



Departamento de Ciencia de Datos e Informática

Facultad de Ingeniería

Universidad de Playa Ancha

Star Forest: Video juego de Plataforma 2D

Italo Arturo Lama Guerra

**Proyecto de Título para obtener el Título de Ingeniería
Informática**

Profesor Guía: Dr. Mario Andrés Bruno Meléndez

Valparaíso, Chile

2023

Tabla de contenidos

Contenido

Tabla de contenidos	2
Índice de Ilustraciones	6
Agradecimientos	13
Resumen.....	14
Abstract	15
Introducción	16
Capítulo 1: Descripción General.....	17
1.1 ¿Qué es un videojuego?	17
1.2 Objetivos.....	19
1.2.1 Objetivo General	19
1.2.2 Objetivos específicos.....	19
1.3 Limitaciones.....	20
Capítulo 2: Los Juegos de Plataforma	21
2.1 Videojuegos de plataforma 2D	21
2.1.1 El Entorno 2D	21
2.1.2 El primer videojuego de plataforma 2D.....	24
2.1.3 ¿Por qué saltamos?	25
2.1.4 El salto en los videojuegos y su funcionamiento	26
2.1.5 El desafío de un salto	29
2.1.6 Evolución de los Videojuegos de Plataforma 2D	31
2.2 Mecánicas y elementos de un videojuego de Plataformas.....	33
2.2.1 Variaciones del salto o diferentes tipos.....	33
2.2.2 Doble Salto	34
2.2.3 Coyote Time	36

2.2.4	Jump Input Buffering.....	37
2.2.5	Wall Jump.....	39
2.2.6	Hitbox en los videojuegos	42
2.2.7	Sistema de Salud	43
2.2.8	Puntos de control.....	44
2.2.9	Game Flow y sus elementos en un videojuego.....	45
2.2.10	Diseño de Nivel	48
2.2.11	Sistema de Invulnerabilidad.....	53
2.3	Experiencia de Usuario (UX).....	55
2.3.1	Señales y Feedbacks.....	55
2.3.2	Beneficios del UX en los videojuegos.....	58
2.3.3	Onboarding y Usability.....	58
2.3.4	Jugabilidad Intuitiva	60
2.3.5	Diseño de Niveles	60
2.3.6	Estilo Visual y Audio	61
2.3.7	Evaluación de la Experiencia de Usuario (UX) en Videojuegos de Plataforma 2D	62
2.3.9	Herramientas para el Desarrollo de Videojuegos de Plataforma 2D en Unity	63
2.3.10	Casos de Estudio Relevantes.....	64
2.3.11	Conclusiones	65
Capítulo 3: Metodología		67
3.1	Fases de la Metodología.....	68
3.2	Evaluación UX	74
Capítulo 4: Desarrollo Star Forest en Unity		76
4.1	Introducción a Unity.....	76
4.2	Pantalla Principal de Unity	77

4.3	Los objetos de juego (gameObject).....	82
4.4	Tilemap Editor (Editor de mapas de mosaico)	83
4.5	El jugador.....	88
4.5.1	Configuración básica	88
4.5.2	Movimiento básico.....	90
4.5.3	El Salto.....	92
4.5.3	Evitando saltos infinitos	93
4.5.4	Volteando al jugador.....	94
4.5.5	Evitando quedarse atascado en las murallas.....	95
4.5.6	Animando al personaje	96
4.5.7	Cambiando de animaciones.....	97
4.6	Mecánicas de salto y movimiento.....	99
4.6.1	Doble Salto	99
4.6.2	Coyote Time y Jump Buffering.....	100
4.6.2	Dash.....	101
4.7	Cámara.....	102
4.8	Obstáculos, Daño y Sistema de Salud	108
4.8.1	Creando el sistema de salud	109
4.8.2	Dañando al Jugador.....	111
4.8.3	Interfaz del Sistema de Salud.....	111
4.8.4	Actualizando el Sistema de Salud	114
4.8.5	Añadiendo invulnerabilidad al jugador	116
4.8.6	Retroalimentación visual del sistema de invulnerabilidad	118
4.9	Puntos de Control y Reparación.....	121
4.9.1	Añadiendo un punto de control.....	121
4.9.2	Desactivando puntos de control	123

4.9.3	Almacenando el punto de reaparición.....	124
4.9.4	Sistema de reaparición.....	125
4.10	Coleccionables y Efectos visuales	126
4.10.1	Añadiendo coleccionables	127
4.10.2	Coleccionable de Salud	129
4.10.3	Interfaz para las cerezas.....	129
4.11	Creando al enemigo	132
4.12	Añadiendo Sonidos	136
4.13	Menú.....	138
4.14	Finalizando el videojuego.....	142
Capítulo 5: Resultados		144
5.1	Resultados de la Evaluación	144
Capítulo 6: Conclusión.....		151
Referencias.....		152

Índice de Ilustraciones

Ilustración 1: Espacio bi-dimensional.	22
Ilustración 2: Proyección Ortográfica	23
Ilustración 3: Cuphead en Acción.....	24
Ilustración 4: Space Panic.....	24
Ilustración 5: Donkey Kong Arcade.....	25
Ilustración 6: Super Mario Bros. para la consola NES	27
Ilustración 7: Salto de 29 Frames.....	28
Ilustración 8: Mario cambiando de dirección mientras está en el aire	29
Ilustración 9: Obstáculos en Super Mario Bros	29
Ilustración 10: Zona de Salto.....	30
Ilustración 11: Super Mario Bros. en Acción 2	32
Ilustración 12: Sonic The Hedgehog para la consola Sega Genesis	32
Ilustración 13: Double Dragon para la Arcade en Acción	34
Ilustración 14: Doble Salto	35
Ilustración 15: Ejemplo de Coyote Time	36
Ilustración 16: Ejemplo de Jump Input Buffering.....	37
Ilustración 17: Explicación de Jump Input Buffering.....	38
Ilustración 18: Ejemplo de Wall Jump	39
Ilustración 19: Wall Jump en Acción	40
Ilustración 20: Wall Jump en Detalle	40
Ilustración 21: Hitbox en Super Mario Bros.	42
Ilustración 22: Hitbox obstáculos	43
Ilustración 23: Ejemplo de Sistema de Salud.....	44
Ilustración 24: Ejemplo de Checkpoints o Puntos de Control.....	45
Ilustración 25: Recompensas en Super Mario Bros.	46
Ilustración 26: Feedback en Dark Souls	47
Ilustración 27: Enemigo en Super Mario Bros.	48
Ilustración 28: Obstáculos en Super Meat Boy Forever.....	49
Ilustración 29: Obstáculos en Sonic The Hedgehog	50

Ilustración 30: Argollas en Sonic The Hedgehog	51
Ilustración 31: Timothy and the Tower of Mu	52
Ilustración 32: Señales en Star Wars Jedi: Fallen Order	56
Ilustración 33: Feedback en Call of Duty Mobile.....	56
Ilustración 34: Phoenix Wright Ace Attorney Trilogy en Acción	57
Ilustración 35: Tutorial en Kirby Star Allies	59
Ilustración 36: Jugabilidad intuitiva en Donkey Kong Country.....	60
Ilustración 37: Detalles de Nivel en Castlevania Symphony of the Night	61
Ilustración 38: Estilo Visual en Pizza Tower.....	61
Ilustración 39: Vista general de Unity.....	64
Ilustración 40: Hollow Knight en Acción	64
Ilustración 41: Celeste en Acción	65
Ilustración 42: Diagrama de metodología iterativa incremental	68
Ilustración 43: Nuevo proyecto en Unity.....	77
Ilustración 44: Ventana de Proyecto en Unity	77
Ilustración 45: Ventana de Jerarquía en Unity	78
Ilustración 46: Ventana de Juego en Unity	79
Ilustración 47: Ventana de Inspector en Unity	80
Ilustración 48: Game Objects en Unity	82
Ilustración 49: Tilemap Editor en Unity	83
Ilustración 50: Creando un Game Object de tipo Tilemap.....	84
Ilustración 51: Seleccionando Tiles en Unity	84
Ilustración 52: Pintando en la escena	85
Ilustración 53: Componente Rigidbody 2D	86
Ilustración 54: Componente Tilemap Collider	87
Ilustración 55: Composite Collider 2D en Acción	87
Ilustración 56: Añadiendo al Personaje Jugable en la ventana de Escena	88
Ilustración 57: Configurando el Rigidbody 2D del Personaje	89
Ilustración 58: Caja de Colisiones del Personaje 1	89
Ilustración 59: Caja de Colisiones del Personaje 2	89
Ilustración 60: Configurando la Caja de Colisiones	90
Ilustración 61: Código inicial del script Player Controller	91

Ilustración 62: Código de movimiento del Jugador	91
Ilustración 63: Código del script Better Jump	92
Ilustración 64: Verificaciones para evitar el salto infinito	93
Ilustración 65: Creando el Game Object "Ground Check"	93
Ilustración 66: Función Flip en el script Player Controller	94
Ilustración 67: Llamando la función en el método Update	94
Ilustración 68: Creando el material No Friction	95
Ilustración 69: Asignando valores para el material No Friction	95
Ilustración 70: Asignando el material en el Rigidbody 2D del Jugador	95
Ilustración 71: Creando distintas animaciones para el Jugador	96
Ilustración 72: Añadiendo sprites o avatares para la animación "Idle"	96
Ilustración 73: Configurando la animación "Idle"	97
Ilustración 74: Creando condiciones en la ventana Animator	97
Ilustración 75: Creando transiciones entre animaciones	98
Ilustración 76: Cambiando animaciones en el script "Player Controller"	98
Ilustración 77: Código de mecánica Doble Salto	99
Ilustración 78: Doble Salto en Acción	99
Ilustración 79: Implementación de Jump Buffering y Coyote Time en el script "Player Controller"	100
Ilustración 80: Coyote Time en Acción	101
Ilustración 81: Corrutina Dash en el script Player Controller	101
Ilustración 82: Verificaciones y llamada de la corrutina Dash en el método Update	102
Ilustración 83: Mecánica Dash en Acción	102
Ilustración 84: Añadiendo Cinemachine al proyecto	103
Ilustración 85: Creando un Game Object de tipo Cinemachine	104
Ilustración 86: Componente Cinemachine en la ventana de Inspector	105
Ilustración 87: Cinemachine sin límites aplicados	105
Ilustración 88: Componente Polygon Collider 2D en el Game Object "Confiner"	106
Ilustración 89: Configurando los límites para la cámara	106
Ilustración 90: Añadiendo el objeto "Confiner" para los límites de la cámara	107
Ilustración 91: Límites aplicados a la cámara	107
Ilustración 92: Añadiendo un obstáculo al videojuego	108

Ilustración 93: Ajustando el tamaño de la caja de colisiones del obstáculo	108
Ilustración 94: Configurando la caja de Colisiones en el inspector	109
Ilustración 95: Función "Deal Damage" en el script Player Health Controller	109
Ilustración 96: Asignando etiquetas al Jugador	110
Ilustración 97: Instanciando el script Player Health Controller	111
Ilustración 98: Función para dañar al jugador y restarle puntos de salud	111
Ilustración 99: Añadiendo un game object de tipo Interfaz	112
Ilustración 100: Interfaz de tipo imagen en Unity.....	112
Ilustración 101: Cambinado la imagen del cuadrado a la de un corazón.....	113
Ilustración 102: Posicionando los corazones en el juego.....	113
Ilustración 103: "UnityEngine.UI" y variables en el script UI Controller	114
Ilustración 104: Asignando corazón lleno en el inspector	115
Ilustración 105: Estructura switch en el script UI Controller	116
Ilustración 106: Actualizando la interfaz en el script Player Health Controller	116
Ilustración 107: Variables para el sistema de invulnerabilidad en el script Player Health Controller	117
Ilustración 108: Añadiendo verificaciones con las variables del sistema de invulnerabilidad.....	118
Ilustración 109: Verificando si el jugador se encuentra en estado de invulnerabilidad	118
Ilustración 110: Representación visual del sistema de invulnerabilidad	119
Ilustración 111: Cambiando la opacidad del personaje en el script Player Health Controller	119
Ilustración 112: Restaurando la opacidad del jugador en el método Update	120
Ilustración 113: Añadiendo punto de control en el proyecto.....	121
Ilustración 114: Variables del script Checkpoint	121
Ilustración 115: Activando el punto de control cuando colisiona con el jugador	122
Ilustración 116: Configurando las imágenes del punto de control en el inspector ...	122
Ilustración 117: Puntos de control activados simultáneamente	122
Ilustración 118: Función "ResetCheckpoint" en el script UI Controller	123
Ilustración 119: Código base del script "Checkpoint Controller"	123

Ilustración 120: Recorriendo todos los puntos de control presentes para desactivarlos	123
Ilustración 121: Llamando la función para desactivar puntos de control	124
Ilustración 122: Puntos de control funcionales	124
Ilustración 123: Función para almacenar la posición del jugador	124
Ilustración 124: Almacenando la posición del jugador cuando colisiona con un punto de control	125
Ilustración 125: Código base del script Level Manager	125
Ilustración 126: Haciendo reaparecer al jugador cuando su punto de salud llega a cero	126
Ilustración 127: Estableciendo punto de reaparición inicial	126
Ilustración 128: Añadiendo objeto coleccionable en el proyecto	127
Ilustración 129: Variables en el script Pickup	128
Ilustración 130: Variable de recolección en el script Level Manager	128
Ilustración 131: Código de recolectando cereza	128
Ilustración 132: Código de recolectando corazón	129
Ilustración 133: Función "Heal Player" en el script Player Health Controller	129
Ilustración 134: Añadiendo texto en el proyecto	130
Ilustración 135: Componente "Outline" para el texto	130
Ilustración 136: Texto con sombra aplicada	130
Ilustración 137: Función "Update Cherry Count" en el script "UI Controller"	130
Ilustración 138: Actualizando el texto con la cantidad de cerezas coleccionadas	131
Ilustración 139: Representación visual de cantidad de cerezas coleccionadas en Acción	131
Ilustración 140: Estableciendo puntos de patrullaje del Enemigo	132
Ilustración 141: Código base de movimiento del Enemigo	132
Ilustración 142: Enemigo en movimiento	133
Ilustración 143: Añadiendo el objeto "Stompbox" en el jugador	133
Ilustración 144: Asignando etiqueta al Game Object "StompBox"	134
Ilustración 145: Asignando etiqueta al Enemigo	134
Ilustración 146: Stompbox en Acción	135
Ilustración 147: Sonidos añadidos en el proyecto	136

Ilustración 148: Código base del script Audio Manager	136
Ilustración 149: Función para reproducir efecto de sonido	137
Ilustración 150: Almacenando sonidos en un Array o lista	137
Ilustración 151: Reproduciendo el sonido en la función "Deal Damage" del script Player Health Controller.....	137
Ilustración 152: Añadiendo Menú de inicio y botones interactivables	138
Ilustración 153: Código base del script Main Menu	139
Ilustración 154: Asignando la función de Iniciar juego al botón de Iniciar	139
Ilustración 155: Asignando la función de salir del juego al botón de Salir	140
Ilustración 156: Añadiendo menú de pausa	140
Ilustración 157: Código base del script Pause	141
Ilustración 158: Build Settings en Unity	142
Ilustración 159: Ventana "Build Settings" para crear el juego	142
Ilustración 160: ¿La respuesta de los controles es precisa y se siente fluida?.....	144
Ilustración 161: ¿La respuesta del salto se siente fluida y cómoda?.....	144
Ilustración 162: ¿La gravedad del salto se siente cómoda?.....	145
Ilustración 163: ¿La velocidad del personaje se siente correcta?	145
Ilustración 164: ¿La respuesta del salto doble se siente fluida?	146
Ilustración 165: ¿La respuesta del salto coyote se siente correcta?	146
Ilustración 166: ¿La respuesta del impulso o dash se siente correcta?	146
Ilustración 167: ¿La fuerza del salto doble es correcta o necesita más altura?.....	147
Ilustración 168: ¿El juego tiene una interfaz clara y fácil de entender?	148
Ilustración 169: ¿Los elementos visuales del juego son atractivos y coherentes con el estilo artístico?	148
Ilustración 170: ¿El tutorial o instrucciones del juego son claros y te ayudan a comprender las mecánicas?	148
Ilustración 171: ¿El sistema de guardado o checkpoints es justo y te permite avanzar sin frustración excesiva?	149
Ilustración 172: ¿El juego ofrece una cantidad correcta en términos de enemigos y obstáculos en los niveles?	149
Ilustración 173: ¿La música y los efectos de sonido contribuyen a la atmósfera del juego?.....	149

Ilustración 174: ¿La música y los efectos de sonido tienen un volumen adecuado? 150

Agradecimientos

Un agradecimiento sincero se dirige al inmenso apoyo brindado por mi profesor guía, el Dr. Mario Bruno, cuya orientación y asistencia fueron fundamentales en la elaboración de esta tesis. Su guía no solo contribuyó al contenido de este proyecto, sino que también enriqueció mi comprensión sobre los videojuegos en general.

Expreso mi profundo agradecimiento a mi familia, mis tíos, mis primas, cuyo apoyo constante y paciencia infinita fueron elementos cruciales para que este proyecto viera la luz. Su respaldo inquebrantable me impulsa a esforzarme por ser una mejor persona cada día.

Un reconocimiento especial se dirige a mi primo Rodrigo Carmona, quien generosamente proporcionó recursos artísticos para el videojuego desarrollado en esta tesis. Su colaboración fue de gran ayuda, y le estoy eternamente agradecido por ello.

Quiero expresar un agradecimiento especial a mi mejor amiga, Priscila Tapia, quien no solo brindó consejos valiosos para la redacción de este proyecto, sino que también contribuyó con diseños artísticos para el videojuego asociado a esta tesis. A pesar de la distancia, nuestra amistad ha perdurado a lo largo de los años y es algo que valoro profundamente.

Italo Arturo Lama Guerra

Resumen

Este proyecto de título tiene como objetivo desarrollar el videojuego Star Forest, un juego de plataformas 2D en Unity. La tesis busca introducir el proceso de desarrollo del juego, incluyendo el uso del motor gráfico de videojuegos Unity y la programación en C# a través de Visual Studio. Se presentan las funciones y componentes que Unity ofrece para facilitar el proceso de desarrollo, y se utiliza una metodología de fases para gestionar el proceso de desarrollo.

Abstract

This thesis is crafted with the primary objective of acquainting fellow developers and programmers with game development using Unity, presupposing a foundational understanding of C# programming. Within the scope of this exploration, an in-depth introduction to Unity is provided, a preeminent game engine in contemporary times. The narrative unfolds by guiding readers through the step-by-step creation of a 2D platformer game titled "Star Forest."

In culmination, the thesis not only imparts knowledge but also instills practical skills.

Introducción

Este documento presenta el desarrollo del videojuego 2D de plataforma para computadoras llamado *Star Forest*.

- En el capítulo 1, se presenta una descripción general de lo que implica un videojuego y se definen los objetivos generales y específicos del proyecto, así como las limitaciones del mismo.
- En el capítulo 2, se estudian y analizan los videojuegos de plataforma 2D, el origen del primer salto, las mecánicas, el flujo de juego, el diseño de niveles y la experiencia de usuario (UX), sentando las bases para comprender cómo los videojuegos se han convertido en una forma de entretenimiento de gran influencia, y cómo la UX desempeña un papel crucial en la creación de experiencias de juego excepcionales.
- En el capítulo 3, se describen las fases de metodología para el desarrollo de *Star Forest* y cómo se lleva a cabo su evaluación por medio de terceros.
- En el capítulo 4, se describe el motor gráfico Unity, que se utiliza en el desarrollo del videojuego 2D *Star Forest*. Se ofrece una visión general del programa, destacando las herramientas simples pero esenciales que se emplean a lo largo del desarrollo. Se hace uso de la investigación previa realizada en el capítulo 2 para implementar las mecánicas del juego. Se detalla la implementación y programación de cada aspecto del videojuego, incluyendo el diseño del videojuego 2D, la gestión y uso de códigos para diversos elementos del juego, y la implementación de animaciones para personajes, objetos y enemigos, todo esto llevado a cabo dentro del software Unity.
- El capítulo 5 se enfoca en la validación del videojuego a través de pruebas realizadas por terceros.
- El capítulo 6 presenta las conclusiones pertinentes del proyecto.

Capítulo 1: Descripción General

1.1 ¿Qué es un videojuego?

Hay diversas definiciones de un videojuego. Una de ellas es:

Los videojuegos son juegos electrónicos en los que están involucradas una o más personas interactuando entre sí, es decir, es todo tipo de juego digital interactivo que su base principal es entretener y divertir por un lapso de tiempo prolongado, utilizando soportes de interface como los ordenadores, las videoconsolas, las consolas portátiles o máquinas recreativas. El programa informático del juego permite ser participativo de una experiencia que en la realidad no se puede hacer como los juegos de rol. [1].

Las personas juegan juegos para relajarse, competir con otros jugadores o simplemente para pasar el tiempo. Como dijo una vez el presidente de Nintendo, Satoru Iwata (1959-2015): “Los videojuegos deben ser solo una cosa. Diversión. Diversión para todos” [2]

Al pasar los años, los videojuegos se volvieron muy populares y la industria creció enormemente [3]. Cada día, nuevos videojuegos son presentados a miles de personas, ya sea por empresas grandes internacionales, compañías de videojuegos indie¹ o videojuegos hechos por una sola persona. Las noticias sobre los lanzamientos de videojuegos nuevos aparecen día a día en redes sociales, ya sea por páginas como GameSpot [5], IGN [6], y Tarreo.cl [7], entre muchos otros.

Al igual que la misma televisión, un correcto uso de los videojuegos puede tener impactos positivos sobre los niños/as o adolescentes. Un claro ejemplo vendría siendo los videojuegos educativos, con contenidos variados; pueden enseñar desde matemáticas, enseñar lenguajes, enseñar a leer, enseñar distintos idiomas, entre muchos otros contenidos.

Los programadores y desarrolladores de software requieren habilidades adicionales para el desarrollo y diseño de videojuegos: animación, diseño gráfico, artes

¹ Derivado de “independiente”. En éste contexto, y siguiendo la definición de wikipedia, son videojuegos desarrollados por un grupo reducido de individuo que no suelen contar con el apoyo monetario de distribuidoras y publicistas, por lo que su difusión depende principalmente de la voluntad de los jugadores para compartirlas con otros [4], es decir, los jugadores son sus mismos distribuidores.

conceptuales, diseño de niveles, etc. Los videojuegos AAA tienen la nota más alta en presupuesto, materias primas y el tiempo invertido (triple-A), y son desarrollados por empresas grandes. Pero eso no significa que un solo individuo o un equipo pequeño también pueden hacer un buen videojuego por si mismo. Las posibilidades ofrecidas en la internet, libros u otros lugares de información permiten el autoaprendizaje de cómo crear videojuegos pero aún así, existe una gran diferencia entre el conocimiento que uno puede alcanzar y las habilidades requeridas en un estudio, para abordar las dificultades y las técnicas necesarias en la industria del desarrollo de videojuegos.

En este proyecto de tesis, el objetivo principal es investigar el desarrollo de juegos a través del proceso de creación de un juego de plataformas 2D llamado *Star Forest*. Al abordar conceptos clave de un juego de plataformas 2D, como mecánicas de juego, hitbox, saltos, diseño de niveles, eje de coordenadas y uso de la cámara, el enfoque es proporcionar una introducción integral al desarrollo de juegos utilizando el lenguaje de programación C# y el motor de juegos Unity.

Se integra el concepto de Experiencia de Usuario (UX) desde el inicio del proceso para validar el videojuego y llevar a cabo pruebas de aceptación. Cada uno de estos componentes se presenta de manera detallada, respaldados con ejemplos para facilitar la comprensión de los lectores. Además, se explica la importancia práctica de comprender y aplicar estos conceptos en la creación de un juego de plataformas 2D.

Este enfoque no solo busca la creación de un juego específico, sino que también proporciona a los programadores y desarrolladores una comprensión fundamental del desarrollo de juegos, permitiéndoles iniciar sus propios proyectos de forma independiente al final de la tesis. Utilizando las herramientas que ofrece Unity, uno de los motores de juegos más ampliamente utilizados, se establece la base para construir el videojuego *Star Forest* y demostrar la aplicabilidad práctica de los conceptos discutidos.

1.2 Objetivos

1.2.1 Objetivo General

- Diseñar el videojuego 2D de plataforma *Star Forest* usando el motor de juegos Unity.

1.2.2 Objetivos específicos

- Estudiar conceptos clave de un videojuego de plataformas 2D.
- Diseñar el videojuego 2D de plataforma *Star Forest*
- Validar el videojuego construido mediante pruebas de aceptación y Experiencia de Usuario (UX).

