] Unity Technologies, 2024. Unity - Manual: Physics 2D Reference: <https://docs.unity3d.com/Manual/Physics2DReference.html>
- [8] Unity Technologies, 2024. Unity - Manual: Physics 2D Raycaster: [Unity - Manual: Physics 2D Raycaster \(unity3d.com\)](https://docs.unity3d.com/Manual/Physics2DRaycaster.html)
- [9] Unity Technologies, 2024. Unity - Manual: Animation system overview: [Unity - Manual: Animation system overview \(unity3d.com\)](https://docs.unity3d.com/Manual/AnimationSystemOverview.html)
- [10] Unity Technologies, 2024. Unity - Manual: Animation Clips: [Unity - Manual: Animation Clips \(unity3d.com\)](https://docs.unity3d.com/Manual/AnimationClips.html)
- [11] Unity Technologies, 2024. Unity - Manual: Animator Controller: [Unity - Manual: Animator Controller \(unity3d.com\)](https://docs.unity3d.com/Manual/AnimatorController.html)
- [12] Unity Technologies, 2024. Unity - Manual: Prefabs: [Unity - Manual: Prefabs \(unity3d.com\)](https://docs.unity3d.com/Manual/Prefabs.html)
- [13] Patrón de Diseño Singleton: <https://aspnetcoremaster.com/csharp/patron-de-dise%C3%B1o-singleton-csharp.html>
- [14] Unity Technologies, 2024. Unity - Manual: Audio Clip: [Unity - Manual: Audio Clip \(unity3d.com\)](https://docs.unity3d.com/Manual/AudioClip.html)
- [15] Unity Technologies, 2024. Unity - Manual: Audio Source: [Unity - Manual: Audio Source \(unity3d.com\)](https://docs.unity3d.com/Manual/AudioSource.html)
- [16] Unity Technologies, 2024. Unity - Manual: Canvas: [Unity - Manual: Canvas \(unity3d.com\)](https://docs.unity3d.com/Manual/Canvas.html)
- [17] Unity Technologies, 2024. Unity - Manual: Event System: [Unity - Manual: Event System \(unity3d.com\)](https://docs.unity3d.com/Manual/EventSystem.html)

[18] Unity Technologies, 2024. Unity - Manual: Canvas Components: [Unity - Manual: Canvas \(unity3d.com\)](#)

[19] Unity Technologies, 2024. Unity - Manual: Canvas Scaler: [Unity - Manual: Canvas Scaler \(unity3d.com\)](#)

[20] Unity Technologies, 2024. Unity - Manual: Visual Components: [Unity - Manual: Visual Components \(unity3d.com\)](#)

[21] Unity Technologies, 2024. Unity - Manual: Tilemap: [Unity - Manual: Tilemaps \(unity3d.com\)](#)

Audios del juego: [TMK | Downloads | Sounds & Music | Sound Clips | Super Mario Bros. \(NES\) \(themushroomkingdom.net\)](#)

Sprites del juego: [NES - Super Mario Bros. - The Spriters Resource \(spriters-resource.com\)](#)

Libros:

Mastering Unity 2D Game Development - Second Edition por Simon Jackson

Learning C# by Developing Games with Unity 2020: An enjoyable and intuitive approach to getting started with C# programming and Unity por Harrison Ferrone

Unity 2018 Game Development in 24 Hours, Sams Teach Yourself (3rd Edition) por Ben Tristem y Mike Geig