1.3 Limitaciones

- **Sistemas operativos**

Para poder ejecutar el videojuego, se requerirá del sistema operativo de Windows 8/10/11 en adelante. Este sistema operativo debe estar instalado y estar corriendo de manera funcional en el computador del usuario.

- **Hardware**

El hardware requerido consta de un teclado y un mouse. El mouse es requerido para poder ejecutar el videojuego; aunque éste puede ser ejecutado con el uso del teclado, se recomienda al usuario que tenga un mouse a su disposición para facilitar el proceso de ejecución del videojuego.

Capítulo 2: Los Juegos de Plataforma

El género de los videojuegos de plataformas 2D ha sido un pilar fundamental en la industria de los videojuegos desde sus primeros días. Estos juegos ofrecen una experiencia de juego icónica que ha mantenido su atractivo para jugadores de todas las edades a lo largo de las décadas. En este capítulo, se explora el estado actual del desarrollo de videojuegos de plataforma 2D y su relación con la Experiencia del Usuario (UX) en Unity.

2.1 Videojuegos de plataforma 2D

Los videojuegos tienen distintas formas de juego, lo que corresponde a un género. Se puede considerar los géneros de los videojuegos como seleccionar el género específico de una película. Se utilizan para clasificar y organizar a los videojuegos por elementos clave de su jugabilidad como, por ejemplo, la forma del juego, los controles y sus respectivos objetivos. Existen una gran variedad de géneros, los más comunes son: estrategias, puzzles, carreras, supervivencia y plataforma.

Dado que el género que estudiamos es el de un videojuego de plataforma, una de las definiciones más acertadas con respecto a éste es el siguiente:

Los videojuegos de plataforma consisten en el jugador teniendo que saltar sobre plataformas suspendidas o a lo largo de obstáculos y enemigos hasta que llegue a una meta definida. Su mecanismo clave consiste de un botón o función que le permita al usuario saltar, siendo éste la parte central de la forma del juego de este género [8].

Pero antes de comenzar con la historia del primer videojuego de plataformas, debemos primero entender en qué se trabaja un videojuego 2D. ¿Qué significa 2D en un videojuego?, ¿cómo funciona el espacio en el que funcionan los videojuegos 2D?, ¿cuál es el nombre de las imágenes que son mostradas en pantalla en un videojuego 2D?

2.1.1 El Entorno 2D

El entorno de un videojuego es el espacio en el que éste se encontrará desarrollado, introduciendo los varios elementos que mostrará en la pantalla al momento de jugar un videojuego. De acuerdo a la guía de Unity:

En los videojuegos 2D, se utiliza un entorno desarrollado en dos dimensiones, que se enfocará primordialmente en gráficos planos llamados “sprites”; estos se dibujan en la pantalla como imágenes planas, que son mostradas por una cámara que mostrará todo el entorno 2D [9].

La cámara en los videojuegos es utilizada con el propósito de mostrar la acción del personaje que estamos controlando en el mejor ángulo posible. Dicha cámara seguirá al personaje² alrededor del escenario en el que el jugador se encuentre, ya sea en la pantalla principal, mostrando elementos de tutoriales para enseñar a cómo jugar el videojuego o también mostrar al personaje en el escenario con los obstáculos y plataformas presentes. Cada videojuego siempre cuenta con una cámara que muestra todo lo que compone un videojuego, ya sea 2D o 3D. Las cámaras en los videojuegos 2D se encuentran en un entorno que tiene como nombre *Espacio bidimensional* (Ver Ilustración 1).

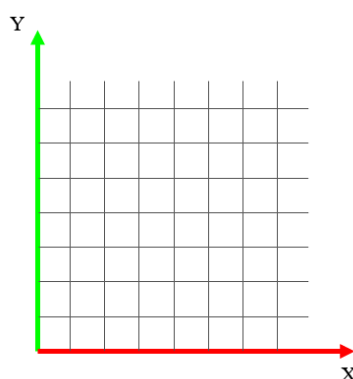


Ilustración 1: Espacio bi-dimensional.

El espacio bidimensional corresponde al eje de coordenadas en el espacio que se representa por dos ejes³, X e Y. El origen del eje de coordenadas es el cruce de los dos ejes X e Y, que es la referencia inicial para posicionar elementos sobre el eje de coordenadas. El eje X corresponde al ancho y representa el dominio donde todo objeto, ya sea el personaje, la cámara, enemigos o incluso obstáculos, se moverán de manera horizontal. El eje Y se trata del alto y representa el dominio donde los elementos se

² Depende de la clase de videojuego. Legend Of Zelda de la NES, por ejemplo, mantiene una cámara estática por cada escenario que el jugador atraviese.

³ También existe un tercer eje, denominado eje Z, que se trata de la profundidad de los elementos. Sin embargo, éste eje no es utilizado comúnmente en los videojuegos 2D, ya que su utilidad se da en los videojuegos 3D.

trasladan de manera vertical como, por ejemplo, el personaje de un videojuego saltando. En los juegos 2D, también se usa el eje Z, o dimensión de profundidad, para indicar qué elementos están encima de otros elementos en el plano. En este caso, el elemento tiene una propiedad que indica su orden para la representación gráfica del juego 2D.

La cámara hace un uso de proyección ortográfica, lo que significa que estamos hablando del mismo tipo de proyección que utilizan nuestros ojos al mirar. Es lo mismo que el campo de visión de un ser humano. En los videojuegos con entorno 2D con un tipo de proyección ortográfica, se utiliza un volumen imaginario del campo de visión de la cámara llamado Frustum. Se trata de un cubo donde el videojuego 2D será mostrado (Ver Ilustración 2).

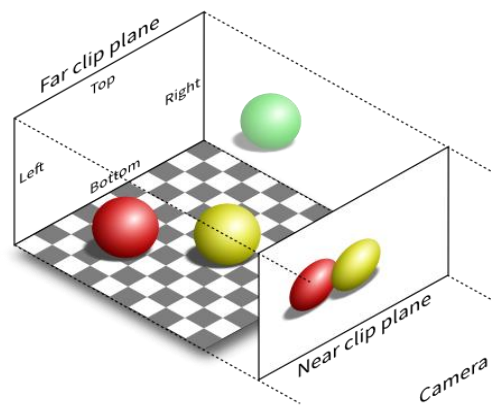


Ilustración 2: Proyección Ortográfica

Un ejemplo claro de la proyección ortográfica viene siendo con el videojuego indie de plataformas 2D, *Cuphead* (2017). Como podemos ver, la cámara muestra al jugador junto con los obstáculos y plataformas, siguiendo al personaje a lo que recorre el nivel. La forma en que se aprecia la proyección ortográfica es que los árboles, por ejemplo, se encuentran detrás del escenario. En realidad, se encuentran en el mismo plano y poseen el mismo tamaño, solo que han sido dibujados de tal forma que se nos hace creer que se encuentran en lo profundo del escenario, cuando en realidad no es así (Ver Ilustración 3).



Ilustración 3: Cuphead en Acción

2.1.2 El primer videojuego de plataforma 2D

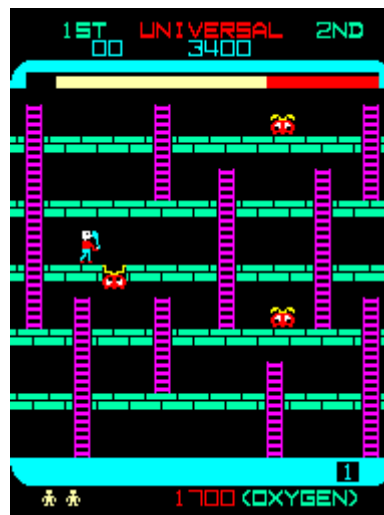


Ilustración 4: Space Panic

El videojuego *Space Panic* fue desarrollado por la compañía Universal (después conocida como Aruze) en el año 1980. Se le considera como el primer videojuego 2D de plataforma según Chris Crawford (diseñador de videojuegos y ex-empleado en *Atari, Inc*). Este videojuego le permite al jugador cavar agujeros en las plataformas para atraer a los enemigos en ellos y subir escaleras (Ver Ilustración 4). El jugador deberá de empujarlos hacia los agujeros hasta que desaparezcan del escenario para poder avanzar a los siguientes niveles.

Un año más tarde, Nintendo, una compañía que anteriormente estaba orientada al desarrollo de juguetes, lanzó su primer videojuego que dominaría la industria de los videojuegos por dos décadas enteras: **Donkey Kong** para las máquinas Arcade (Ver Ilustración 5). Por lo tanto, *Space Panic*, al no poseer un mecanismo de salto, lo descalificó completamente de ser un videojuego 2D de plataforma según varios críticos, y pasó al olvido poco tiempo después [10]. A primera vista, *Donkey Kong* parecía ser una copia directa de *Space Panic*, ya que contenía varias similitudes, como, por ejemplo, las escaleras, las plataformas y uno de los enemigos que asimilaba a los Aliens en *Space Panic*. En el caso de *Donkey Kong*, el fuego.



Ilustración 5: Donkey Kong Arcade

Sin embargo, estas similitudes pronto dejarían a *Space Panic* en el pasado, y *Donkey Kong* se volvería un clásico enorme debido a que implementó por primera vez, en la historia de los videojuegos, la habilidad de saltar. *Donkey Kong* mantuvo una jugabilidad simple al haber implementado la capacidad de saltar: Si un obstáculo se dirigía hacia ti, lo saltabas para evitar dicho obstáculo y obtener puntos como recompensa, otorgando una experiencia satisfactoria para los jugadores. Pero la pregunta viene siendo la siguiente: ¿Por qué saltar en un videojuego se consideró algo muy importante?

2.1.3 ¿Por qué saltamos?

La cantidad de obstáculos que suelen ser presentados en los videojuegos pueden llegar a ser frustrantes para los jugadores. Anteriormente, los personajes en los videojuegos

no podían saltar una simple piedra en el camino, por lo que arruinaba la inmersión y la narrativa en el progreso del videojuego.

Hoy en día, saltar se considera una capacidad o regla que se espera tener en los videojuegos, dado que resulta ser muy útil en un mundo que involucra moverse en un espacio definido, añadiendo realismo y alternativas estratégicas al momento de sentarse a jugar un videojuego. A veces, saltar es lo único que los jugadores hacen, simplemente porque es divertido y no posee ningún otro objetivo.

Jason Begy, un autor y asistente de investigación en la “Technoculture, Art and Games Research Center”, dice:

Saltar sobre precipicios, paredes o escaleras es desafiar a los intentos del ambiente de limitarte o influenciarte en tu navegación por el mundo. El dominar el salto puede ser una forma de ataque o una forma de evadir ciertos obstáculos, pero la dominancia siempre estará presente [11].

El personaje de *Donkey Kong*, Jumpman, salta para poder dominar los obstáculos que se le presentan. Por ejemplo, si Jumpman no salta sobre los barriles, estos lo dominarán. Esto pasaría a ser un concepto que Jason Begy denominó como *Salto Dominante*, que básicamente consiste en que en el mundo real no siempre se les da la oportunidad a las personas de poder controlar sus vidas como desean. Sin embargo, por medio del salto, los videojuegos proveen un mundo virtual en donde la libertad, el dominio y la autonomía puede ser expresado con solo presionar un simple botón [11]. Y esto nos lleva a la siguiente pregunta: ¿Cómo funciona un salto en los videojuegos?

2.1.4 El salto en los videojuegos y su funcionamiento

En simples términos, se trata de una fuerza que se aplica contra una superficie. Esto causa que impulse al personaje en la dirección opuesta en la que la fuerza originaria se aplica. Pero la fuerza aplicada no es el único factor que se debe tomar en cuenta, sino que también se debe contar con elementos físicos, como lo son el peso del personaje, la gravedad, la resistencia del aire, la velocidad, entre muchos otros.

Existen métodos matemáticos y físicos que involucran la utilización de dichos elementos, pero son procedimientos que se estudiaron, analizaron y utilizaron a medida que la tecnología avanzó. Por ahora, analizamos el funcionamiento del salto en los

videojuegos desde sus inicios, con aquel que revolucionó la industria de videojuegos de consola, y que dio una definición concreta a lo que significa ser un videojuego de plataforma, *Super Mario Bros.* de la consola NES (Nintendo Entertainment System) (Ver Ilustración 6).

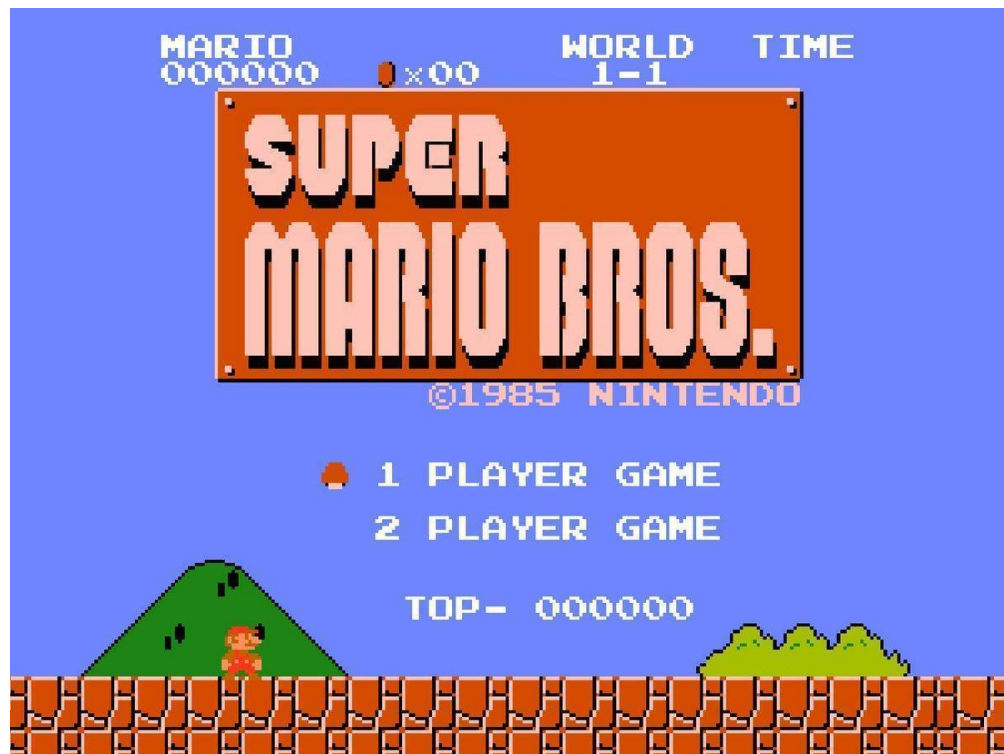


Ilustración 6: Super Mario Bros. para la consola NES

Si comparamos el salto de éste videojuego con el de *Donkey Kong* junto con otros más de esa época, existe una gran diferencia. El salto tiene una altura, velocidad y dirección definida, tanto el salto que realiza en el mismo lugar como el salto cuando se encuentra moviéndose, ya sea desplazándose a la izquierda o a la derecha. Esto puede resultar en un desafío para el jugador, ya que no posee control del personaje una vez ejecutado el salto, resultando en que la única variable en donde tenemos control del personaje es cuándo decidimos saltar.

Una mecánica muy audaz que implementó *Super Mario Bros.* es cómo le permite al jugador controlar la duración del salto, y funciona de la siguiente manera:

- Cada vez que el jugador presiona el botón de salto, se inicia un temporizador. Este temporizador tiene una duración aproximada de 29 frames (alrededor de unos 0,8 segundos).

- Mientras que el botón de salto se mantenga presionado y el temporizador no se haya acabado, el personaje ascenderá, y mientras más se acerque al límite de frames (que son los 29), el personaje comenzará a ralentizarse, y cuando el temporizador se acaba, este comenzará a descender⁴ (Ver Ilustración 7).

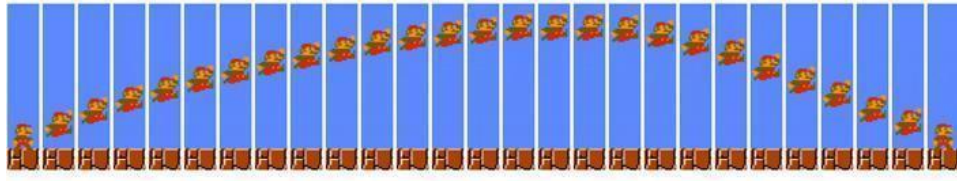


Ilustración 7: Salto de 29 Frames

Esta técnica le permite al jugador pensar que se trata de un movimiento sensible a la presión, cuando en realidad no es así. La razón del porqué esta mecánica resultó ser muy ingeniosa en ese tiempo es simple: les permitió a los desarrolladores de videojuegos poder darles un desafío a los jugadores en términos de diseño. Anteriormente, había solo una clase de salto, y solo permitía saltar una pequeña cantidad de obstáculos u objetos. El jugador poseía un buen control del personaje, pudiéndose mover libremente en el aire mientras mantiene pulsado el botón de salto. Esto significaba que los diseñadores podían desafiar a los jugadores con una gran variedad de obstáculos y no llegar a sentirse limitados por el mismo movimiento del personaje.

En 1988, en *Super Mario Bros. 3.*, el jugador podía controlar a Mario para girar de izquierda o derecha cuando saltaba o mientras se encontraba en el aire. Si bien esto no afectaba en nada a la jugabilidad del videojuego, le daba una sensación de que el usuario tenía un gran control del personaje, ya que la respuesta auditiva y visual son igual de importantes (Ver Ilustración 8).

⁴ Nota: Mientras más rápido corra el personaje, más alto saltará, aumentando el límite de frames en el salto por 31 (de 29).

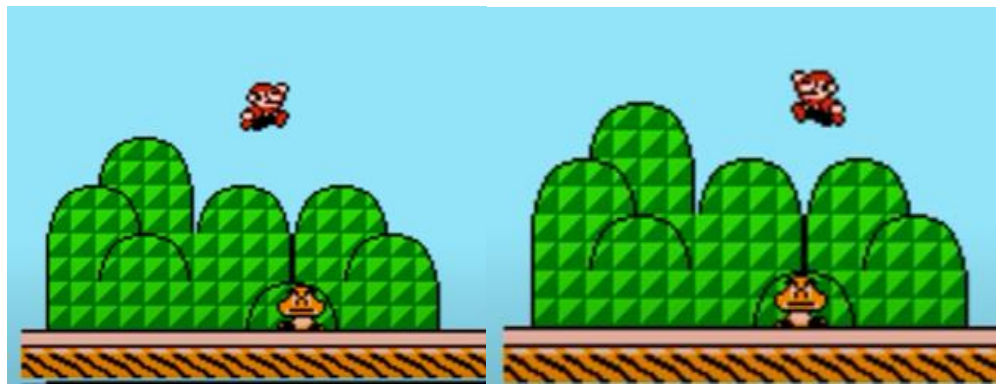


Ilustración 8: Mario cambiando de dirección mientras está en el aire

2.1.5 El desafío de un salto

Lograr que un salto se sienta satisfactorio en un videojuego va más allá de tener mecánicas responsivas; también implica que sea justo y preciso para el jugador. La clave es evitar que el juego se sienta demasiado complicado de manera injusta o demasiado fácil. Este equilibrio es esencial para garantizar que los jugadores disfruten de la experiencia.

Históricamente, los videojuegos han demostrado que el desafío no se limita solo a los saltos, sino a los obstáculos en su conjunto. Desde simples barriles que deben ser saltados en títulos clásicos como Donkey Kong, hasta una amplia gama de obstáculos en juegos modernos, como enemigos, espinas, bombas, fuego y mucho más (Ver Ilustración 9).

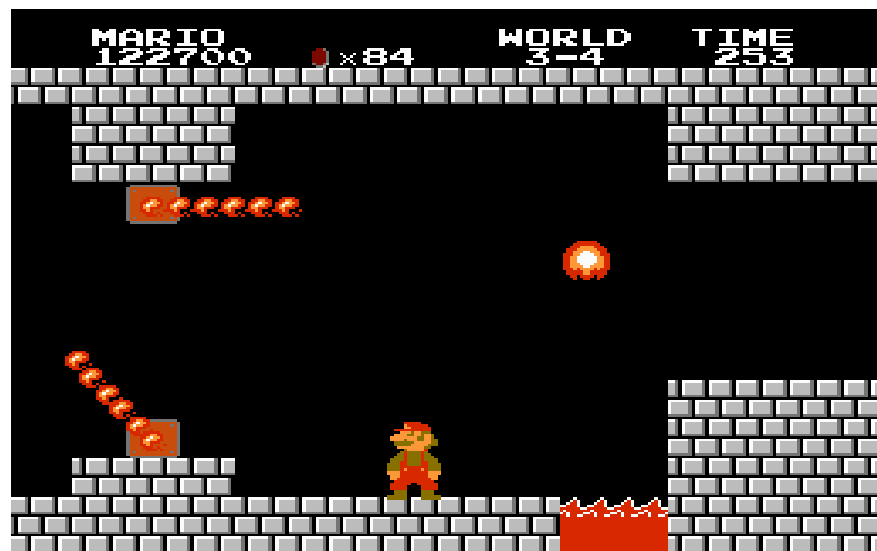


Ilustración 9: Obstáculos en Super Mario Bros

Junto con los obstáculos, otro factor clave que influye en la dificultad de los saltos es la ubicación desde la cual los jugadores saltan. Los jugadores tienden a evitar saltar desde el borde de un precipicio, ya que esta ubicación suele generar nerviosismo. En cambio, prefieren saltar desde una posición que no esté justo en el borde. Scott Rogers [12], un destacado desarrollador de videojuegos conocido por su trabajo en títulos como God of War, Darksiders, Soul Edge y muchos otros, define esta área desde la cual los jugadores saltan como la "zona de salto" [13] (Jump zone) (Ver Ilustración 10).

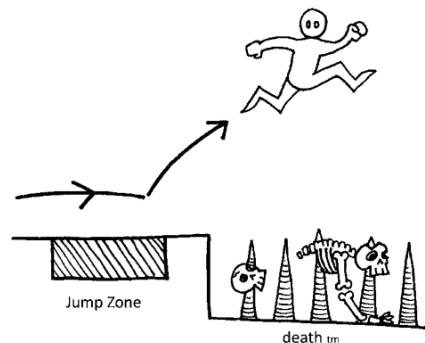


Ilustración 10: Zona de Salto

La idea es proporcionar a los jugadores suficiente espacio para saltar antes de llegar al borde o para esquivar obstáculos mientras están en el aire. Las zonas de salto varían en cada videojuego, ya que no solo deben tener en cuenta la longitud máxima del salto, sino también cómo responde ese salto a las acciones del jugador.

El desarrollo de un desafío en un videojuego es un proceso complejo. Si se hace demasiado difícil, el juego se vuelve injusto; si se hace demasiado fácil, resulta aburrido. La clave está en hacer que los saltos sean desafiantes para los jugadores, pero también accesibles a través de métodos sutiles. Los jugadores deben sentir que han superado un desafío debido a su habilidad, aunque existen elementos que se pueden ajustar de manera imperceptible para hacer que la experiencia sea más amigable. Estos elementos involucran mecánicas relacionadas con los saltos, el diseño de la cámara, los movimientos del personaje, las hitboxes (áreas de colisión) del personaje, entre otros.

La historia de los videojuegos de plataforma 2D comenzó con una mecánica aparentemente simple: el salto. *Donkey Kong* para la Arcade marcó un hito en la industria de los videojuegos, y sin su contribución, la industria no sería lo que es hoy en día. La mecánica de salto única ha evolucionado de diversas formas a lo largo de los

años, y a continuación, se presenta un resumen de las diferentes maneras en que se ha implementado en varios videojuegos.

- *Donkey Kong (1981)*: Fue el primer videojuego de plataformas 2D en haber incluido una mecánica de saltos. En ese tiempo, los saltos eran usados principalmente para evadir obstáculos. Poseían la misma altura y velocidad cada vez que se ejecutaban.
- *Dragon Buster (1985)*: Fue el primer videojuego en haber incluido un doble salto. El personaje principal también tenía la habilidad de poder atacar con su arma mientras saltaba, combinando la mecánica de atacar y saltar al mismo tiempo.
- *Super Mario Bros. (1985)*: Les permitió a los jugadores calcular qué tan alto o qué tan largo querían que su salto fuese, convirtiendo la mecánica de saltar en una habilidad tanto ofensiva como defensiva.
- *Sonic The Hedgehog (1991)*: Creado por Sega como una respuesta a la popularidad que tuvo Mario al convertirse en la mascota de Nintendo, *Sonic The Hedgehog* se caracterizó por su sentido de velocidad, teniendo a la inercia jugando una gran parte en las mecánicas del salto.
- *Super Mario 64 (1996)*: Pese a ser un videojuego 3D, éste marcó nuevamente una plantilla para videojuegos de plataforma, para esta ocasión, para los videojuegos 3D. Sus mecánicas también fueron tomadas para muchos otros videojuegos 2D. *Super Mario 64* involucró un gran rango de habilidades para saltar, dándole a los jugadores toda clase de libertad para lidiar con todos los desafíos presentes en el videojuego. Una de esas habilidades, por ejemplo, constaba de un salto triple.
- *Braid (2008)*: Viniendo de un desarrollador indie, Jonathan Blow, el creador del videojuego, deconstruye el género de plataforma. Forzó a los jugadores a aprender nuevamente lo que habían aprendido de las mecánicas de saltar y las físicas con un sistema de jugabilidad que jugaba con el flujo del tiempo.

2.1.6 Evolución de los Videojuegos de Plataforma 2D

Los videojuegos de plataforma 2D se remontan a los primeros días de la industria de los videojuegos, con títulos clásicos como *Super Mario Bros* y *Sonic the Hedgehog* estableciendo las bases para el género. Estos juegos se caracterizan por la acción en un

entorno bidimensional, donde los jugadores deben superar obstáculos, derrotar enemigos y recoger objetos mientras avanzan de izquierda a derecha.



Ilustración 11: Super Mario Bros. en Acción 2

A lo largo de los años, este género ha experimentado una evolución significativa en términos de diseño, mecánicas de juego y gráficos. Hoy en día, podemos encontrar una amplia variedad de videojuegos de plataforma 2D, desde homenajes nostálgicos a juegos retro hasta títulos modernos que incorporan mecánicas innovadoras y gráficos de alta resolución.



Ilustración 12: Sonic The Hedgehog para la consola Sega Genesis

2.2 Mecánicas y elementos de un videojuego de Plataformas

De acuerdo a la definición propuesta por *Tokio School*:

Las mecánicas de juego son cualquier acción realizada por el jugador que modifique el game state, es decir, la posición y características concretas de todos los objetos y entornos de un juego en un momento preciso en el tiempo. El game state, o estado del videojuego, es modificado debido a una de las características definitorias de los juegos: la interactividad. [14].

En otras palabras, una mecánica es cada función que le permite al jugador realizar acciones en el videojuego de manera que logre alcanzar objetivos propuestos. Según las reglas que el videojuego tenga establecidas, se produce una interacción que genera una respuesta para llevar a cabo acciones sobre los elementos del videojuego, y al mismo tiempo, para que se genere más interacciones.

Como ya hemos analizado en el capítulo anterior, un videojuego de plataforma 2D ofrece la mecánica clave del salto, lo que permite navegar por los niveles del videojuego que se verán más adelante, ya sea saltando plataformas y evadiendo obstáculos. Asimismo, el videojuego de plataforma 2D incorpora que el personaje se mueva a la izquierda o a la derecha hacia una meta definida, junto con coleccionar objetos y atacar enemigos con el salto.

Además del salto, existen otras mecánicas más básicas que son esenciales al momento de la definición y el desarrollo del videojuego de plataforma 2D. Muchos videojuegos dieron origen a variaciones del salto para permitirles al jugador una cómoda experiencia, una nueva forma de explorar los niveles y lograr superar los obstáculos que se le presentan al jugador, dando inicio a una nueva generación de mecánicas. A continuación, se enumeran estas variaciones que provienen de distintos videojuegos y que han sido adaptadas con el pasar de los años.

2.2.1 Variaciones del salto o diferentes tipos

Las variaciones del salto se refieren como una nueva forma de saltar en un videojuego, manteniendo el mismo principio que ya habíamos estudiado, es decir, saltar con una cierta altura y velocidad definida, y al mismo tiempo, añadir una nueva función que expande el concepto del salto. La primera variación del salto es el doble salto.

2.2.2 Doble Salto

Su origen se debe al videojuego japonés de plataforma llamado *Dragon Buster* (1985) (Ver Ilustración 11). Este videojuego de Arcade fue el primero en implementar el concepto de doble salto, dando origen a otras mecánicas de un videojuego de plataforma que se revisan más adelante [15].



Ilustración 13: Double Dragon para la Arcade en Acción

El funcionamiento del doble salto es simple. El jugador presiona el botón del salto para hacer que el personaje en el videojuego salte, y presiona el botón de salto nuevamente para hacerlo saltar a lo que se encuentra en el aire. El concepto de doble salto se refiere a que el personaje podrá saltar en medio del aire para poder evadir obstáculos de manera más sencilla. El jugador puede planear mejores estrategias, y decidir cuándo es el momento más indicado para ejecutar el doble salto y cuándo no. Muchos videojuegos implementaron de inmediato esta mecánica, y es muy utilizada actualmente:

- *Super Ghouls n' Ghosts* (1991): Tenía la mecánica del doble salto ya implementada como una mecánica esencial del videojuego. Con solo un simple salto no se podían conseguir la mayoría de las recompensas o poder esquivar los obstáculos, o incluso poder alcanzar plataformas, por lo que se debía estar usando de manera constante el doble salto.

- *Castlevania Symphony of The Night (1997)*: Tenía la mecánica del doble salto como un recurso que se tenía que conseguir a lo que se progresaba en el videojuego, y era requerido para poder seguir avanzando al objetivo final, porque de otra manera, el jugador se encontraría estancado y no podría terminar el videojuego⁵.
- *Battleblock Theater (2013)*: Introduce la mecánica del doble salto como una de las mil maneras de superar los obstáculos en el videojuego, pudiendo hacer uso de ella tanto para divertirse como para llegar a la meta final del videojuego.

En resumen, el doble salto mantiene el mismo principio que un salto normal. Se puede tener la misma altura y velocidad o bien puede tener estos dos componentes reducidos (también puede venir con una animación visual distinta a la de un salto normal), para hacer que el doble salto se sienta distinto de un salto normal (Ver Ilustración 14).

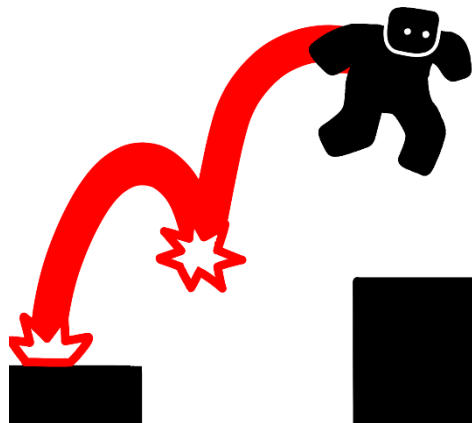


Ilustración 14: Doble Salto

A partir de la descripción del funcionamiento de doble salto, se presenta la siguiente mecánica: Coyote time (o “tiempo coyote”).

⁵ Esto en los videojuegos es referido como “*Progress Wall*” (o muralla de progreso), que corresponde a que el jugador debe obtener un recurso específico para poder seguir avanzando en el juego, si no, se encontrará atascado en una “muralla” y no podrá progresar.

2.2.3 Coyote Time

Antiguamente era conocido como “Ghost Jump” o salto fantasma, que residía en una mecánica escondida por los desarrolladores que le permitía al jugador saltar en medio del aire apenas dejaran la plataforma, dándoles una ventana de tiempo suficiente para poder ejecutar el salto. Sigue más o menos el mismo principio del doble salto, salvo que, en la mayoría de los casos, mantiene la misma velocidad y altura de un salto normal (Ver Ilustración 15).



Ilustración 15: Ejemplo de Coyote Time

Al inicio, pocos juegos tienen esta mecánica escondida. Los más comunes que dieron uso de Coyote Time (que en ese tiempo era conocido como Ghost Jump) fueron *Super Mario World* y *Donkey Kong Country* de la consola de videojuegos Super Nintendo (SNES), con una pequeña excepción en el segundo juego, dado que tenía ciertas modificaciones.

El origen de su nombre se debe a la famosa caricatura animada de Looney Tunes, “*El Correcaminos*”, donde se tiene al personaje, Wile E. Coyote, persiguiendo al correcaminos, desafiando las leyes de la física en numerosas ocasiones, una de ellas resulta ser la forma en que corre de un precipicio, flotando por un buen tiempo antes de que caiga al vacío [16]. La idea es hacer pensar al jugador que saltó justo en el momento preciso antes de caer al vacío, dándole un sentimiento de logro, cuando en realidad, falló en su salto, pero gracias al *Coyote Time*, este termina dando un resultado positivo. En los videojuegos de plataforma que no incluían “*Coyote Time*” como una

mecánica oculta, eran usualmente referidos como videojuegos muy difíciles o nada divertidos. Por ello, los videojuegos de plataforma que hacen uso de dicha mecánica y logran implementarla bien resultan ser videojuegos que son recordados y afamados por los jugadores.

2.2.4 Jump Input Buffering

La mecánica avanzada del salto Jump Input Buffering se refiere a cuando el jugador presiona un botón (en este caso, el botón para saltar) en el videojuego, que le permite ejecutar un comando específico. Jump Input Buffering permite al jugador que el salto se mantenga durante un corto periodo de tiempo después de que el botón fue presionado (Ver Ilustración 16). Para ponerlo en términos más sencillos, al presionar el botón de salto en medio del aire cuando el personaje se encuentre descendiendo, el jugador presionará el botón de salto antes de llegar al suelo, y el personaje en el videojuego saltará una vez que toque el terreno. Esta técnica le da al jugador un pequeño margen de error si es que el salto es presionado con anticipación, haciendo que cada salto intencionado se ejecute de manera exitosa [17].

Supongamos que tenemos un nivel complicado con muchas plataformas. Sin la implementación del Jump Input Buffering, el jugador necesitará que todos sus saltos sean precisos y perfectos, lo que puede llegar a frustrar a muchos de ellos. Y ahí es donde Jump Input Buffering entra en acción.



Ilustración 16: Ejemplo de Jump Input Buffering

Hay muchas formas de implementar esta mecánica, pero la manera más sencilla es que se tendrán dos variables al momento de presionar el botón de salto, que serían

un contador y un número máximo del contador, asimilando el máximo de frames que puede tener un. Entonces, cada vez que el botón de salto sea presionado, el contador tomará el valor del número máximo del contador; por ejemplo, cuatro (Ver Ilustración 17).



Ilustración 17: Explicación de Jump Input Buffering

2.2.5 Wall Jump

La mecánica de Wall Jump se trata de un salto que se puede ejecutar cuando el jugador se encuentra apegado a una muralla en mitad del aire (Ver Ilustración 16). Hace uso de las colisiones (se presentan más adelante), para que la mecánica pueda ejecutarse de manera exitosa. El jugador puede buscar más oportunidades para superar obstáculos con tan solo saltar las murallas. El desarrollador del videojuego, usando la mecánica del *Wall Jump*, puede explorar cómo el videojuego puede desafiar más al jugador.

En general, la forma en que funciona un Wall Jump es debido a la detección de colisiones. Supongamos que nuestro personaje tiene una caja invisible alrededor de éste que posee el mismo tamaño, y dicho personaje se encuentra con una muralla que posee una caja invisible también alrededor de ésta. El personaje, al saltar hacia la muralla, se encontrará con el hecho de que ambas cajas invisibles se conectaron entre ellas, habiendo una detección de colisiones. Por lo tanto, el personaje podrá saltar una vez detecte la colisión o caja invisible con la muralla. Al saltar, la caja invisible de nuestro personaje dejará de estar en contacto con la de la muralla, hasta que se encuentre con otra (Ver Ilustración 18).



Ilustración 18: Ejemplo de Wall Jump

Sus orígenes son variados⁶, pero los más comunes vienen siendo en *Ninja Gaiden* y *Batman: The Video Game* para la consola NES. En el primer videojuego, el personaje

⁶ Técnicamente, el primer Wall Jump viene de *Super Mario Bros.* de la NES, pero más que una mecánica implementada, era más un “glitch” o error del videojuego que permitía ejecutar el Wall Jump.

podía aferrarse a las murallas, escalarlas y saltar en una dirección opuesta en la que el jugador se encontraba, dando inicio al llamado “Wall Jump”.



Ilustración 19: Wall Jump en Acción

En el videojuego *Batman: The Videogame* para la consola NES, el personaje puede utilizar las murallas con colisiones para saltar una tras otra. Las murallas son acompañadas de una visual animada mostrando cómo el personaje se encontrará ejecutando la acción, con el único detalle de que el jugador no tendrá control del personaje una vez realizada dicha acción (Ver Ilustración 20).

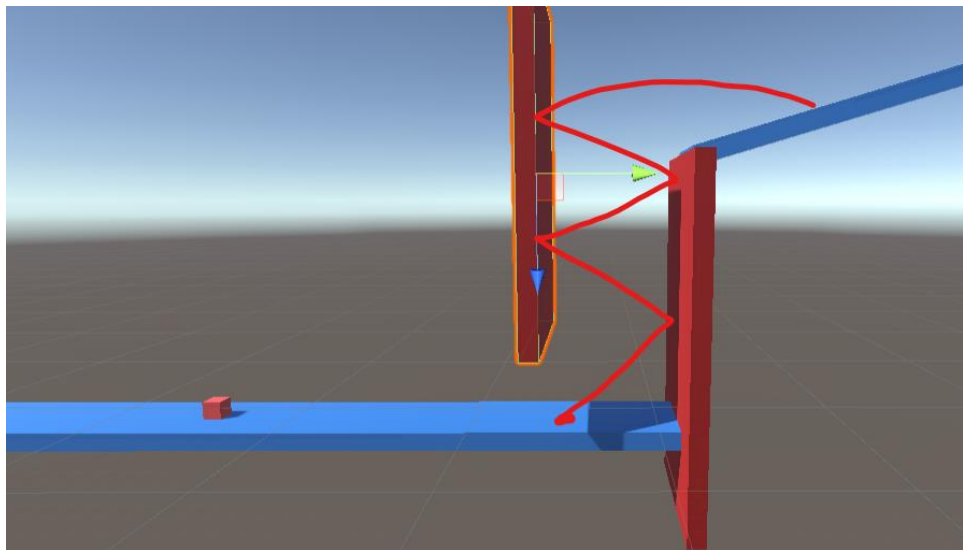


Ilustración 20: Wall Jump en Detalle

La velocidad, control y altura del Wall Jump depende de cada videojuego/desarrollador. Algunos lo dejan con los mismos valores de un salto normal,

otros lo hacen más corto, o simplemente le dan más altura y velocidad. Un control total y visual sobre el personaje le da un sentimiento de satisfacción al jugador.

Algunos videojuegos que utilizan la mecánica del Wall Jump son:

- *Alex Kidd in Shinobi World (1990)*: Era considerado el primer videojuego en implementar el Wall Jump que va de muralla en muralla, aunque como se pudo apreciar, *Batman: The Videogame* para la NES tenía implementada una mecánica similar.
- *Megaman X (1993)*: Introdujo el doble salto por primera vez en su franquicia, haciendo un uso frecuente de esta mecánica en todos los niveles presentes. El personaje al posarse sobre una muralla, este se arrastrará hacia abajo debido a la gravedad, y le permitirá saltar cuando el jugador estime necesario.
- *Super Metroid (1994)*: Exploró la mecánica del doble salto combinado con otra que era la de la gravedad cero, ya que el juego se encontraba ambientado en el espacio. Aunque se tenía total control del personaje alrededor de los niveles, también debía ser usado de manera calculada y estratégica al momento de enfrentarse a enemigos más grandes.
- *Super Meat Boy (2010)*: Es un videojuego indie de plataformas que se concentra enormemente en llegar a la meta final de un nivel, evadiendo los diversos obstáculos que se encuentran presentes. El uso del Wall Jump es vital en este videojuego, ya que con el total control que se tiene del personaje, se debe ser preciso al momento de saltar y moverse en todo momento, esto incluye la mecánica del Wall Jump.

Las variaciones o mecánicas avanzadas del salto en los videojuegos de plataforma resultan ser muy significativas al momento de desarrollar un videojuego. El uso de las variaciones de salto expande los conceptos del salto, y la capacidad del jugador de mover un personaje y poder hacerlo saltar para llegar de una plataforma a otra es característico de un videojuego.

Los videojuegos de hoy en día siempre están intentando innovar sus ideas, diseños y conceptos, tratando de darles una esencia única e inigualable a sus videojuegos, o simplemente tomando una idea tan simple como lo es hacer saltar al

personaje e intentar expandir su habilidad con el salto, puliendo todos los detalles que muchos otros videojuegos no pudieron terminar o no fueron capaces de innovar.

Existen otros saltos en los videojuegos modernos y los desarrolladores de videojuegos pueden crear otras variaciones, con el objetivo de hacer que cada juego tenga su propia esencia distintiva, que lo diferencie de los demás videojuegos.

2.2.6 Hitbox en los videojuegos

La “hitbox” es un elemento que existe en todos los videojuegos, ya sean de tipo 2D o 3D. Usando una forma específica en el videojuego, permite registrar colisiones en tiempo real (Ver Ilustración 21). Por ejemplo, cuando el personaje choca con una pared. Siguiendo la definición propuesta por “Make Use Of”:

Pese a que nos referimos a ellas como una caja de colisiones, éstas pueden tomar cualquier tipo de forma, ya sean formas circulares, esféricas, triangulares, pero la más usada, y la más común, resulta ser un cuadrado o rectángulo [18].

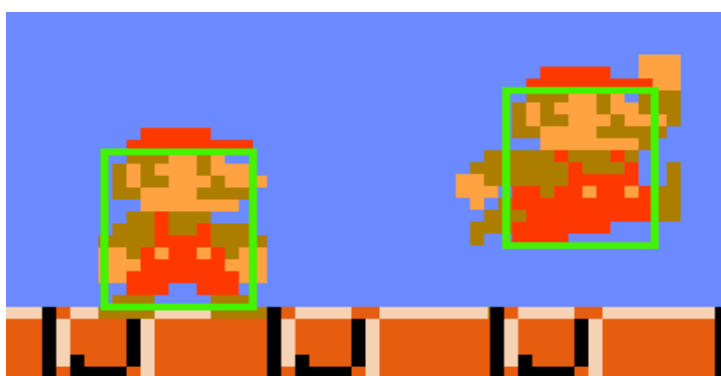


Ilustración 21: Hitbox en Super Mario Bros.

Como podemos observar en la Ilustración 19, el personaje tiene dibujada una caja alrededor de él, que representa la hitbox, y cualquier elemento que haga contacto con ésta tendrá distintos efectos. Si es una pared, el personaje no podrá seguir avanzando; si es un obstáculo enemigo, el personaje perderá y tendrá que reiniciar el nivel. Al mismo tiempo, el mismo terreno en donde se encuentra el personaje también tiene dibujada una hitbox, todo esto para decirle al videojuego que el suelo es un terreno sobre el cuál el personaje puede estar parado, y también podrá moverse en éste. Y al mismo tiempo, los obstáculos poseen su propio hitbox (Ver Ilustración 20).

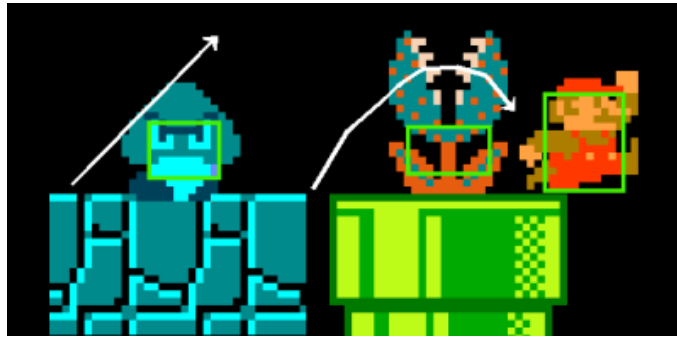


Ilustración 22: Hitbox obstáculos

En resumen, todo elemento en un videojuego que interactúa con el jugador posee una hitbox y, al igual que el mismo salto, debe ser dibujado sobre el personaje de manera precisa. La razón de esto viene siendo a que, si la hitbox resulta ser de, por ejemplo, un tamaño grande, puede llevar a una experiencia frustrante, ya que cada componente que haga contacto con nuestro personaje llega a dar resultados injustos y nada divertidos.

Tener un buen conocimiento de las HitBox ayuda a identificar las áreas exactas en donde el personaje hará contacto con un elemento del videojuego. Una hitbox bien diseñada se correlaciona con un videojuego justo. Son un elemento clave de la experiencia en jugabilidad y un gran contribuyente en qué tan divertido puede ser un videojuego.

2.2.7 Sistema de Salud

Un sistema de salud se refiere a la mecánica que representa la salud actual del jugador, y permite al jugador supervisar su la salud o la del personaje y determinar su capacidad para poder seguir jugando.

El sistema de salud se puede apreciar de varias maneras: Desde una barra de vida, un set de corazones, etc. Cada vez que el jugador impacta con un enemigo u obstáculo, su vida se reduce por un cierto monto. En el caso de ser una barra de salud o un set de corazones, se representan con ambos elementos encogiéndose o desapareciendo.

El sistema de salud puede también incluir objetos que restauran la salud del personaje cuando son coleccionados. Estos se encuentran alrededor de los niveles como objetos coleccionables.

Si la salud del jugador se reduce a cero, pierden el juego o reinicia desde un punto de control.



Ilustración 23: Ejemplo de Sistema de Salud

Las mecánicas de un sistema de salud dependen con el género del videojuego y su dificultad, pero el principal objetivo es otorgar una experiencia desafiante pero justa para el jugador.

2.2.8 Puntos de control

Los puntos de control son objetos o lugares específicos que marcan un punto de progreso para el jugador. Cuando un jugador llega a un punto de control, el juego guarda dicho progreso y crea un punto para reaparecer cuando el jugador, por ejemplo, pierda el juego y tenga que partir desde un cierto punto en vez de comenzar todo el nivel desde cero (Ver Ilustración 24).

Los puntos de control son una mecánica común en los videojuegos que tienen niveles que pueden llegar a ser desafiantes o de larga duración. Son prevaletentes en videojuegos de plataformas, donde el jugador navega por varios obstáculos y enemigos para progresar el nivel. Toma la apariencia de muchas formas, desde banderas, señales, letreros, barriles, entre muchos otros, y son posicionados estratégicamente en ciertas localizaciones alrededor de los niveles.



Ilustración 24: Ejemplo de Checkpoints o Puntos de Control

Por sobre todo, los puntos de control son una parte esencial en el diseño de un videojuego que provee una experiencia justa y disfrutable para el jugador.

2.2.9 Game Flow y sus elementos en un videojuego

Cuando se habla de elementos en un videojuego, nos referimos a componentes que dejan a una persona en un estado de completa inmersión y que solo se concentrará en lo que se encuentra haciendo (en esta caso, jugar un videojuego), ignorando sus alrededores. Dicho estado se denomina *Game Flow* (o *Flujo de Juego*).

Al momento de diseñar un videojuego, el flujo es uno de los aspectos más importantes que deben ser considerados, ya que permite la completa participación del jugador. Se utilizan una gran variedad de elementos para lograr esto, y para esta tesis se utilizan cuatro elementos que son considerados los más importantes al momento de jugar un videojuego:

- **Recompensas:** Suelen ser elementos que atribuyen a los logros del jugador, ya que no basta con que jugar videojuegos sea una grata experiencia, ya que se necesita captar el interés del jugador. Existen varios tipos de recompensas que van desde Objetos Coleccionables, Logros, Desbloques, Sistema de Puntuación, entre muchos otros. En el caso de *Super Mario Bros.*, uno de los varios tipos de recompensas que posee es el de la puntuación y el del objeto. Cada vez que logra

destruir un enemigo, se le atribuye una cantidad de puntos, y al llegar a una cierta cantidad de puntos, se le recompensa una vida extra (Ver Ilustración 21).



Ilustración 25: Recompensas en Super Mario Bros.

Al proporcionar a los jugadores una vasta elección de recompensas les permite cumplir objetivos de mejor manera y una experiencia más comprometida.

- **Metas claras:** Este elemento le muestra al jugador lo que debe lograr y le hace saber lo que tiene que ganar. En la mayoría de los videojuegos, las metas son visualmente claras y visibles, viendo un progreso en la pantalla observando cuánto queda para, por ejemplo, llegar al final de un nivel. Deben ser estructuradas de tal forma que motive a los jugadores, ya que si el jugador no logra saber a dónde dirigirse o no sabe cuál es el objetivo al que tiene que llegar, hará que pierda motivación e inmersión.
- **Feedback directo e inmediato:** De acuerdo a la definición propuesta por eduMe, el Feedback se define como:

Es información proporcionada por un agente con respecto a los aspectos de su desempeño o comprensión. Es una reacción a la acción de los usuarios que tienen como objetivo involucrar, informar y aumentar el conocimiento al reducir las discrepancias entre la comprensión y el desempeño actuales y una meta. [19]

Al recibir Feedback en un videojuego nos ayuda a ver cuánto hemos mejorado en nuestras habilidades. Están diseñados de modo que nos dan una retroalimentación sobre nuestro desempeño, y como respuesta cambiamos nuestras estrategias, nuestro comportamiento que nos ayudan a llegar a la meta, el cual es llegar al final del videojuego. La forma en que el feedback se muestra en un videojuego de una forma típica es:

- Inmediato y sin ningún retraso.
- Frecuente y no intermitente.
- Focalizado en resultados y no en las identidades de los jugadores.
- Una mezcla entre lo positivo y lo negativo.
- Barras de progreso que muestran cuánto falta para llegar a la meta.

Un ejemplo de feedback conciso e inmediato es el videojuego *Dark Souls*, que cada vez que perdemos en el videojuego, nos muestra una pantalla que nos da el efecto de que tenemos que mejorar nuestras habilidades o intentar nuevas estrategias para no llegar a la misma situación (Ver Ilustración 26)



Ilustración 26: Feedback en Dark Souls

- El balance entre las habilidades del jugador y el desafío en un videojuego se refiere a la capacidad del juego para proporcionar una experiencia de juego desafiante y satisfactoria, que permita al jugador sentir que está progresando en el juego a medida que adquiere habilidades y conocimientos, sin que el desafío sea demasiado fácil o demasiado difícil.

Para definir este balance, se pueden tomar en cuenta varios factores, como la dificultad del juego en general, la complejidad de las mecánicas de juego, el ritmo de aprendizaje del jugador, etc.

Una forma común de lograr este balance es a través de la implementación de diferentes niveles de dificultad, que permiten al jugador seleccionar la opción que mejor se adapte a sus habilidades y preferencias de juego. También se pueden utilizar sistemas de progreso que permitan al jugador avanzar a medida que va adquiriendo habilidades y conocimientos, sin que se sienta abrumado por el nivel de dificultad.

2.2.10 Diseño de Nivel

Cada videojuego debe proponer un reto para que el jugador pueda poner a prueba sus habilidades, tanto las del personaje que controla en el videojuego como en las de éste mismo. El diseño de nivel presenta una gran variedad de obstáculos, la forma en que está construido el nivel, coleccionables, entre muchos otros. Por ejemplo, se puede llegar a tener unas espinas en el suelo que deben ser evadidas, enemigos cuyo objetivo es hacernos perder y deben ser derrotados o esquivados para no perder, o incluso una habitación con una puerta con candado que requiere una llave para que esta pueda abrirse, introduciendo el concepto de puzzle en un videojuego. Las posibilidades son infinitas.

Empecemos viendo lo más sencillo que vemos en cada videojuego: Un enemigo (Ver Ilustración 27).



Ilustración 27: Enemigo en Super Mario Bros.

Un enemigo es un personaje no controlable (alias. NPC⁷) cuyo objetivo recae en tratar de detener al jugador en alcanzar la meta final. En *Super Mario Bros.*, existen varios enemigos de distintos tipos en distintos niveles, pero con el mismo objetivo. Cada uno realiza una acción distinta del otro, por ejemplo, en la ilustración 21, el enemigo se encarga de acercarse al personaje en pantalla. Si el personaje en pantalla toca al enemigo, éste perderá. El enemigo se mueve de izquierda a derecha, dependiendo si choca con una muralla o no, debido a como fue programado en su IA (inteligencia artificial). Cada enemigo posee su propia IA, que le permite realizar distintas acciones de los demás enemigos. Estos incluyen la forma en que se comportan, patrullan el escenario, saltar, entre muchas otras.

El siguiente obstáculo son los mismos que presenta el escenario, que también son introducidos para detener el avance del jugador (Ver Ilustración 28).



Ilustración 28: Obstáculos en Super Meat Boy Forever

Como podemos observar en la ilustración 24, el jugador debe llegar a la meta final que es alcanzar al personaje, evadiendo las sierras que presentan un obstáculo para el jugador. El jugador debe hacer un buen uso de los saltos y el movimiento para así no perder el videojuego. Al igual que los enemigos, que pueden venir de varias formas. Puede ser algo tan simple como una sierra tratando de obstruir el paso, o un muro de fuego que se apaga cada cierto segundo, y el personaje debe reconocer el patrón y

⁷ NPC: Non-controlable Player

calcular cuándo avanzar. Cuando éstos dos son combinados, se crea una armonía entre ellos, aumentando la forma en que se trata de desafiar al jugador (Ver Ilustración 29).



Ilustración 29: Obstáculos en Sonic The Hedgehog

Luego tenemos los objetos coleccionables que pueden tomar varias formas. Desde objetos que nos ayudan en el progreso del juego, objetos que desbloquean secretos, entre muchos otros. Muchos desarrolladores utilizan los coleccionables como una forma de darle a los jugadores más información sobre la historia del videojuego o del mismo nivel en el que se encuentran jugando o simplemente dar recompensas. Los coleccionables se pueden evaluar de tres formas:

1. La forma en que son presentadas al jugador.
2. El esfuerzo que toma para obtenerlos.
3. El valor que tienen una vez logras obtenerlos.

Tomemos como ejemplo el videojuego *Sonic The Hedgehog 1* para la consola Sega Genesis. Existe una gran variedad de objetos que ayudan al jugador o que le dan una recompensa. Consideremos las argollas presentes en el videojuego (Ver Ilustración 30). El jugador requiere de estas argollas para poder recibir más de un impacto, y si el jugador tiene cero argollas, un solo golpe le hace perder el juego. El coleccionar este objeto no solo nos permite recibir más de un impacto, sino que también nos permite recibir una puntuación, y con la suficiente cantidad de argollas (que suelen ser cien) y puntuación gracias a éstas, al jugador también se le recompensa con una vida extra.



Ilustración 30: Argollas en Sonic The Hedgehog

Un objeto coleccionable puede tener un gran valor para el jugador de forma sencilla, siempre y cuando el videojuego lo presente de manera ingeniosa.

El concepto de diseñar niveles va más allá de simplemente introducir obstáculos, enemigos y objetos coleccionables puestos al azar, ya que también necesita pintar un nuevo mundo en el que el jugador y el personaje estarán explorando juntos. Cada videojuego contiene su propia historia, y necesita ser contada por medio del ambiente que hay en el juego y por medio de los personajes. Tomemos el videojuego de plataformas 2D *"Timothy and the Tower of Mu"* (Ver Ilustración 31) para PC como ejemplo y analicemos su historia basada en su diseño de nivel: ¿Por qué estamos en un castillo?, ¿por qué el castillo se ve viejo y destruido?, ¿por qué iniciamos en el castillo y no en un bosque?



Ilustración 31: Timothy and the Tower of Mu

Todas esas preguntas se nos hacen al iniciar cada videojuego con su propia historia y su propio ambiente. Estamos en un castillo porque en la cima de dicho castillo se puede pedir un deseo y nuestro personaje desea salvar a su abuelo. El castillo se ve destruido probablemente porque han pasado años desde que alguien ha habitado sus pasillos.

Esta clase de historia que son contadas por medio de los niveles provienen de un concepto llamado: "Environmental Storytelling", que significa contar historias por medio del ambiente presentados en el videojuego.

Por lo tanto, los diseñadores de niveles son los encargados de la implementación de la jugabilidad o gameplay de un videojuego; en la modernidad, hablar de un "nivel" es minimizar la complejidad. Se introduce el concepto de "diseñador de espacios de gameplay" para asimilar de mejor manera el trabajo de un diseñador. Tanto un Level

Designer es tratado como un Game Designer (diseñador de videojuegos). Al mismo tiempo, se debe tomar en cuenta el pacing o progreso del juego, que vendría siendo la relación entre una actividad, la intensidad de la misma y su duración. Es decir, ¿cuál será el ritmo en cuanto a intensidad durante toda la experiencia desde que comienza el nivel hasta que termina?

Para finalizar con un ejemplo: los primeros niveles son construídos, la mayoría del tiempo, de la siguiente manera:

- Se crea un arquitecto en el mundo virtual.
- Se crean espacios jugables.
 - Se guía al jugador con dichos espacios.
- Se controla el flujo del objetivo.
- Se programan eventos.

2.2.11 Sistema de Invulnerabilidad

En el contexto de un videojuego de plataforma 2D, un sistema de invulnerabilidad se refiere a un mecanismo que otorga temporalmente al jugador o a un personaje la incapacidad de recibir daño después de haber sufrido algún tipo de ataque o evento perjudicial. Este sistema de invulnerabilidad es comúnmente conocido como "invencibilidad" o "invulnerabilidad temporal".

El propósito principal de este sistema es proporcionar al jugador un breve período de tiempo durante el cual no puede ser dañado después de haber enfrentado una situación peligrosa, como ser golpeado por un enemigo, tocar un objeto dañino o cualquier otro evento que normalmente causaría daño al personaje.

Algunas características comunes de un sistema de invulnerabilidad en juegos de plataformas 2D incluyen:

- **Duración Temporal:** La invulnerabilidad generalmente tiene una duración limitada, después de la cual el jugador vuelve a ser vulnerable a los ataques.
- **Efectos Visuales:** Para indicar visualmente que el personaje está en un estado de invulnerabilidad, se suelen incorporar efectos visuales como destellos,

cambios de color o algún otro indicador visual que destaque la temporal invulnerabilidad.

- **Retroalimentación Auditiva:** Puede haber efectos de sonido específicos que acompañen al estado de invulnerabilidad para proporcionar una indicación adicional al jugador.
- **Uso Estratégico:** Los sistemas de invulnerabilidad a menudo se diseñan para fomentar estrategias tácticas por parte del jugador. Por ejemplo, el jugador podría aprovechar este estado temporal para atravesar obstáculos peligrosos o enfrentarse a enemigos sin temor inmediato.

En resumen, un sistema de invulnerabilidad en un juego de plataforma 2D agrega una capa de estrategia y seguridad temporal, proporcionando a los jugadores la oportunidad de recuperarse después de eventos perjudiciales y continuar con el juego.

2.3 Experiencia de Usuario (UX)

Sin lugar a dudas, la experiencia del usuario es uno de los aspectos más cruciales en el desarrollo de videojuegos. Se trata de una disciplina cuyo propósito radica en la mejora de la experiencia y el método mediante el cual las personas interactúan con un producto. Aunque no existe una definición única y exacta para lo que es la Experiencia del Usuario (UX), Starloop ofrece una definición que arroja luz sobre este concepto:

Es una rama del diseño de videojuegos centrada en la psicología de los jugadores, su comportamiento y el proceso de su pensamiento. A través de la UX, los desarrolladores de videojuegos se aseguran de que la visión que querían crear llegará sin cambios a la mente de los usuarios. Por lo tanto, necesitan un conocimiento profundo de los procesos de pensamiento y comportamiento de los jugadores. [20]

La Experiencia de Usuario, comúnmente abreviada como UX, se asocia frecuentemente con la interfaz de usuario (UI) ya que involucra elementos de navegación y usabilidad, aplicados en contextos como sitios web, aplicaciones móviles o videojuegos.

La implementación de una buena experiencia de usuario en un producto implica tener en cuenta diversos aspectos cognitivos, que abarcan desde las expectativas funcionales hasta las emocionales. Cuando se trata de incorporar estos principios en el diseño de videojuegos, existen varias estrategias para lograr una aplicación efectiva de la UX. Los desarrolladores buscan crear experiencias de juego que sean atractivas, inmersivas y satisfactorias para los jugadores. La UX abarca varios aspectos clave en los juegos de plataforma 2D:

2.3.1 Señales y Feedbacks

En el mundo del diseño de videojuegos, las señales y los feedbacks son elementos clave para mejorar la experiencia del jugador. Señales se refiere a las informaciones proporcionadas a los jugadores para avisarles de un evento inminente, indicando que deben tomar una acción específica. Tomemos el ejemplo del videojuego "Star Wars Jedi: Fallen Order". Cuando un enemigo se ilumina de rojo (acompañado de un sonido), se señala que su ataque no puede ser bloqueado, lo que requiere que el jugador esquive en lugar de bloquear, otorgándole una ventana de tiempo para reaccionar antes del impacto. (Ver Ilustración 32).



Ilustración 32: Señales en Star Wars Jedi: Fallen Order

Por otro lado, los Feedbacks son respuestas inmediatas a las acciones del jugador que indican si tuvo éxito o fracaso en su elección. Estos feedbacks pueden manifestarse visualmente, auditivamente o incluso físicamente, como la vibración del control. Los feedbacks se consideran recompensas que hacen que cada acción sea más gratificante. Tomemos como ejemplo "Call of Duty Mobile", donde cada vez que un jugador elimina a un enemigo, se le otorga una puntuación, confirmando su éxito y recompensándolo por su habilidad. (Ver Ilustración 33).



Ilustración 33: Feedback en Call of Duty Mobile

En el diseño de videojuegos, los diseñadores incorporan cuidadosamente estas señales y feedbacks para que los jugadores puedan interactuar de manera efectiva en diversas situaciones. Esto no solo agrega un toque único a la jugabilidad, sino que

también permite a los jugadores sentirse recompensados y motivados a seguir avanzando.

Los feedbacks también desempeñan un papel crucial al proporcionar información y emociones en el videojuego, dando a los jugadores una idea clara de lo que pueden lograr y haciéndolos más comprensibles y atractivos. Esto se logra a través de aspectos visuales, auditivos y el control del personaje, lo que contribuye significativamente a la inmersión en el juego.

En el año 2019 se lanzó el videojuego *Phoenix Wright: Ace Attorney Trilogy*, donde cada personaje, diálogo y sonido están diseñados para expresar las diversas emociones presentes en el juego. La forma en que un personaje muestra miedo, enojo, duda, entre otras emociones, se refleja tanto en el diálogo como en las animaciones de los personajes. Incluso la cámara se mueve de manera frenética cuando un personaje grita, creando una experiencia inmersiva.

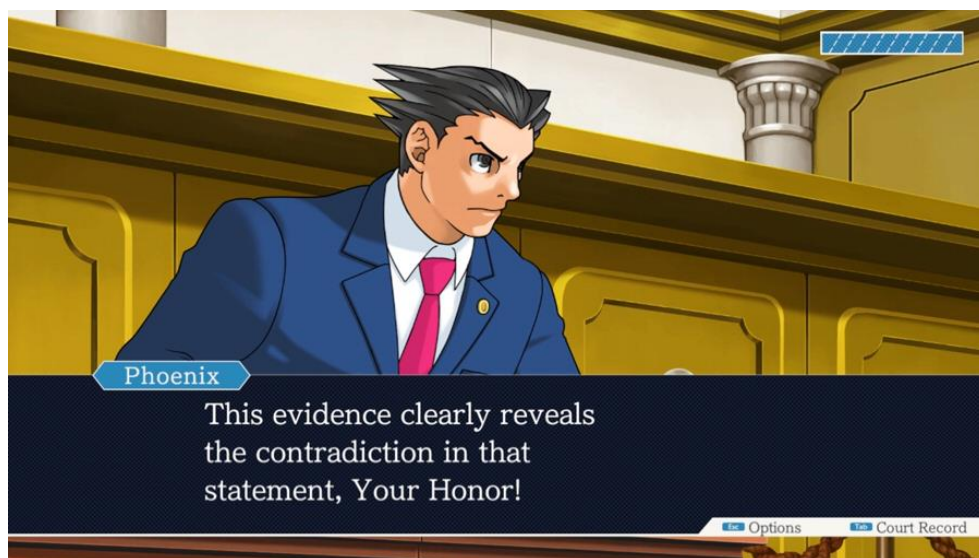


Ilustración 34: Phoenix Wright Ace Attorney Trilogy en Acción

¿Qué conclusiones podemos extraer de esto? Observamos cómo ciertas palabras se realzan para otorgarles un impacto más profundo, ya sea a través de animaciones, efectos sonoros, paletas de colores y movimientos de cámara que añaden profundidad a los diálogos entre los personajes. Estos elementos visuales y auditivos nos permiten apreciar la importancia de ciertas líneas, al mismo tiempo que desarrollamos empatía por los personajes de la historia. Todo esto se logra a través de la combinación de texto y la representación escénica.

2.3.2 Beneficios del UX en los videojuegos

A través de estos ejemplos, podemos apreciar cómo las mecánicas simples pueden aprovecharse para enriquecer la profundidad del UX en los videojuegos.

Cuando se diseña teniendo en mente la experiencia del juego, el principal objetivo es brindar una amplia gama de respuestas satisfactorias en cada acción.

Y cuando hablamos de inmersión, no se refiere únicamente a gráficos deslumbrantes, sino a la coherencia del juego dentro de su universo ficticio, y a cómo la sensación de juego hace que la experiencia resulte aún más atractiva.

Explorar las posibilidades de interacción entre los jugadores y el videojuego, y descubrir cómo fortalecerlas a través de las sensaciones y emociones generadas por la jugabilidad, el diseño de niveles y la narrativa del juego, es un desafío fundamental en el desarrollo de videojuegos.

2.3.3 Onboarding y Usability

En el desarrollo de videojuegos, cuando se busca la forma de interactuar con el jugador y cómo reforzarla, se utilizan dos procesos llamados *Onboarding* y *Usability* (o usabilidad).

De acuerdo a la definición otorgada por Yu-Kai Chou, autor sobre gamificación y diseño conductual:

Onboarding consiste en enseñar a los usuarios las reglas y herramientas para jugar. La incorporación comienza tan pronto como el usuario se registra y finaliza cuando los usuarios dominan las habilidades fundamentales necesarias para jugar y alcanzar los estados ganadores de la etapa inicial [21].

En otras palabras, el *Onboarding* trata de guiar a los nuevos jugadores a través de las mecánicas del juego, los controles y otros aspectos esenciales para que puedan comenzar a jugar y entender cómo funciona el juego. Puede incluir tutoriales, consejos, pistas visuales y otros elementos que ayudan a los jugadores a aprender mientras juegan.



Ilustración 35: Tutorial en Kirby Star Allies

Un *Onboarding* efectivo es crucial para retener a los jugadores y garantizar una experiencia de juego satisfactoria desde el principio. Y al momento de implementar el *Onboarding* en un videojuego, es aquí donde nacen preguntas como: ¿Cómo enseñarles a los jugadores el objetivo del juego?, ¿cómo desarrollarán los jugadores la forma en que juegan el videojuego?, ¿cómo validar el conocimiento de los jugadores con respecto a los distintos aspectos del videojuego (mecánicas, narrativa, música)?

Por otro lado, la usabilidad o *Usability*, de acuerdo a HeadSpin, la define como:

La usabilidad es la métrica utilizada para medir qué tan bien un usuario específico en un contexto particular puede usar un producto o diseño para lograr un objetivo definido de manera efectiva, eficiente y satisfactoria. No es una característica única y unidimensional de un producto o sistema. La usabilidad de un producto representa una combinación de factores, incluido el diseño intuitivo, la facilidad de aprendizaje, la memorabilidad, la frecuencia y gravedad de los errores y la satisfacción subjetiva. Sin embargo, ofrecer la mejor experiencia de usuario con menos interrupciones y desafíos innecesarios se considera usabilidad en la industria del juego [22].

En el contexto de los videojuegos, implica que los controles y las mecánicas del juego deben ser intuitivos y fáciles de entender. Los jugadores no deben enfrentar obstáculos innecesarios para disfrutar del juego. Esto incluye la disposición de los controles, la

claridad en las indicaciones visuales y auditivas, y la capacidad de los jugadores para aprender y utilizar las mecánicas del juego sin sentirse frustrados.

Es un concepto fundamental en el diseño de productos y sistemas, y su objetivo principal es asegurarse de que los usuarios puedan utilizar un producto de manera efectiva, eficiente y satisfactoria.

Tanto el onboarding como la usabilidad son aspectos esenciales del diseño de juegos exitosos, ya que garantizan que los jugadores tengan una experiencia de juego atractiva y accesible, lo que a su vez puede contribuir a una mayor retención de los jugadores y a la satisfacción general del usuario.

2.3.4 Jugabilidad Intuitiva

La jugabilidad en los videojuegos de plataforma 2D debe ser intuitiva. Los controles deben ser fáciles de aprender y permitir que los jugadores se sumerjan en la acción sin problemas. La respuesta a las entradas del jugador, como saltar y moverse, debe ser precisa y receptiva. Un ejemplo claro de jugabilidad intuitiva es el videojuego *Donkey Kong Country* para la consola SNES.



Ilustración 36: Jugabilidad intuitiva en Donkey Kong Country

2.3.5 Diseño de Niveles

El diseño de niveles desempeña un papel fundamental en la experiencia del usuario en juegos de plataforma 2D. Los niveles deben ser desafiantes pero justos, evitando

frustrar a los jugadores. La progresión debe ser fluida, y los obstáculos deben presentarse de manera que fomente la exploración y el aprendizaje.



Ilustración 37: Detalles de Nivel en Castlevania Symphony of the Night

2.3.6 Estilo Visual y Audio

El aspecto visual y el sonido son aspectos cruciales para la UX. Los juegos de plataforma 2D a menudo se distinguen por sus gráficos vibrantes y estilos artísticos únicos. La música y los efectos de sonido contribuyen a la atmósfera del juego y pueden influir en las emociones del jugador.



Ilustración 38: Estilo Visual en Pizza Tower

2.3.7 Evaluación de la Experiencia de Usuario (UX) en Videojuegos de Plataforma 2D

Durante el desarrollo de un videojuego de plataforma 2D, es fundamental someter cada aspecto del juego a una evaluación rigurosa. Esta evaluación minuciosa desempeña un papel crucial ya que permite a los desarrolladores recopilar información valiosa y obtener retroalimentación esencial sobre diversos elementos del juego. Además, recibir opiniones de terceros, como jugadores beta o expertos en el campo, es fundamental para garantizar que el juego cumpla con los estándares de calidad y satisfacción de los jugadores.

La evaluación durante el desarrollo no se limita a corregir errores, sino que también permite optimizar la experiencia del usuario, garantizando que los jugadores disfruten del juego en su máxima expresión.

Para llevar a cabo esta evaluación, existen varios modelos que facilitan la creación de una lista de las funcionalidades del videojuego, con William J. Shelstad, Barbara S. Chaparro y Joseph R. Keebler investigando en su texto "Assessing the User Experience of Video Games: Relationships Between Three Scales" [23] y el modelo GEQ de K.L. Norman:

- **ENJOY (Davidson, 2018) [24]:** Este modelo fue diseñado para medir el nivel de entretenimiento de cualquier actividad. Evalúa aspectos como el placer, la relación con el juego, la competencia, el desafío/mejora y el compromiso del videojuego.
- **GUESS-24 (Phan et al., 2016) [25]:** Fue desarrollado para medir la satisfacción del usuario en un videojuego y permite realizar pruebas rápidas e investigaciones. Esto facilita que los desarrolladores y diseñadores puedan utilizar la escala en pruebas iterativas del videojuego. Este modelo busca medir diversos constructos que contribuyen a la satisfacción general del videojuego, como usabilidad/jugabilidad, narrativa, diversión, libertad creativa, estética sonora, gratificación personal, conectividad social y estética visual.
- **UEQ-S (Schrepp et al., 2017) [26]:** Este modelo fue creado como una medida integral de la experiencia del usuario (basado en el trabajo previo de Laugwitz, Held y Schrepp, 2008). El UEQ-S consta de dos subescalas: calidad pragmática y

calidad hedónica. Este modelo es versátil y se adapta a diferentes situaciones, como cuando se necesita un cuestionario rápido de experiencia del usuario o cuando se comparan varios productos u opciones en una misma sesión.

- **GEQ (Norman, K. L., 2013) [27]:** Este modelo se separa en dos cuestionarios, uno que revisa la experiencia del juego general y otra sobre la evaluación violenta de los videojuegos y cómo los jugadores reaccionan ante esto, ambos centrándose en el desarrollo de cuestionarios para medir la participación en los videojuegos. El primer modelo, “GexpQ” o Cuestionario de Experiencia de Juego, está diseñado para probar juegos y evaluar experiencias subjetivas. Sin embargo, carece de respaldo empírico sobre la confiabilidad del cuestionario. El segundo modelo, “GengQ” o Cuestionario de Participación en el Juego, tiene como objetivo identificar a individuos susceptibles a la violencia en los videojuegos. El GengQ se somete a un extenso desarrollo y pruebas psicométricas, demostrando confiabilidad y validez. Sin embargo, el enfoque en juegos violentos limita su aplicación en contextos más amplios. Es destacable la necesidad de conciliar las características del jugador y las características del juego, enfatizando la importancia de evaluar la participación como un rasgo positivo.

2.3.9 Herramientas para el Desarrollo de Videojuegos de Plataforma 2D en Unity

La elección de la plataforma de desarrollo es esencial para la creación de videojuegos de plataforma 2D. Unity, uno de los motores de juego más populares, ha demostrado ser una elección común para desarrolladores de juegos 2D. Unity proporciona una amplia gama de herramientas y recursos para simplificar el proceso de desarrollo, desde la creación de activos hasta la implementación de mecánicas de juego.

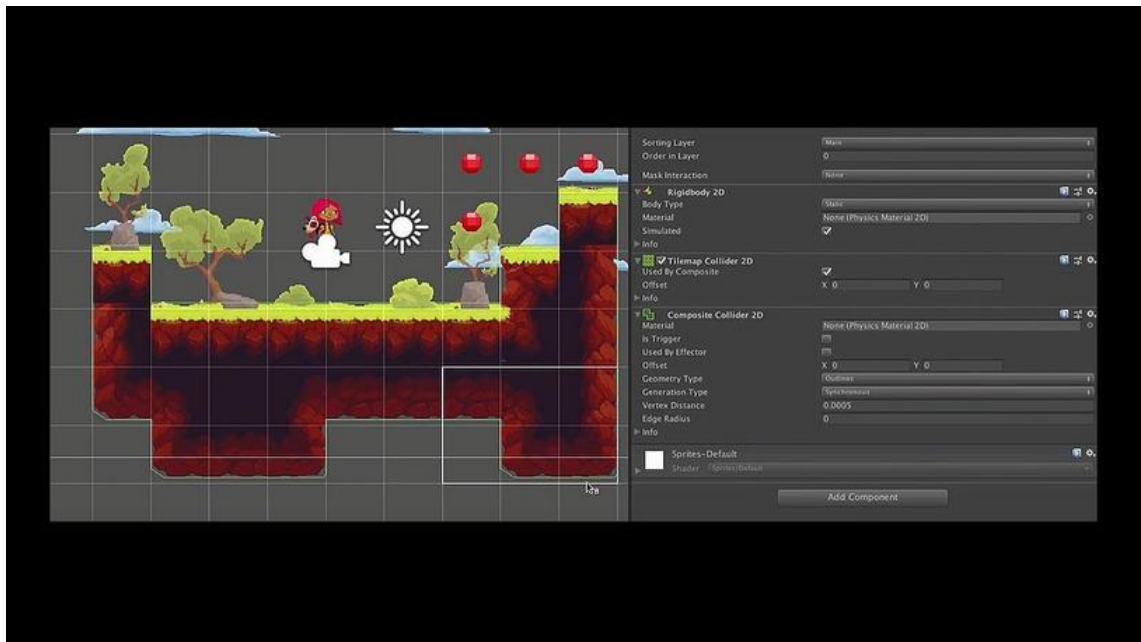


Ilustración 39: Vista general de Unity

2.3.10 Casos de Estudio Relevantes

Hollow Knight: Este aclamado juego de plataforma 2D destaca por su hermoso diseño de niveles, su jugabilidad pulida y su profunda inmersión en su mundo oscuro y misterioso. La UX en "Hollow Knight" se logra mediante un diseño de niveles ingenioso y un equilibrio cuidadoso entre desafío y recompensa.



Ilustración 40: Hollow Knight en Acción

Celeste: Este videojuego es un ejemplo de cómo los juegos de plataforma 2D pueden abordar temas emocionales y personales. La UX se destaca por su narrativa conmovedora y su atención meticulosa a los detalles en el diseño de niveles.

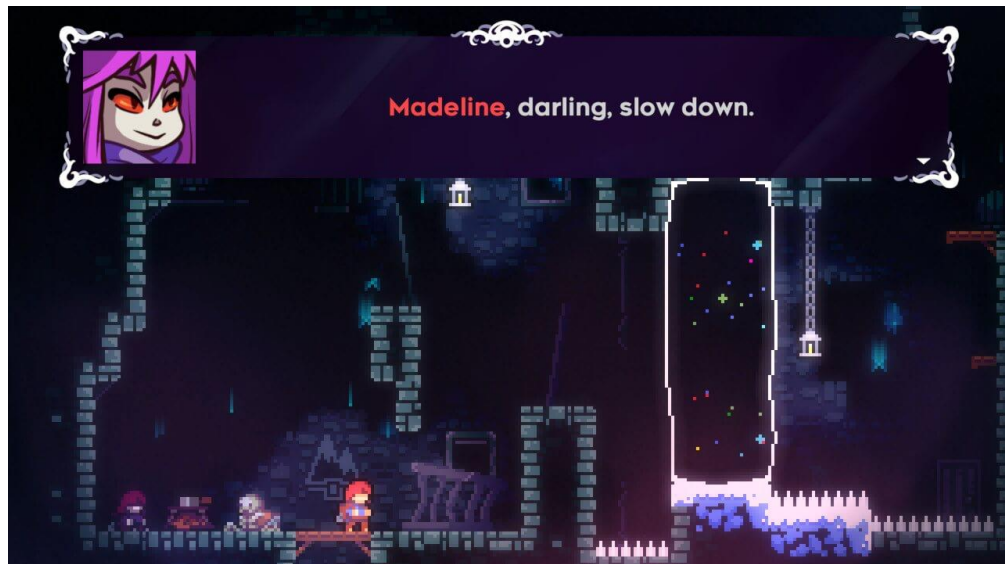


Ilustración 41: Celeste en Acción

2.3.11 Conclusiones

Es importante destacar que los desafíos relacionados con la Experiencia del Usuario (UX) son una constante en el desarrollo de videojuegos. Cuando se enfrentan a estos desafíos, los diseñadores y desarrolladores pueden recurrir a un enfoque analítico para encontrar soluciones efectivas. Inicialmente, esto implica el análisis de otros juegos con aspectos de diseño similares, donde se identifican las necesidades de los jugadores para realizar tareas específicas y se exploran los posibles enfoques para abordarlas. Este proceso también implica la consideración de una amplia gama de reacciones y comportamientos que los jugadores pueden manifestar en situaciones particulares.

Una vez que se ha identificado una posible solución, es fundamental someterla a una serie de preguntas clave:

- ¿Pueden todos los jugadores interpretar la información correctamente?
- ¿La solución se alinea con las intenciones establecidas en el diseño del juego?
- ¿Existen opciones alternativas que podrían ser más adecuadas?

Para evaluar la efectividad de estas soluciones, se realizan pruebas de juego que permiten la iteración y el refinamiento continuo. Posteriormente, se pueden implementar encuestas que brinden a los jugadores la oportunidad de compartir sus perspectivas valiosas.

El objetivo de este enfoque es obtener una comprensión más profunda del funcionamiento del juego, así como de las emociones, tanto positivas como negativas, que experimentan los jugadores al interactuar con él. Al formular preguntas abiertas desde la perspectiva de los usuarios, se obtiene una visión auténtica y valiosa que resulta esencial al evaluar la experiencia global del juego.

Capítulo 3: Metodología

Para llevar a cabo el desarrollo de *Star Forest*, se ha optado por la implementación de una metodología iterativa incremental. De acuerdo a la definición descrita por León Yacelga, Andrés Roberto, Acosta Espinoza, Jorge Lenin, & Díaz Vásquez, Rita Azucena. en su revista “*Aplicación de la metodología incremental en el desarrollo de sistemas de información*”:

La metodología incremental parte de un diseño inicial con características básicas, conocida como bosquejo de descripción y luego conforme avanza el desarrollo se van realizando versiones cada vez más completas del sistema, hasta llegar a una versión final que satisface completamente las necesidades del usuario y cumple con todos los requerimientos de información y administración de esta. [28].

Esta metodología se caracteriza por organizar el proceso de desarrollo en ciclos, cada uno representado por fases bien definidas, y evoluciona de manera secuencial y estructurada. La clave de esta metodología radica en comenzar con una implementación simple de los requerimientos del sistema y, progresivamente, mejorar la secuencia evolutiva mediante un sistema de versiones. En cada ciclo, se tiene la flexibilidad de realizar ajustes en el diseño y realizar cambios en las funcionalidades, ya sea agregando o eliminando según las necesidades emergentes.

Ventajas

- A medida que los componentes se van incorporando, se pueden adaptar requisitos que puedan surgir.
- Es un modelo fácil de probar y depurar.
- Se reduce el riesgo de cambios en los requisitos.

Desventajas

- El modelo incremental necesita una planificación extensa para integrar cada componente. Esta planificación puede retrasar el proyecto.
- Se necesita un diseño adecuado para integrar cada componente.

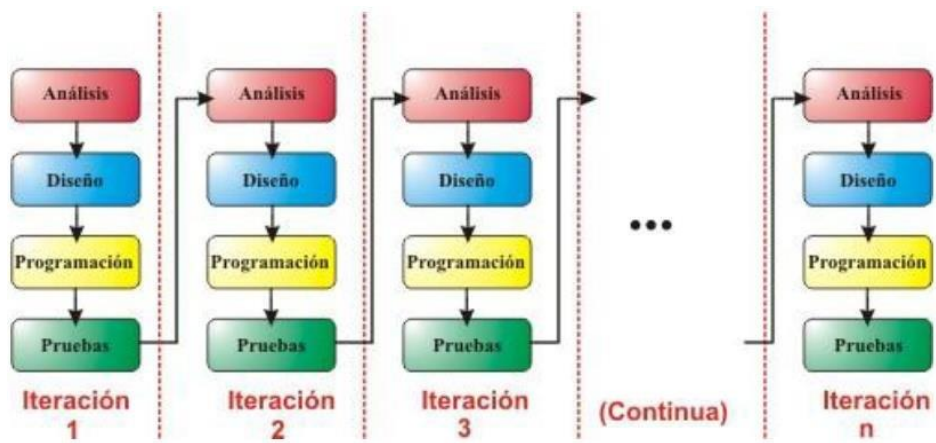


Ilustración 42: Diagrama de metodología iterativa incremental

Se incorporan nuevos objetivos/requisitos y se optimizan aquellos ya abordados con base en los logros obtenidos en las iteraciones previas. Un elemento crucial para orientar el desarrollo de manera iterativa e incremental consiste en la jerarquización de los objetivos/requisitos.

3.1 Fases de la Metodología

1. Planificación del juego (2 semanas): Se planifica el diseño del juego, incluyendo su temática, personajes, escenarios, mecánicas de juego y objetivos.

- Concepto del Videojuego.
- Lista de Enemigos.
- Diseño de escenario.
- Lista de mecánicas.
- Objetivos del videojuego.

2. Programación y Pruebas básica del juego (6 semanas): Se realiza la programación básica del juego en el motor gráfico, incluyendo la creación del personaje controlable, la definición de las mecánicas de juego, entre otras.

- Crear movimiento básico del personaje.
- Crear mecánica de doble salto.
- Crear mecánica de saltar en la muralla.
- Animar al personaje.
- Asignar físicas al personaje.
- Cambiar la dirección del personaje.
- Pruebas de experiencia de usuario (UX).
- Pruebas de Aceptación

3. Diseño de niveles (10 semanas): Se diseñan los niveles del juego, considerando aspectos como el flujo del juego, la dificultad progresiva y los desafíos.

- Crear paleta de terreno.
- Añadir físicas.
- Añadir objetos coleccionables.
- Añadir obstáculos.
- Añadir enemigo básico.
- Añadir puntos de control.

4. Programación y Pruebas de Pulido y refinamiento (11 semanas): Esta etapa implica la revisión y optimización del código del juego, eliminando errores, ajustando la mecánica del juego y perfeccionando el diseño de los niveles.

- Incorporar sistema de salud.
- Incorporar gráficas para el sistema de salud.
- Incorporar efecto parallax.
- Añadir invencibilidad.
- Añadir retroceso.
- Añadir enemigo volador.
- Añadir enemigo final.
- Añadir objetos coleccionables.
- Añadir plataformas flotantes.
- Pruebas de experiencia de usuario (UX).
- Pruebas de aceptación.

5. Integración de elementos multimedia (5 semanas): Se integran los elementos multimedia del juego, como efectos de sonido, música y efectos visuales.

- Añadir efectos de sonido.
- Añadir música.
- Añadir menú.
- Añadir selección de niveles.
- Pruebas de experiencia de usuario (UX).

6. Pruebas de experiencia de usuario (UX) (11 semanas): En esta fase, se ejecutan pruebas exhaustivas del juego con el objetivo de identificar errores, fallas y otros problemas para su subsanación. Katherine Isbister y Celia Hodent, en su libro *Game Usability: Advice from the Experts for Advancing UX Strategy and Practice in Videogames* [29] proporcionan una guía integral para la implementación de diversas pruebas destinadas a evaluar la Experiencia del Usuario (UX). Estos pasos se dividen de la siguiente manera:

Paso 1: Selección del Juego y Definición de Objetivos.

- En este paso, se elige el juego a examinar, en este caso, *Star Forest*. Se formulan preguntas específicas que se abordarán durante las pruebas y se selecciona la audiencia objetivo para asegurar que las pruebas sean representativas y relevantes.
- Para lograr una investigación efectiva del usuario en juegos, resulta crucial establecer con claridad los objetivos deseados durante las pruebas. Es fundamental formular preguntas específicas que se enfoquen en aspectos particulares del juego, abordando elementos simples como las dificultades en los niveles, la claridad del tutorial, el atractivo visual, la sensación de juego y los problemas de usabilidad. En lugar de recurrir a preguntas genéricas como "¿Les gusta?" o "¿Es divertido?", se deben articular preguntas más detalladas, tales como "¿Qué opinan los jugadores sobre esta mecánica y por qué?" o "¿Qué hace que esta sección del nivel sea divertida para ellos y cuál es la razón?". Este enfoque investigativo, centrado en preguntas que indagan en el "por qué", adquiere una destacada relevancia cuando se aplica a nivel individual con el propósito de comprender la experiencia del usuario.

Paso 2: Planificación de la Sesión de Juego:

- El siguiente paso implica la planificación detallada de la sesión de juego. Se determina qué información se recopilará durante las pruebas, asegurándose de abordar aspectos clave de la Experiencia del Usuario, como la jugabilidad, la interfaz de usuario y la satisfacción general del jugador.

Paso 3: Sesión de Juego:

- En esta etapa, se guía a un grupo de participantes seleccionados a través del estudio del juego. Se observan y registran sus interacciones, comentarios y

reacciones para obtener una comprensión profunda de cómo los usuarios experimentan y perciben el juego. Es vital definir claramente los objetivos de las pruebas ya que se requiere un plan detallado para evitar imprevistos y garantizar la consistencia en las sesiones de juego. Esto es crucial porque, en el mundo real, la falta de preparación puede resultar costosa y afectar el tiempo y el presupuesto. Es esencial tener tareas de juego bien definidas, determinar la duración de las sesiones y establecer un flujo de trabajo para recopilar datos significativos. Se destaca la importancia de tener un guión completo y una lista de verificación para mantener la coherencia en todas las pruebas y reducir los riesgos de sesgos en el estudio. Para ello, se definen ciertas piezas que se sugiere evaluar:

- **Tareas del Juego:** Es fundamental tener claridad sobre dónde comenzarán, cuánto tiempo jugarán y qué acciones se desea que se realicen durante la prueba. Si no se comienza justo al principio del juego, es posible que aún se desee incluir el tutorial en las tareas de prueba para que se familiaricen (o recuerden) los controles.
- **Datos a recopilar:** ¿Se registrará la sesión de juego? ¿Con qué? (Al ser un juego de computadora, el proceso de poder tener un registro de la sesión de juego se hace más sencilla, ya que se cuenta con herramientas de grabación). ¿Se les pedirá que piensen en voz alta? ¿Se podrá recopilar análisis en la parte posterior? ¿Se les sugiere que hagan preguntas específicas? ¿Se les hará una entrevista al final? ¿Se les pedirá que completen una encuesta en línea de algún tipo? ¿Dónde irán todos estos datos?
- **Guion completo y lista de verificación:** Crear un guion completo y una lista de verificación es vital para todo el proceso. Esto es para verificar todo lo que se necesita tener listo antes de que los usuarios lleguen a probar el juego, lo que se hace una vez que están allí y lo que se hará después. De esta manera, se puede hacerlo de la misma manera cada vez que vayan a realizar las distintas pruebas, ya que así todos los que vayan a probar el videojuego puedan seguir este mismo proceso. Además, un procedimiento estandarizado reduce los riesgos de sesgo en el estudio.

Paso 4: Análisis y Resumen:

- Después de concluir las sesiones de juego, se procede a realizar un detallado resumen de la información recopilada. Durante este proceso, se identifican patrones, tendencias y áreas de mejora que ofrecen una valiosa perspectiva sobre los puntos fuertes y débiles del juego desde la óptica del usuario. En las pruebas de usuario de juegos, la agilidad en el manejo de resultados es fundamental debido al vertiginoso desarrollo del juego. Se requiere proporcionar un resumen conciso de los resultados clave para las preguntas de investigación, respaldados con evidencia concreta proveniente de las pruebas de usuario. La clave radica en vincular las ideas compartidas con datos tangibles de las pruebas, lo que implica combinar datos objetivos (observaciones de juego, respuestas a preguntas) con datos subjetivos (opiniones de los jugadores) para responder a las preguntas de investigación planteadas. Se aconseja la creación de presentaciones breves que destaquen preguntas clave, brinden una descripción general del plan de estudio y ofrezcan un resumen de los resultados. La práctica de explicar los resultados a colegas contribuye a perfeccionar la presentación y a priorizar de manera clara los problemas identificados.

7. Nueva versión del juego (1 semana)

3.2 Evaluación UX

Para llevar a cabo la evaluación de los diversos elementos del videojuego *Star Forest*, se toma como base el modelo de evaluación “GUESS-24” creado por “Phan et al.”.

Como se mencionó anteriormente, el modelo propone preguntas que evalúan la satisfacción general de un videojuego, compuesto (en el caso de *Star Forest*) por la diversión que brinda el juego, la narrativa, la estética visual, estética sonora, gratificación personal y jugabilidad.

Dicha evaluación contiene las siguientes preguntas:

Movimiento Básico	
Pregunta	Alternativas
¿La respuesta de los controles es precisa y se siente fluida?	Si / No
¿La respuesta del salto se siente fluida y cómoda?	Si / No
¿La gravedad del salto se siente cómoda?	Si / No
¿La velocidad del personaje se siente correcta?	Si / No
Mecánicas del videojuego	
¿La respuesta del salto doble se siente fluida?	Si / No
¿La respuesta del salto coyote se siente correcta?	Si / No
¿La respuesta del impulso o dash se siente correcta?	Si / No
¿La fuerza del salto doble es correcta o necesita más altura?	Es correcta / Necesita más altura
Interfaz, Diseño de Niveles y Sonidos del Juego	
¿El juego tiene una interfaz clara y fácil de entender?	Si / No
¿Los elementos visuales del juego son	Si / No

atractivos y coherentes con el estilo artístico?	
¿El tutorial o instrucciones del juego son claros y te ayudan a comprender las mecánicas?	Si / No
¿El sistema de guardado o checkpoints es justo y te permite avanzar sin frustración excesiva?	Si / No
¿El juego ofrece una cantidad correcta en términos de enemigos y obstáculos en los niveles?	Si / No
¿La música y los efectos de sonido contribuyen a la atmósfera del juego?	Si / No
¿La música y los efectos de sonido tienen un volumen adecuado?	Si / No

Capítulo 4: Desarrollo Star Forest en Unity

4.1 Introducción a Unity

De acuerdo a la definición propuesta por Ebac:

“Unity es un motor gráfico multiplataforma 2D y 3D, que está transformando la industria del videojuego, pertenece a Unity Technologies y sirve para la creación de videojuegos.

Puedes desarrollar juegos para varios dispositivos sin cambiar de plataforma. Aunque cabe mencionar que también se usa para agregar experiencias con realidad virtual, en el cine, la animación, y sus funciones favorecen sectores como el sector salud, automotriz, el de la construcción, entre otros.” [30]

Unity es un motor de videojuegos flexible y potente que permite a los desarrolladores crear juegos en una amplia variedad de plataformas. Con Unity, los desarrolladores pueden crear juegos 2D y 3D, lo que lo convierte en una opción popular para juegos de plataformas.

Uno de los mayores beneficios de Unity es su facilidad de uso y accesibilidad. La plataforma viene con una amplia variedad de herramientas y recursos que ayudan a los desarrolladores a crear juegos de alta calidad sin tener que escribir código desde cero. Esto significa que incluso si eres un desarrollador principiante, puedes empezar a trabajar en un proyecto sin ningún problema.

Unity también es altamente personalizable. Los desarrolladores pueden agregar sus propios códigos, componentes y recursos para crear un juego único y personalizado.

Entonces, una vez ya con nuestros recursos a mano (avatares, arte, música, efectos de sonido) y el software de Unity instalado en la máquina con todos sus requisitos necesarios, empezamos a desarrollar el videojuego *Star Forest*.

4.2 Pantalla Principal de Unity

Antes de comenzar a trabajar en el proyecto, lo primero que se nos presenta en Unity es una interfaz con distintas secciones que pueden personalizarse al gusto del usuario. Lo que usualmente se ve es la pantalla de **Escenario**, **Jerarquía**, **Proyecto**, **Juego** y finalmente **Inspector**. Vayamos una por una sobre qué hace cada sección.

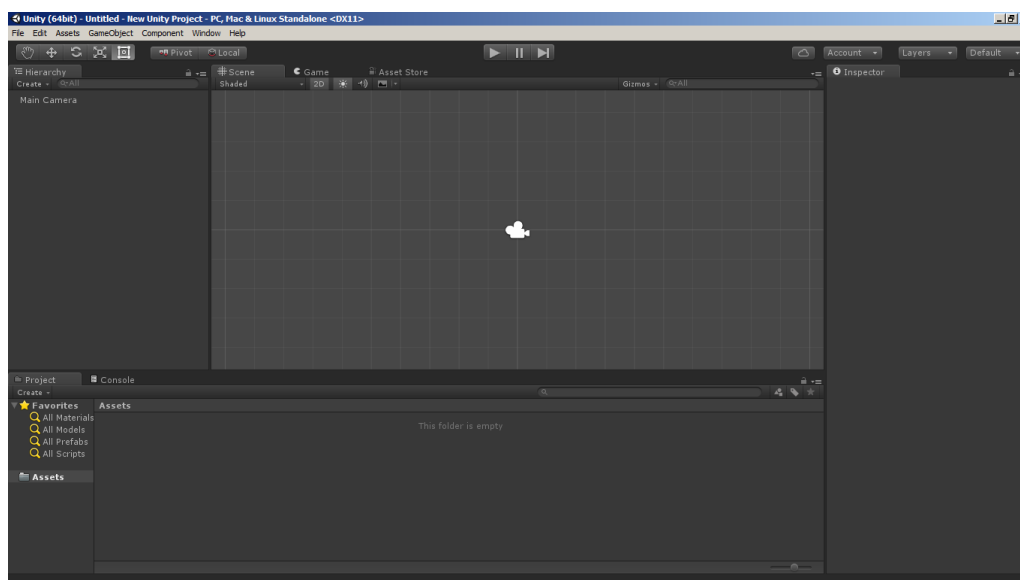


Ilustración 43: Nuevo proyecto en Unity

- **Proyecto:** Es la sección en donde los desarrolladores pueden acceder y administrar todos los recursos y archivos que se utilizan en el proyecto; se pueden agregar, eliminar, importar y organizar todos los archivos del juego, como sprites, materiales, sonidos, código y escenas. También cuenta con la función de crear carpetas para organizar cada contenido importado.

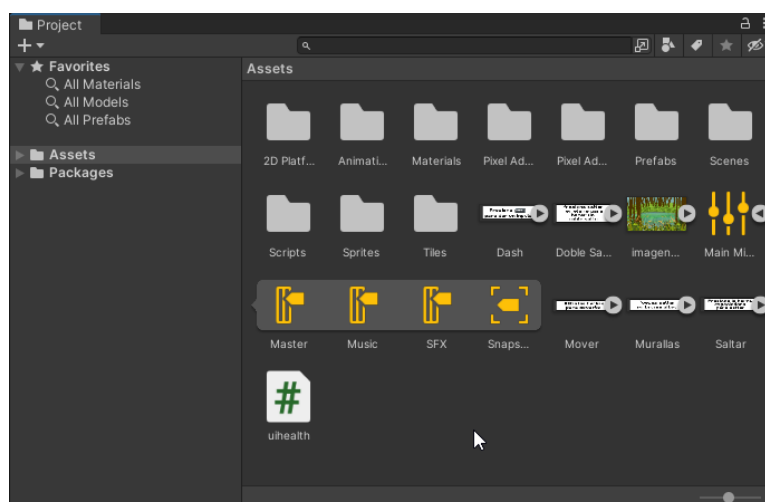


Ilustración 44: Ventana de Proyecto en Unity

- **Jerarquía:** Es donde los desarrolladores pueden visualizar y manipular todos los elementos que se encuentran en la escena del juego. Se pueden ver todos los objetos, como personajes, enemigos, obstáculos y otros elementos, y organizarlos en una estructura jerárquica. Utiliza una estructura de árbol, pudiéndose arrastrar y soltar en diferentes posiciones de la jerarquía. En resumen, cada objeto que es visible o se encuentre presente será añadido a la sección de Jerarquía.

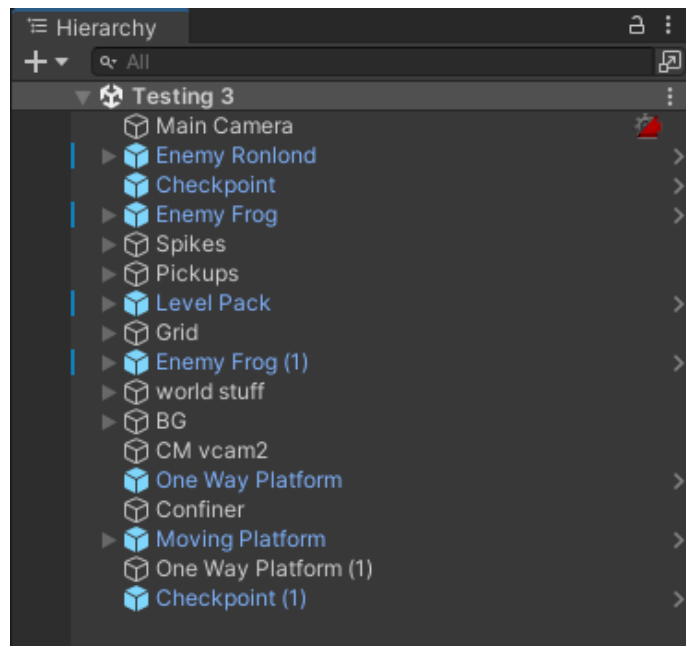


Ilustración 45: Ventana de Jerarquía en Unity

- **Juego:** Representa lo que veremos en el producto final una vez añadido todos los recursos y códigos, básicamente nos permite el juego en tiempo real y probar las funcionalidades del mismo mientras se va construyendo. En la sección de Juego se prueban cosas como la velocidad del personaje, la función de atacar, la función de saltar del personaje, ver si los niveles tienen un buen flujo, entre muchos otros.

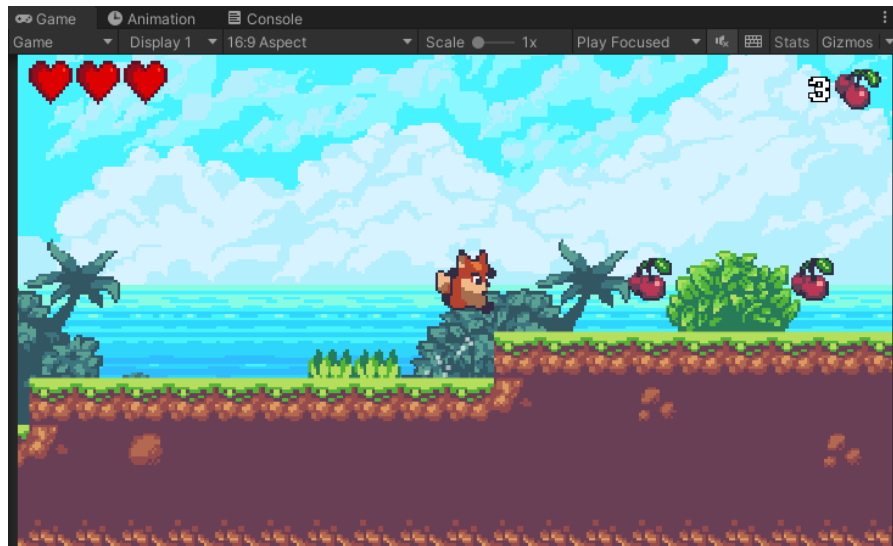


Ilustración 46: Ventana de Juego en Unity

- **Inspector:** Corresponde al área donde se visualiza y se modifican las propiedades de los componentes de los objetos de la escena y, al mismo tiempo, añadir componentes. La sección de inspector está dividida en dos áreas principales: La ventana de detalles y la barra de herramientas. En la de detalles, se pueden ver y modificar propiedades de un objeto que se encuentra en el juego, estos incluyen modificar la posición, rotación, escala, códigos, colisiones o renderizadores de un objeto. La barra de herramientas proporciona acceso a varias funciones, como la creación de nuevos componentes y la búsqueda de objetos específicos. Un ejemplo del uso del Inspector es que ayuda a modificar la velocidad de movimiento de un personaje, sea el jugador o un enemigo. Otro ejemplo es el daño que inflige un obstáculo.

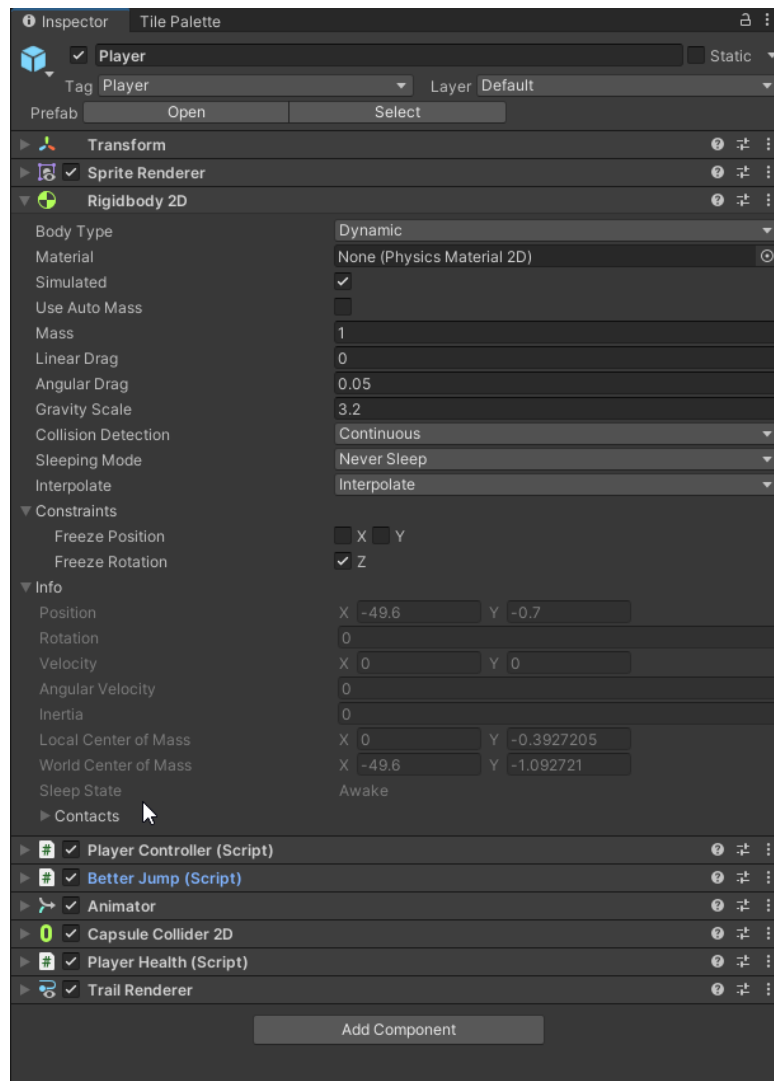


Ilustración 47: Ventana de Inspector en Unity

A medida que se va desarrollando el videojuego, la cantidad de secciones se irán sumando, cada una con distintas funciones, como lo es la sección de Animaciones, la sección de Animador, la sección de Mixer de Música, entre muchos otros, pero se ven en más detalle más adelante en este capítulo.

La cantidad de recursos que se utilizan en Unity es variada y siempre se están añadiendo más, por lo que se recomienda crear diferentes carpetas para importar los recursos del videojuego, cada una con un nombre respectivo, para mantener una organización general de todos los recursos utilizados en el proyecto. El número de carpetas dependerá de la cantidad de recursos que se agreguen.

No existe una regla estricta para las carpetas que deben crearse, pero sí existen algunas de las carpetas comunes que los desarrolladores de juegos suelen crear en sus proyectos de plataformas 2D de Unity:

- **Assets:** Esta carpeta se crea de forma predeterminada y contiene todos los activos utilizados en el juego, como texturas, sprites o archivos de audio (al ser pocos recursos, usualmente se crean las carpetas "Sprites", "Audio", entre otros).
- **Scenes:** Contiene todas las escenas del juego.
- **Scripts:** Contiene todos los códigos en lenguaje C# utilizados en el juego. Estos códigos contienen la lógica que controla el comportamiento de los objetos del juego. Por ejemplo, el movimiento del jugador, el comportamiento de un enemigo, la interacción entre un coleccionable y un jugador, etc.
- **Prefabs:** Contiene todos los prefabricados utilizados o a utilizar en el juego. Un prefab es un objeto que se define, configura y puede ser instanciado en escenas de manera consistente. Los prefabs permiten crear objetos complejos con configuraciones específicas, y luego reutilizar esos objetos fácilmente en diferentes partes de un proyecto.
- **Animaciones:** Contiene todos los archivos de animaciones utilizadas en el juego.
- **Fonts:** Contiene todas las fuentes utilizadas en el juego para la representación del texto.
- **Audio:** Esta carpeta contiene todos los archivos de audio utilizados en el juego, como efectos de sonido y música.

Vale la pena señalar que la organización de carpetas y archivos en un proyecto de Unity puede variar según la complejidad del juego, las preferencias del desarrollador y los flujos de trabajo del equipo de desarrollo.

4.3 Los objetos de juego (gameObject)

Un "gameObject" es un objeto que se utiliza para representar cualquier elemento de una escena de juego, como personajes, enemigos, objetos, interfaces, etc. Se puede pensar como un contenedor que puede tener varios componentes que definen su comportamiento y aspecto en el juego. Por ejemplo, un personaje puede ser un gameObject con componentes como un sprite, un controlador de movimiento, un controlador de animación, y otros componentes necesarios para definir su comportamiento en el juego.



Ilustración 48: Game Objects en Unity

Son creados y añadidos en la sección de escena o en la sección de jerarquía de Unity. En la sección de jerarquía, los gameObject se organizan en una jerarquía de objetos, lo que permite una mejor gestión y organización de la escena. Del mismo modo, un gameObject sirve también para agrupar otros gameObject que pertenecen a la misma categoría. Por ejemplo, sonidos, fondos, obstáculos, plataformas y así sucesivamente.

4.4 Tilemap Editor (Editor de mapas de mosaico)

Tilemap Editor en Unity es una herramienta que permite crear y editar mapas 2D. Proporciona una interfaz intuitiva para colocar, escalar y rotar mosaicos, así como herramientas para pintar, borrar y crear pinceles de mosaico.

En un proyecto de plataformas 2D de Unity, el Editor de mapas de mosaicos se puede usar para crear los niveles del juego mosaicos que representan diferentes tipos de terreno, como plataformas, paredes, pisos y obstáculos. Estos mosaicos se pueden organizar para crear niveles complejos con múltiples capas, desplazamiento de paralaje y otros efectos.

Para poder usar Tilemap Editor en Unity, se requiere ya tener creada una paleta de mosaico que se puede importar a Unity para que ésta pueda ser utilizada.

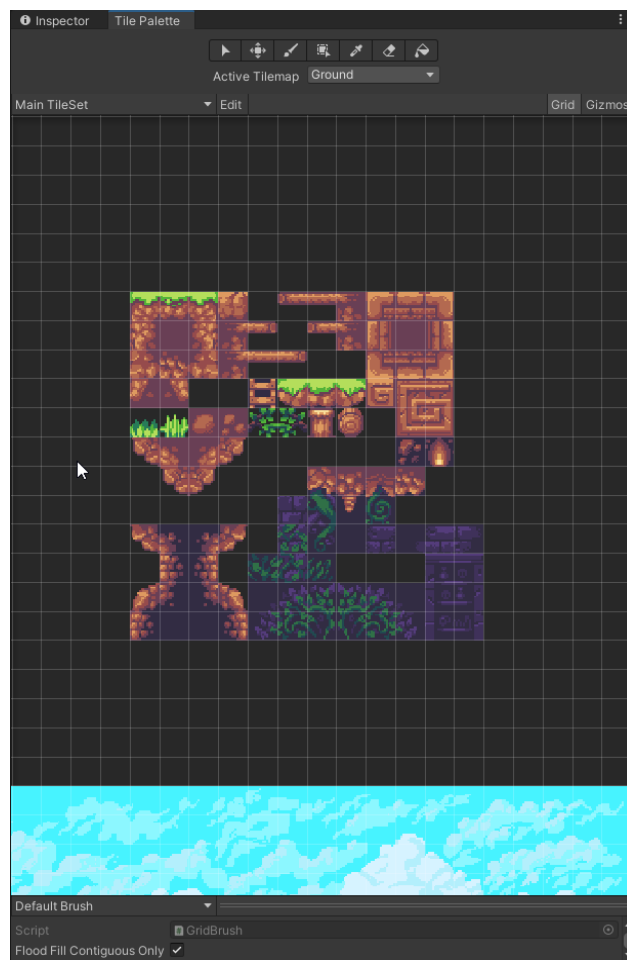


Ilustración 49: Tilemap Editor en Unity

Para poder hacer pruebas de movimiento del jugador, objetos, enemigos, etc., se requiere primero tener armado un nivel para llevar a cabo las distintas pruebas, por lo que se crea un objeto de tipo “Tilemap” en Unity.

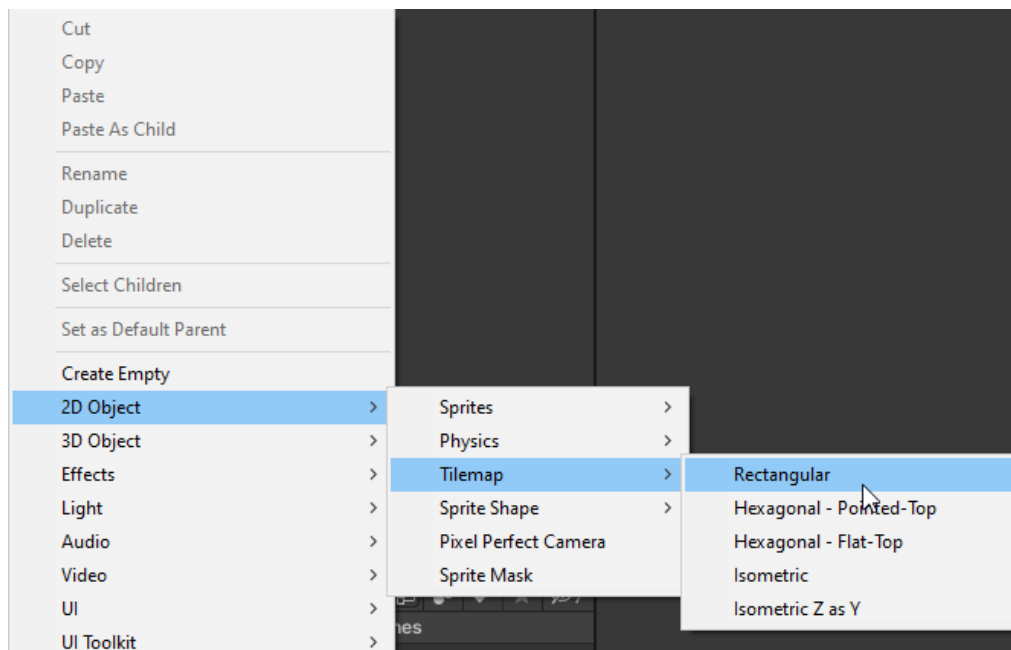


Ilustración 50: Creando un Game Object de tipo Tilemap

Esto creará un objeto en nuestra escena que nos permite crear un nivel con los componentes necesarios. Y una vez importamos nuestra paleta de mosaicos, se comienza a pintar el nivel.



Ilustración 51: Seleccionando Tiles en Unity

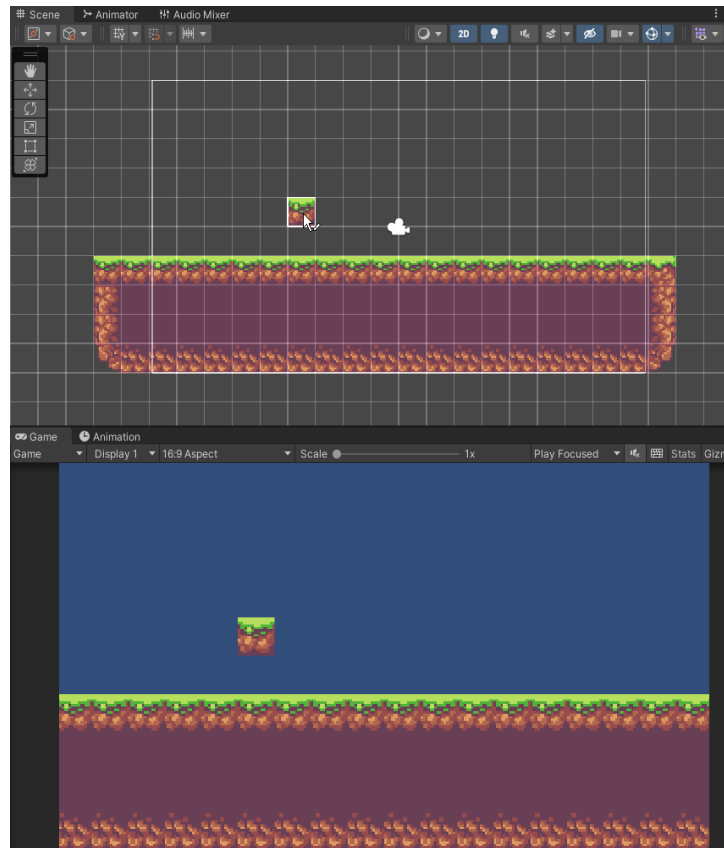


Ilustración 52: Pintando en la escena

Una vez creada la escena, es necesario proporcionar una caja de colisiones al mosaico presente para permitir al jugador caminar sobre él. De lo contrario, durante las pruebas, el jugador atravesaría la plataforma, cuando en realidad queremos que pueda caminar y estar sobre ella.

En primer lugar, se agrega un componente "Rigidbody 2D" que le da la capacidad al mosaico de interactuar con otros elementos mediante físicas. Sin embargo, si lo dejamos así, todo el nivel se derrumbaría debido a la gravedad. Para solucionar esto, vamos al inspector en el editor de Unity y le asignamos el valor de "Kinematic", lo que mantendrá la plataforma estática.

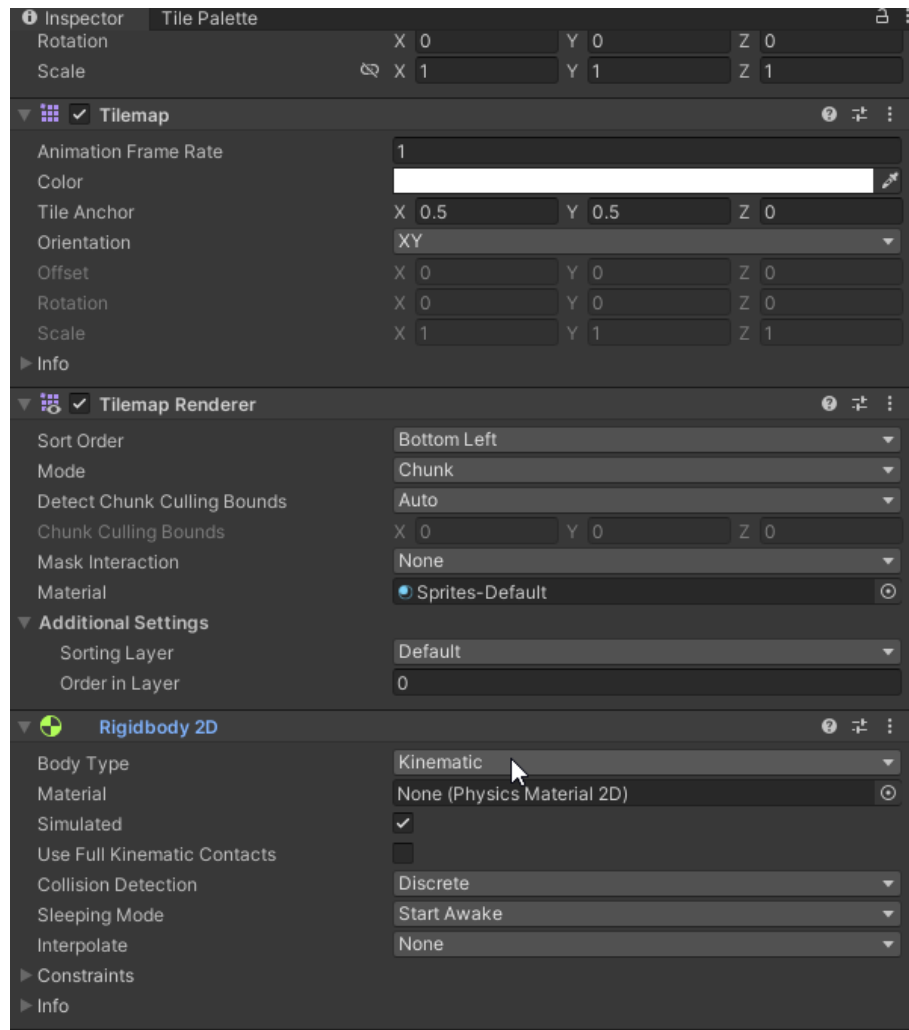


Ilustración 53: Componente RigidBody 2D

Luego, se añade un colisionador de mosaico 2D (Tilemap Collider 2D) para que el nivel pueda interactuar con cualquier elemento que colisione con él. Aunque hemos configurado lo necesario para que el personaje pueda caminar sobre la plataforma, existe una futura complicación que podemos resolver de inmediato.

El Tilemap Collider 2D proporciona una caja de colisiones a cada elemento de la plataforma, lo que podría convertirse en un problema al seguir pintando el nivel con más plataformas mediante el Tilemap Editor. Esto podría consumir demasiados recursos del hardware, ya sea en una computadora, tableta o teléfono celular.

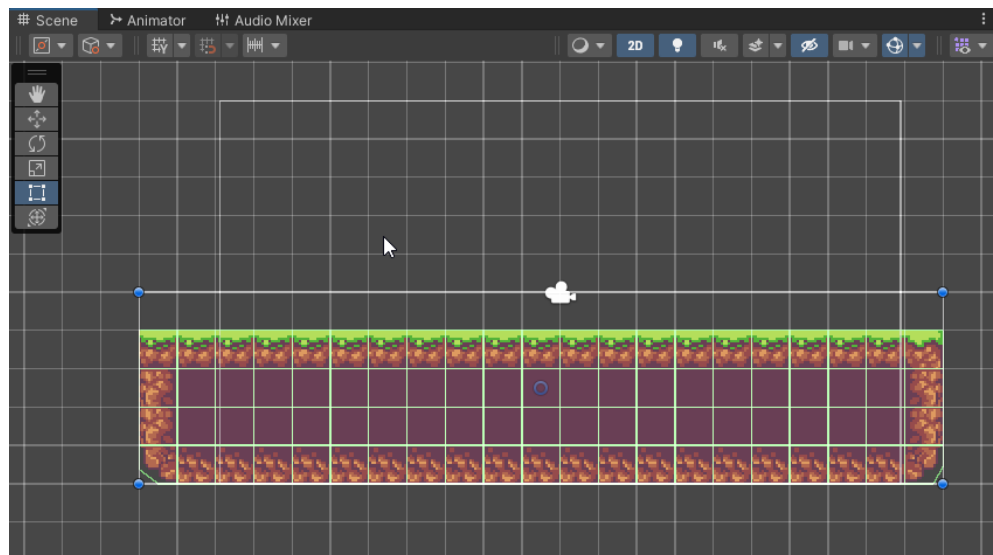


Ilustración 54: Componente Tilemap Collider

Por lo que, para solucionar este problema, se añade un tercer componente llamado “Composite Collider 2D”, que analiza el tipo de plataforma que Tilemap Collider 2D ha construido, y lo simplifica, dando una mejor optimización para el videojuego.

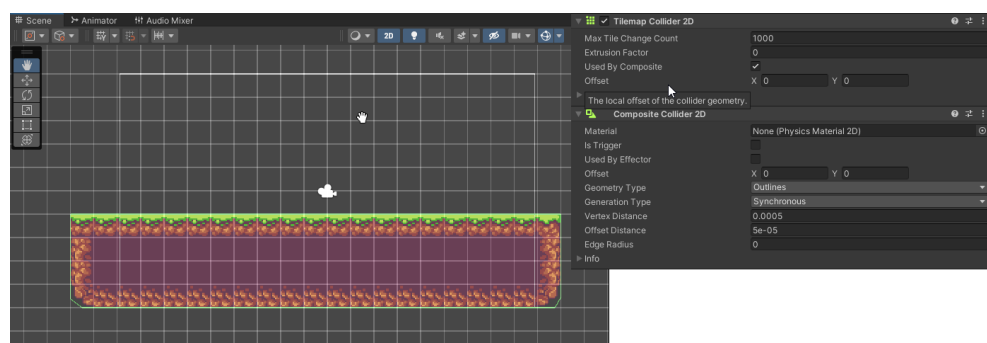


Ilustración 55: Composite Collider 2D en Acción

Con el pintado de plataformas y creación de niveles, pasamos a la siguiente sección para poder probar y establecer las bases del videojuego: El jugador.

4.5 El jugador

En el contexto de un proyecto de Unity, un "jugador" se refiere al personaje o entidad controlada por el usuario dentro del videojuego. Es el avatar virtual a través del cual el jugador interactúa con el mundo del juego y realiza acciones.

El jugador es responsable de tomar decisiones y ejecutar acciones en respuesta a los estímulos del juego, ya sea moviéndose, saltando, atacando, recolectando objetos, interactuando con el entorno o con otros personajes, entre muchas otras posibilidades.

En Unity, el jugador se implementa mediante el uso de scripts y componentes que controlan su movimiento, interacciones y comportamiento en el juego. Esto puede incluir la detección de colisiones, animaciones, control de cámara, manejo de entrada del usuario (como teclado, mouse o controles de consola), entre otros aspectos que permiten la interacción del jugador con el mundo del juego.

4.5.1 Configuración básica

Se crea al jugador arrastrando una imagen a la escena, donde Unity agrega automáticamente el componente "Sprite Renderer" para la visibilidad de éste. Se añade los componentes "Rigidbody 2D" y un "Circle Collider 2D" para habilitar las físicas y la detección de colisiones, cada una ajustable a su medida. La cápsula de colisiones, además de poder interactuar con otros objetos del videojuego (obstáculos, coleccionables, enemigos) también evita a que el jugador atravesase plataformas.

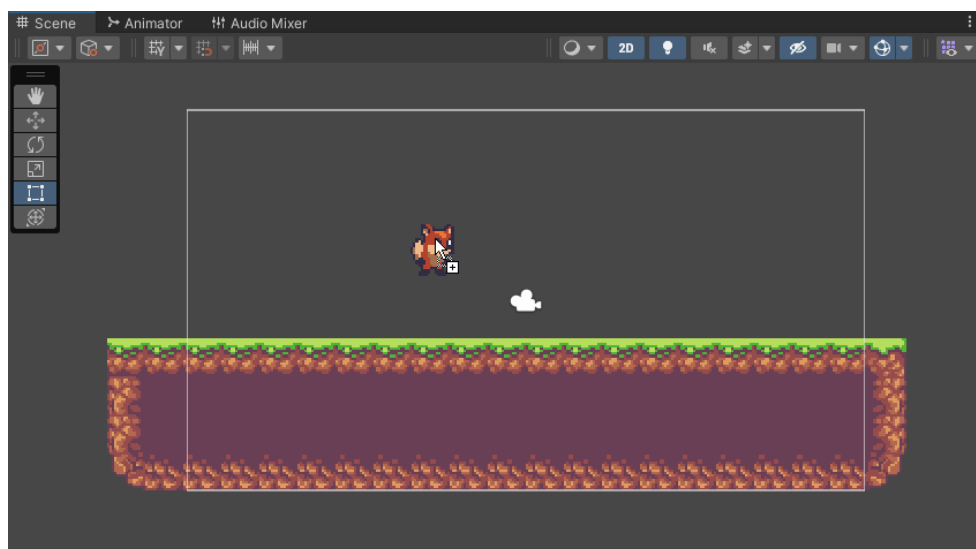


Ilustración 56: Añadiendo al Personaje Jugable en la ventana de Escena

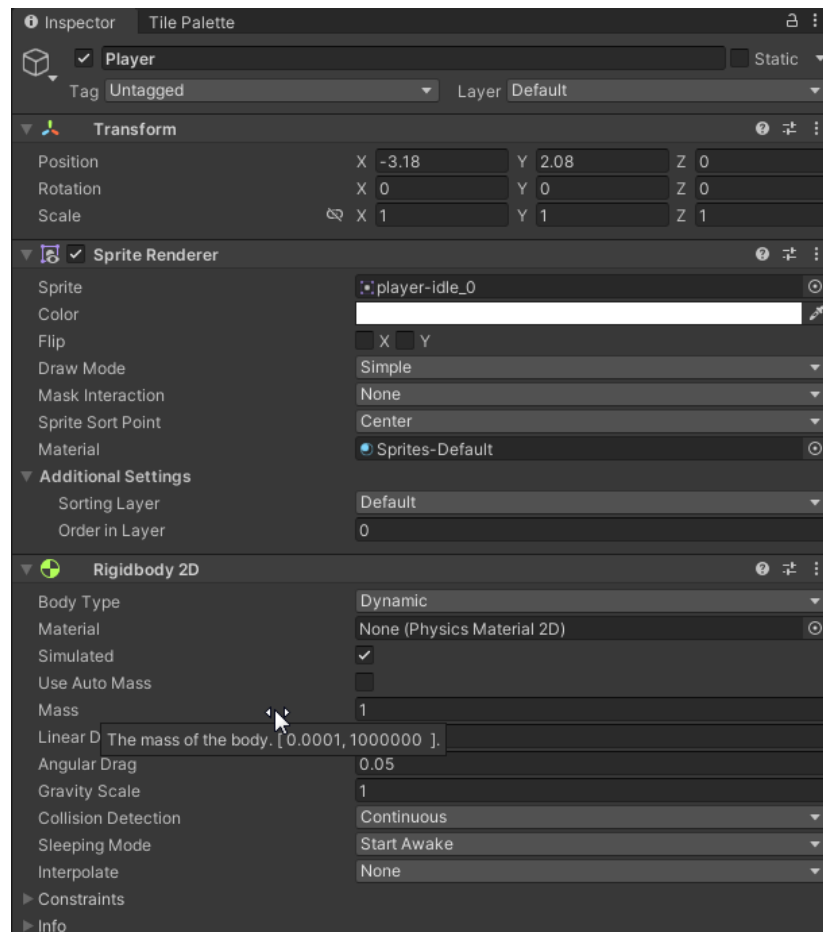


Ilustración 57: Configurando el Rigidbody 2D del Personaje

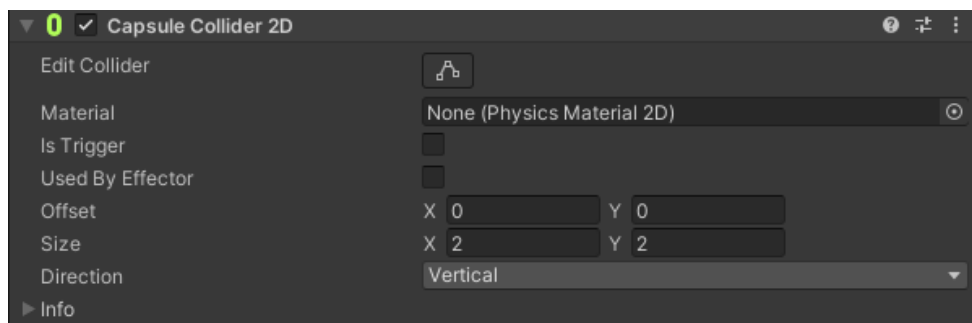


Ilustración 58: Caja de Colisiones del Personaje 1

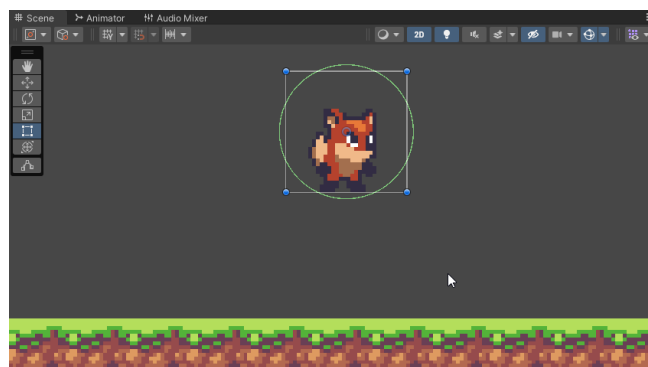


Ilustración 59: Caja de Colisiones del Personaje 2

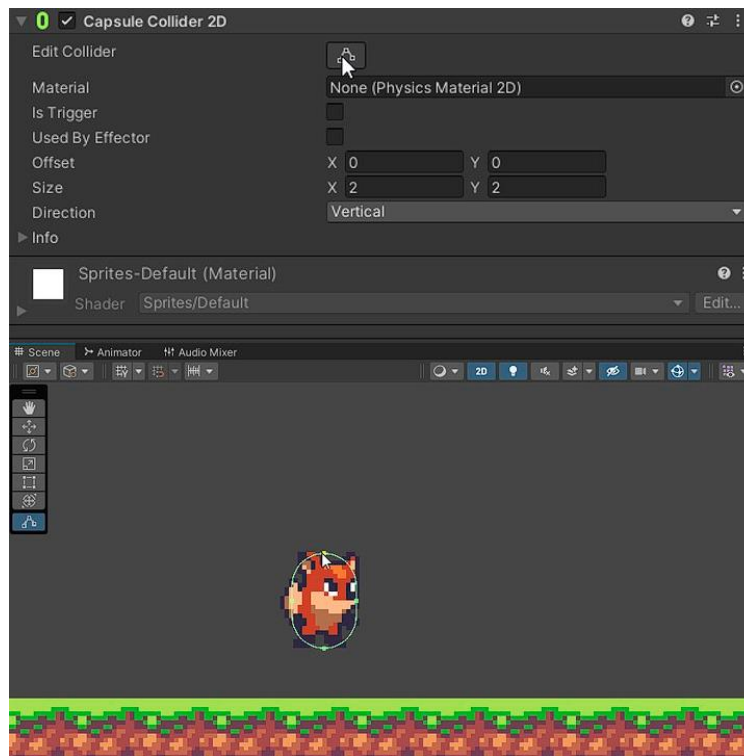


Ilustración 60: Configurando la Caja de Colisiones

4.5.2 Movimiento básico

Se crea el script "PlayerController" para manejar el movimiento general del jugador, desde su movimiento horizontal por el juego hasta el salto; este script también incluye la forma en que el jugador utiliza las mecánicas como Dash, Jump Input Buffering y Coyote Time.

Se implementa el script con métodos de Unity como Start() y Update(), utilizando el componente "Rigidbody 2D" para el control del movimiento del jugador.

- Start(): Este método se ejecuta una vez en el momento en que inicia el juego y activa las líneas de código que contiene.
- Update(): Esta función se ejecuta constantemente en cada fotograma del juego y activa las líneas de código que contiene.

```

private Rigidbody2D rb;

public float speed = 8f;
private float dirX;

// Start is called before the first frame update
Mensaje de Unity | 0 referencias
void Start()
{
    rb = GetComponent<Rigidbody2D>();
}

```

Ilustración 61: Código inicial del script Player Controller

```

// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);
}

```

Ilustración 62: Código de movimiento del Jugador

4.5.3 El Salto

Se crea el script "BetterJump" para mejorar el comportamiento del salto ajustando la gravedad y la velocidad vertical, utilizando una variable para ajustar la potencia del salto, mejorando la caída y el comportamiento del salto.

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

Script de Unity (1 referencia de recurso) | 0 referencias
public class BetterJump : MonoBehaviour
{
    public float fallMultiplier = 2.5f;
    public float lowJumpMultiplier = 2f;

    private Rigidbody2D rb;

    Mensaje de Unity | 0 referencias
    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    Mensaje de Unity | 0 referencias
    private void Update()
    {
        if (rb.velocity.y < 0)
        {
            rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) * Time.deltaTime;
        }
        else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))
        {
            rb.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier - 1) * Time.deltaTime;
        }
    }
}
```

Ilustración 63: Código del script Better Jump

Este script, en conjunto con el del "Player Controller", mejora el comportamiento del salto del personaje en Unity al modificar la gravedad y la velocidad vertical en función de si está cayendo o subiendo en un salto. Esto proporciona un salto más realista y controlado en el juego.

4.5.3 Evitando saltos infinitos

Se añade una función llamada "IsGrounded" para verificar si el jugador está en contacto con el suelo, y para que surta efecto se crea un objeto de juego que detecte el suelo, posicionado a los pies del personaje y con su caja de colisiones respectiva, y se evitan saltos infinitos verificando si este se encuentra tocando el suelo o no.

```
{
    [SerializeField] private Rigidbody2D rb;
    [SerializeField] private Transform groundCheck;
    [SerializeField] private LayerMask groundLayer;

    public float speed = 8f;
    private float dirX;

    [Range(1, 10)] public float jumpVelocity;

    // Start is called before the first frame update
    // Mensaje de Unity | 0 referencias
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    // Mensaje de Unity | 0 referencias
    void Update()
    {
        dirX = Input.GetAxisRaw("Horizontal");
        rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

        if (Input.GetButtonDown("Jump") && IsGrounded())
        {
            rb.velocity = Vector2.up * jumpVelocity;
        }
    }

    // 1 referencia
    private bool IsGrounded()
    {
        return Physics2D.OverlapCircle(groundCheck.position, 0.2f, groundLayer);
    }
}
```

Ilustración 64: Verificaciones para evitar el salto infinito

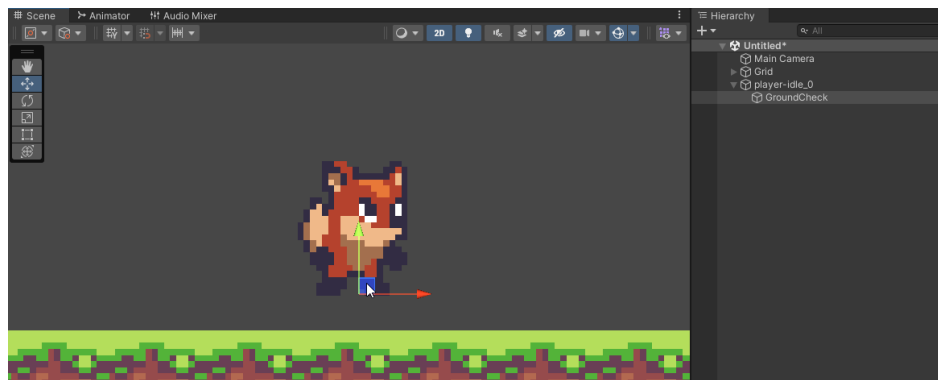


Ilustración 65: Creando el Game Object "Ground Check"

Con esto, el jugador saltará solo cuando el botón de salto es presionado y solo cuando se encuentre tocando una plataforma, evitando así que este salte de manera infinita.

4.5.4 Volteando al jugador

Se implementa una función "Flip" en el script "Player Controller" para voltear al jugador horizontalmente dependiendo de la dirección que se encuentre observando. Utilizan los valores de escala, donde si el valor de la escala es 1, el avatar del personaje se encontrará mirando a la derecha (en este caso), y si se cambia a -1, voltea hacia el otro lado.

```
1 referencia
private void Flip()
{
    if (isFacingRight && dirX < 0f || !isFacingRight && dirX > 0f)
    {
        isFacingRight = !isFacingRight;
        Vector3 localScale = transform.localScale;
        localScale.x *= -1f;
        transform.localScale = localScale;
    }
}
```

Ilustración 66: Función Flip en el script Player Controller

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

    if (Input.GetButtonDown("Jump") && IsGrounded())
    {
        rb.velocity = Vector2.up * jumpVelocity;
    }

    Flip();
}
```

Ilustración 67: Llamando la función en el método Update

De esta forma, el personaje alterna su dirección horizontal de manera visual. Esto resulta especialmente útil al implementar animaciones o para reflejar el movimiento del personaje en la representación gráfica.

4.5.5 Evitando quedarse atascado en las murallas

Se crea un material llamado "No Friction" (ver ilustración 68) que permite evitar que el jugador se quede atascado en las paredes o en los bordes de las plataformas, donde se asigna dicho material al componente "Rigidbody 2D" del jugador.

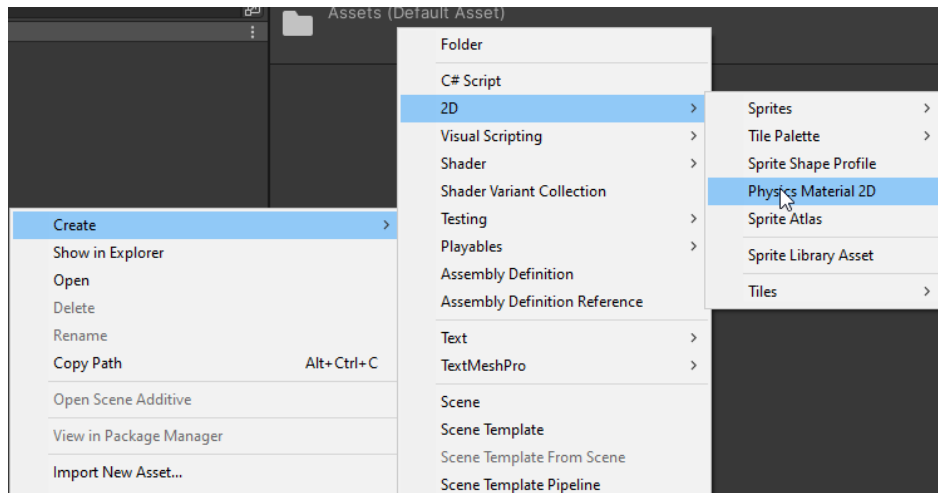


Ilustración 68: Creando el material No Friction

A este material le modificamos los valores en el inspector, asegurándonos que tenga sus valores de fricción y rebote en 0.

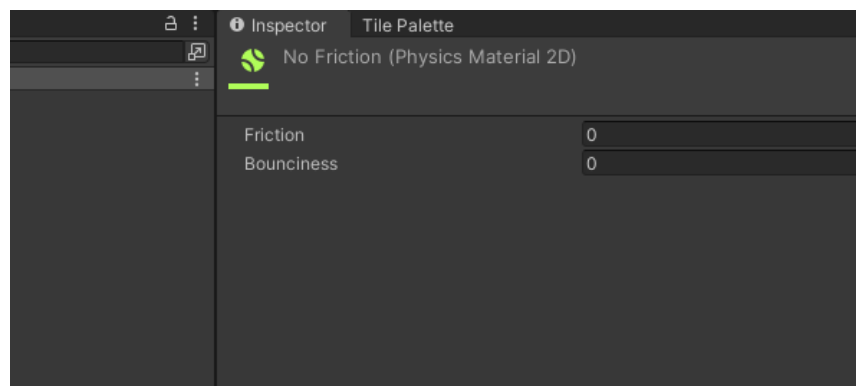


Ilustración 69: Asignando valores para el material No Friction

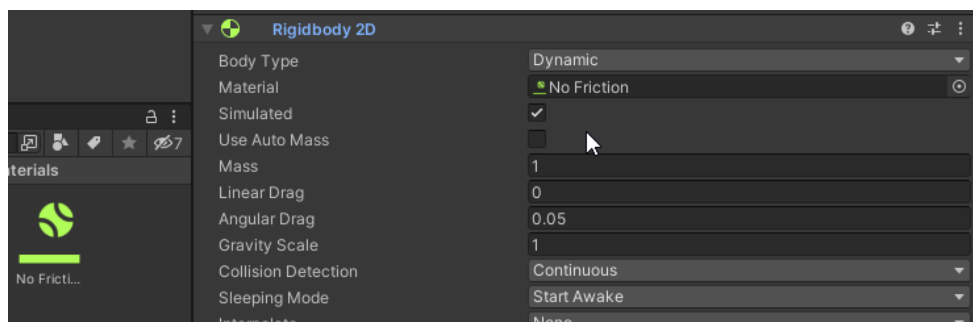


Ilustración 70: Asignando el material en el RigidBody 2D del Jugador

Con esto, el personaje ya no se quedará atascado en las murallas o en los bordes de las plataformas al llegar a colisionar con ellas.

4.5.6 Animando al personaje

Se crean animaciones para el personaje, cada una siendo una animación para correr, de saltar, de ser dañado, de estar quieto, entre otras. Cada objeto de animación se le da un nombre y se asignan a través de la ventana de “Animation”.

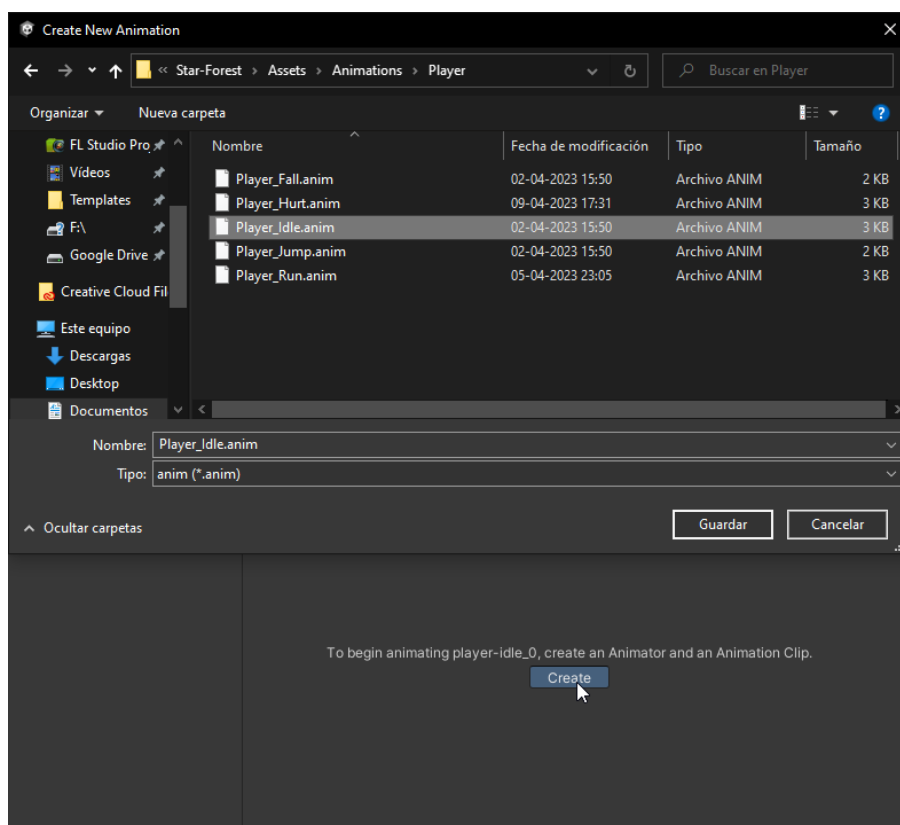


Ilustración 71: Creando distintas animaciones para el Jugador

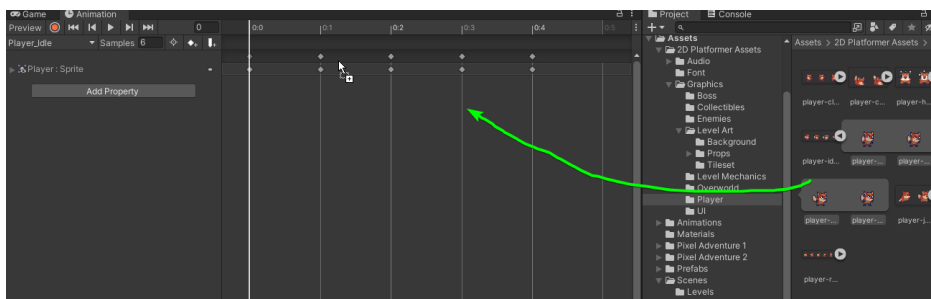


Ilustración 72: Añadiendo sprites o avatares para la animación "Idle"

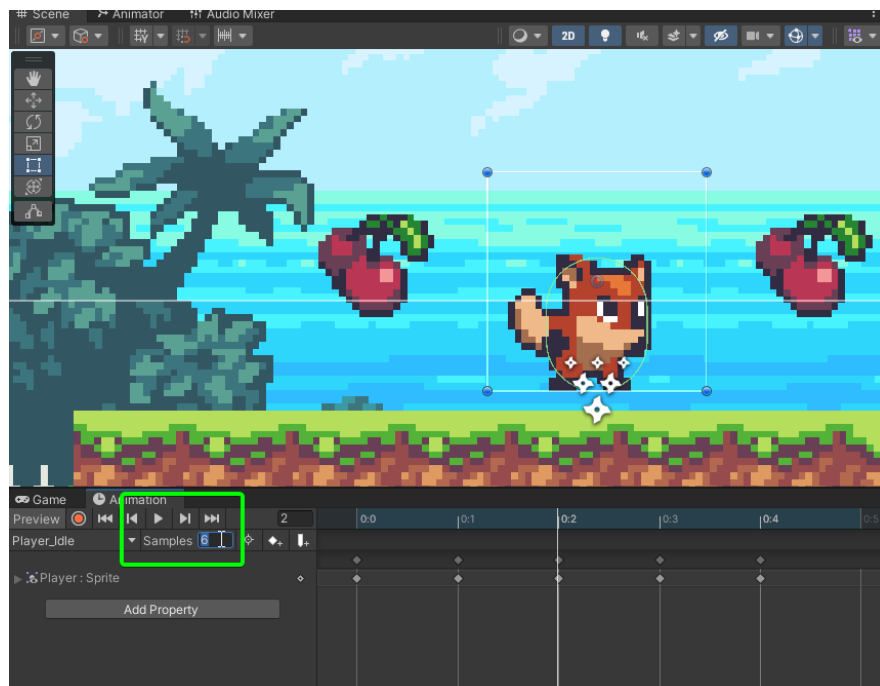


Ilustración 73: Configurando la animación "Idle"

Este proceso se repite para las animaciones de salto, de caída, de correr y de ser lastimado, cada una con visuales distintas.

4.5.7 Cambiando de animaciones

Se crean condiciones en el Animator para cambiar entre animaciones (ver ilustración 74), utilizando variables y transiciones en el código para activar las condiciones y cambiar las animaciones según el estado del jugador.

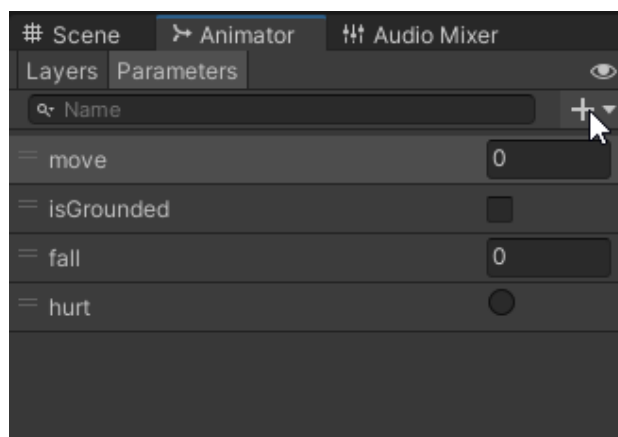


Ilustración 74: Creando condiciones en la ventana Animator

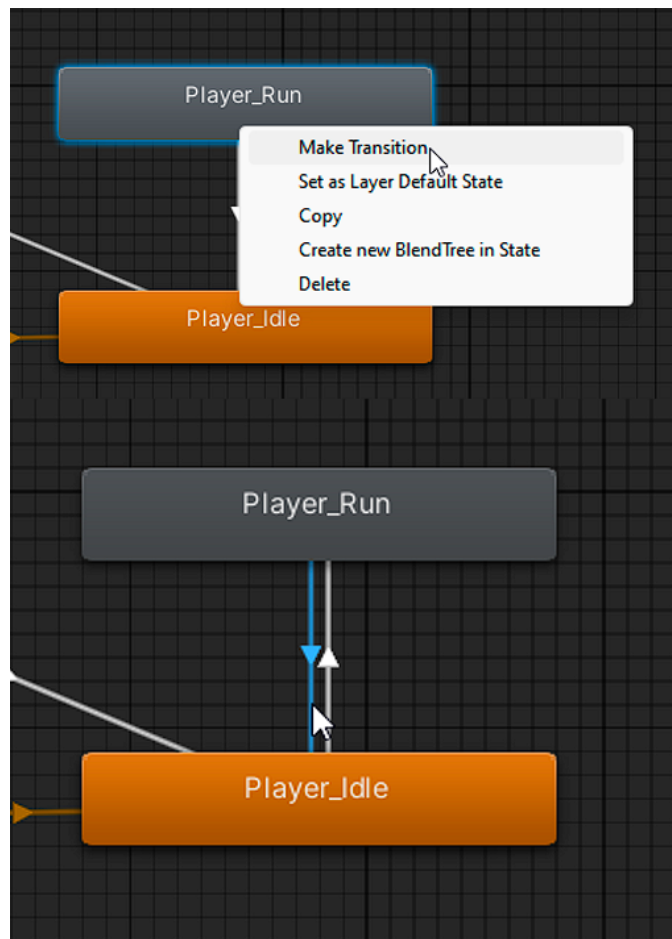


Ilustración 75: Creando transiciones entre animaciones

Cada animación es llamada en el método "Update()", activando el parámetro correspondiente de las animaciones creadas en la ventana "Animator".

```
private Animator anim;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    anim = GetComponent<Animator>();
}

// Update is called once per frame
void Update()
{
    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

    if (Input.GetButtonDown("Jump") && IsGrounded())
    {
        rb.velocity = Vector2.up * jumpVelocity;
    }

    anim.SetFloat("move", Mathf.Abs(rb.velocity.x));
}
```

Ilustración 76: Cambiando animaciones en el script "Player Controller"

4.6 Mecánicas de salto y movimiento

Con el movimiento básico del personaje finalizado, lo único que queda es implementar las mecánicas que ayudan a que el control del personaje se sienta fluido y que no se encuentre ningún defecto. Como se mencionó anteriormente en capítulos anteriores, el principal objetivo es buscar que el jugador sienta total control sobre el personaje. Se procede a implementar las distintas mecánicas de Doble Salto, Wall Jump, Coyote Time y Jump Buffering.

4.6.1 Doble Salto

En el script “Player Controller”, se agregan dos variables, una que verifique si el doble salto está en curso y otra que defina la altura del doble salto. Luego se añade una condición adicional al botón de salto para permitir que la mecánica del doble salto se ejecute, y se ajusta la potencia de éste. Luego se invierte el estado de la variable que comprueba si el doble salto está en curso (“doubleJump”), esto es para evitar que el jugador de saltos infinitos.

```
void Update()
{
    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

    if (IsGrounded() && !Input.GetButton("Jump"))
    {
        doubleJump = false;
    }

    if (Input.GetButtonDown("Jump"))
    {
        if (doubleJump || IsGrounded())
        {
            rb.velocity = new Vector2(rb.velocity.x, doubleJump ? doubleJumpForce : jumpVelocity);
            doubleJump = !doubleJump;
        }
    }
}
```

Ilustración 77: Código de mecánica Doble Salto

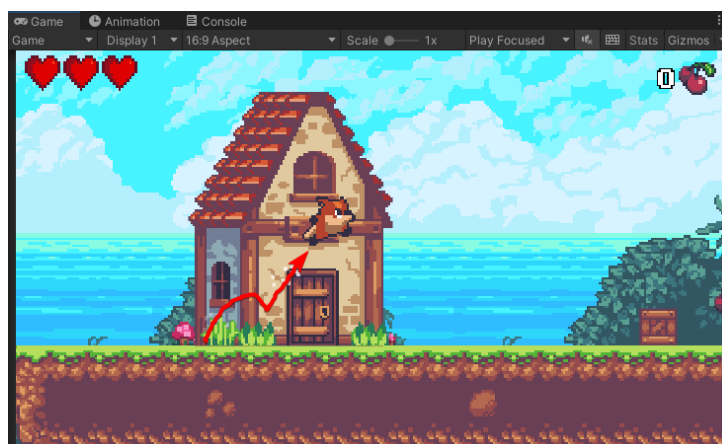


Ilustración 78: Doble Salto en Acción

4.6.2 Coyote Time y Jump Buffering

Para mejorar la sensación de control del personaje en el juego, se utilizan las mecánicas de "Coyote Time" y "Jump Buffering" (Tiempo de Coyote y Buffer de Salto). Estas mecánicas brindan una mayor capacidad de control al realizar saltos entre diferentes plataformas y obstáculos en el juego.

"Coyote Time" permite que el jugador realice un salto incluso después de que el personaje haya abandonado una plataforma. Por otro lado, "Jump Buffering" permite que el juego registre saltos antes de que el personaje toque el suelo.

Para implementar el "Coyote Time", se utiliza una variable para determinar el tiempo que tiene el jugador luego de que abandona una plataforma ("coyoteTime"). Luego se utiliza una variable que se utiliza como contador ("coyoteTimeCounter") para rastrear este tiempo, disminuyendo gradualmente cuando el jugador no está en contacto con el suelo. La condición de salto ahora verifica el contador en lugar de su anterior condición que revisaba si el jugador se encontraba en contacto con el suelo.

Y para implementar la mecánica de Jump Buffering (Buffer de Salto), se agregan dos variables para permitir que el juego registre los saltos que realiza el jugador antes de tocar el suelo. Cuando se presiona el botón de salto, la variable "jumpBufferCounter" se establece en la duración de "jumpBufferTime". Si no se presiona, disminuye gradualmente.

```
void Update()
{
    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

    // Jump Buffering
    if (Input.GetButtonDown("Jump"))
    {
        jumpBufferCounter = jumpBufferTime;
    }
    else
    {
        jumpBufferCounter -= Time.deltaTime;
    }

    // Coyote Time
    if (IsGrounded())
    {
        coyoteTimeCounter = coyoteTime;
    }
    else
    {
        coyoteTimeCounter -= Time.deltaTime;
    }

    if (Input.GetButtonDown("Jump") && coyoteTimeCounter > 0f)
    {
        rb.velocity = Vector2.up * jumpVelocity;
        jumpBufferCounter = 0f;
    }

    if (Input.GetButtonUp("Jump") && rb.velocity.y > 0f)
    {
        coyoteTimeCounter = 0f;
    }
}
```

Ilustración 79: Implementación de Jump Buffering y Coyote Time en el script "Player Controller"



Ilustración 80: Coyote Time en Acción

4.6.2 Dash

Para implementar la mecánica del Dash (o Impulso), se emplea una función especial llamada "IEnumerator". Estas funciones son conocidas como corrutinas y resultan útiles para pausar y reanudar el flujo de ejecución de un script. Esta capacidad es especialmente beneficiosa en tareas como animación, espera de eventos o ejecución de secuencias de acciones en intervalos de tiempo específicos.

Se utilizan cinco variables: "canDash" (booleana), "isDashing" (booleana), "dashingPower" (float), "dashingTime" (float) y "dashingCooldown" (float). Las dos primeras variables se emplean para verificar si el jugador puede ejecutar la mecánica de "Dash" y si ya se encuentra en medio del "Dash". Las últimas dos variables definen la duración del "Dash" y el tiempo de reutilización de esta habilidad. Posteriormente, se crea una nueva corrutina que desactiva la gravedad y aplica una velocidad horizontal cada vez que se ejecuta el "Dash".

```
private IEnumerator Dash()
{
    canDash = false;
    isDashing = true;
    float originalGravity = rb.gravityScale;
    rb.gravityScale = 0f;
    rb.velocity = new Vector2(transform.localScale.x * dashingPower, 0f);
    yield return new WaitForSeconds(dashingTime);
    rb.gravityScale = originalGravity;
    isDashing = false;
    yield return new WaitForSeconds(dashingCooldown);
    canDash = true;
}
```

Ilustración 81: Corrutina Dash en el script Player Controller

Finalmente, se llama a esta corrutina en el método "Update" cada vez que el jugador presiona la tecla Shift izquierdo, activando así el Dash. Además, evitamos que el jugador realice otras acciones mientras está en Dash al verificar si actualmente está ejecutando esta acción o no (ver ilustración 82).

```
void Update()
{
    if (isDashing) Evita que el jugador se pueda mover
    {
        return;
    }

    dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * speed, rb.velocity.y);

    if (Input.GetButtonDown("Jump") && IsGrounded())
    {
        rb.velocity = Vector2.up * jumpVelocity;
    }

    if (Input.GetKeyDown(KeyCode.LeftShift) && canDash)
    {
        StartCoroutine(Dash()); Comienza la corrutina
    }
}
```

Ilustración 82: Verificaciones y llamada de la corrutina Dash en el método Update

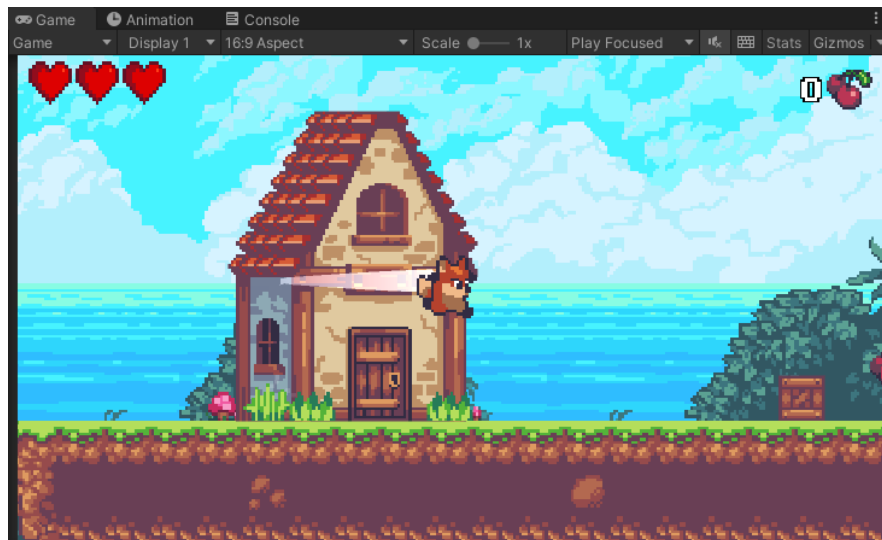


Ilustración 83: Mecánica Dash en Acción

4.7 Cámara

Para la implementación de la cámara en *Star Forest*, se emplea una potente herramienta de Unity llamada "Cinemachine". Esta solución facilita la gestión y creación de sistemas de cámaras en juegos y aplicaciones interactivas. Su diseño tiene como objetivo simplificar el proceso de configuración y control de cámaras en Unity, permitiendo a los

desarrolladores lograr cinematografía de alta calidad y efectos visuales en proyectos sin necesidad de escribir grandes cantidades de código personalizado.

Para incorporar Cinemachine en el proyecto, primero se dirige a la pestaña "Window" y se selecciona "Package Manager". Luego, buscamos el paquete "Cinemachine" y lo añadimos al proyecto.

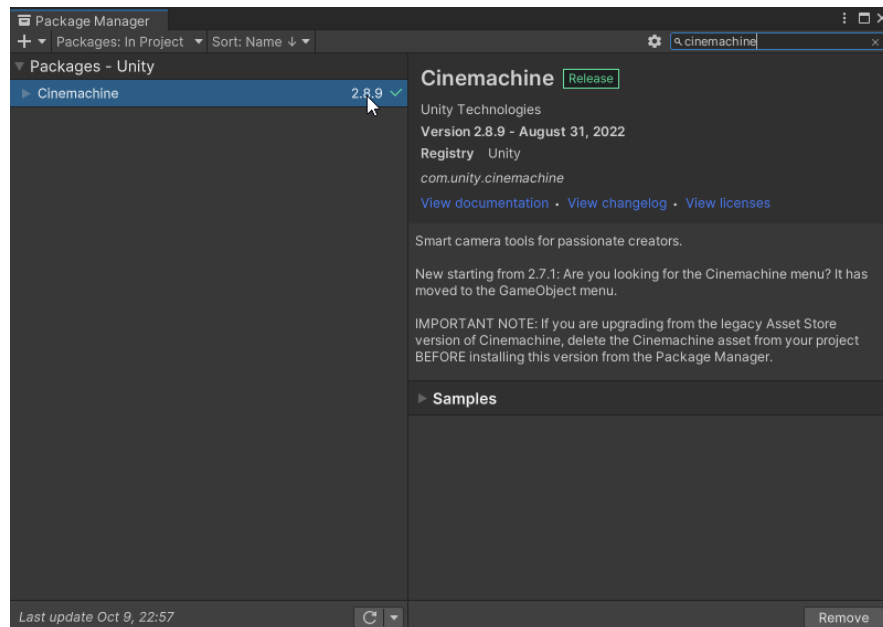


Ilustración 84: Añadiendo Cinemachine al proyecto

Una vez que tenemos Cinemachine en el proyecto, se puede agregar una cámara haciendo clic derecho en la pestaña "Hierarchy". En la nueva sección de "Cinemachine", seleccionamos "2D Camera". Esto creará automáticamente un objeto con los componentes necesarios de la cámara.

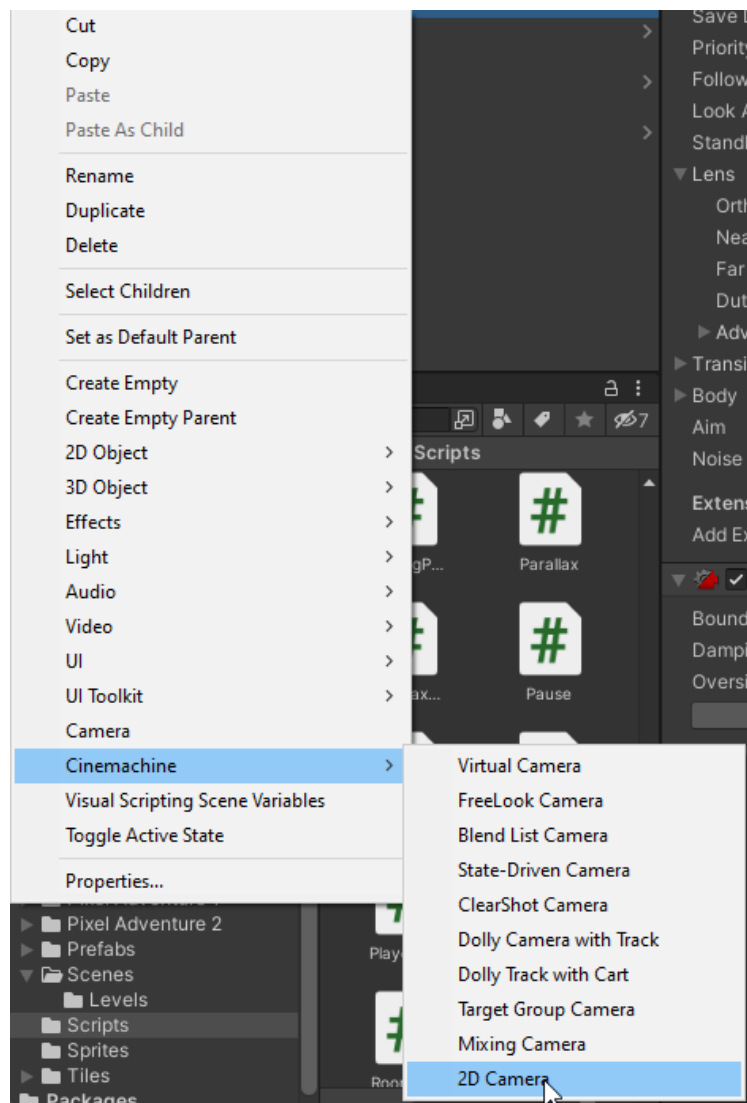


Ilustración 85: Creando un Game Object de tipo Cinemachine

Aunque existen varias opciones para personalizar el comportamiento de la cámara, la más común es la función "Follow", que permite que la cámara siga al objeto "Player" para seguir al personaje por el nivel sin necesidad de escribir código adicional.

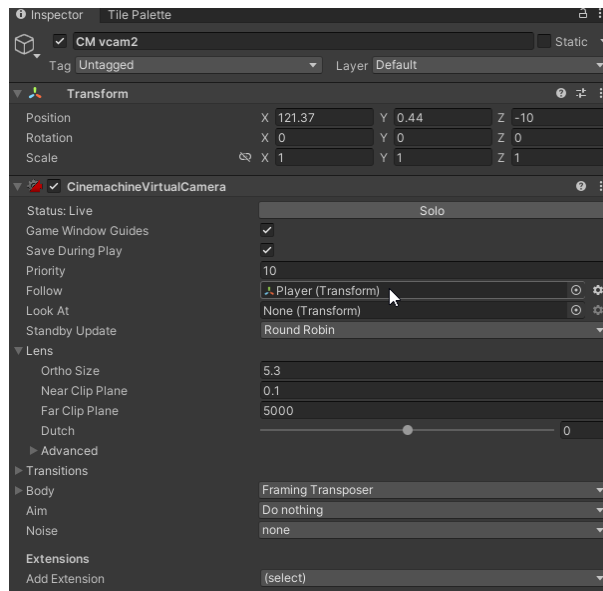


Ilustración 86: Componente Cinemachine en la ventana de Inspector

Sin embargo, a menudo surge un problema: la cámara puede mostrar detalles del nivel que no deberían ser visibles, como se ilustra en la ilustración 87.

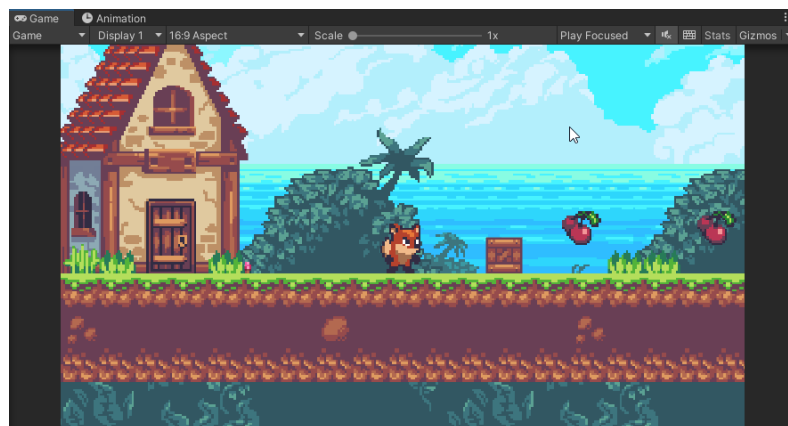


Ilustración 87: Cinemachine sin límites aplicados

Para resolver este problema, Cinemachine incluye una útil extensión llamada "Confiner". Esta característica permite restringir el área en la que la cámara virtual puede moverse dentro de la escena en Unity.

Para hacer funcionar el "Confiner", creamos un objeto con un colisionador de tipo Poligonal llamado "Confiner" y le agregamos el componente "Polygon Collider 2D". Aseguramos que la casilla "Is Trigger" esté marcada.

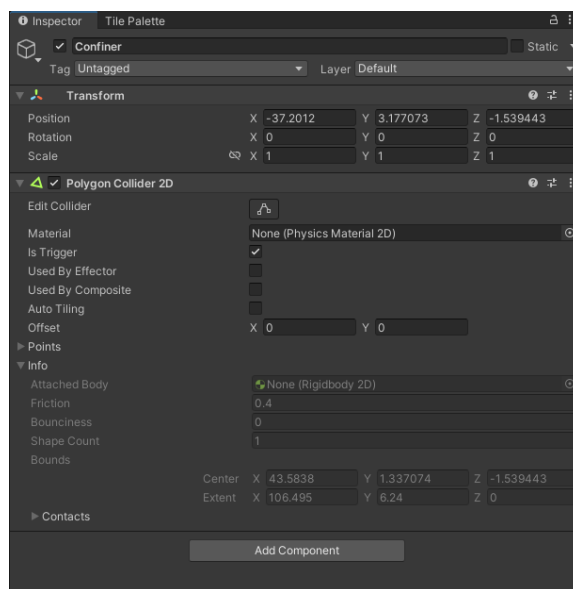


Ilustración 88: Componente Polygon Collider 2D en el Game Object "Confiner"

Luego, ajustamos la forma del "Confiner" para que abarque el tamaño deseado del nivel y oculte los objetos no deseados, como se muestra en la ilustración 89.

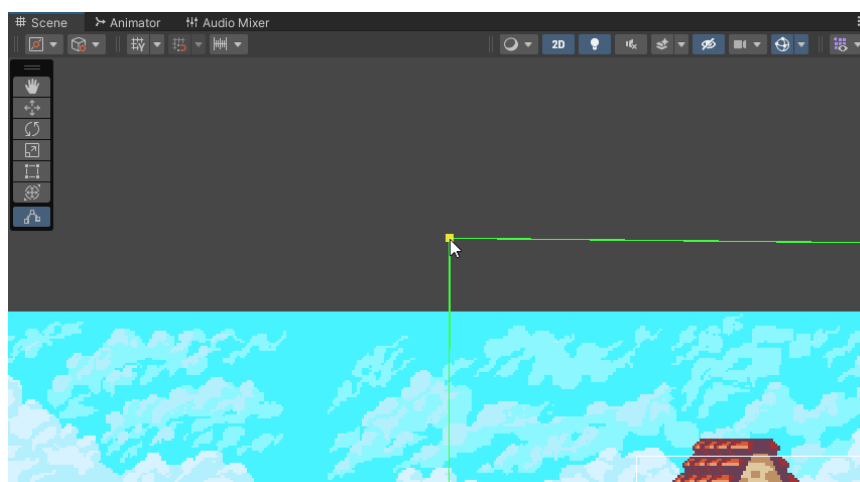


Ilustración 89: Configurando los límites para la cámara

Una vez configurados los límites y el tamaño del "Confiner", lo añadimos como extensión en Cinemachine. De esta manera, la cámara tiene restricciones definidas sobre dónde puede seguir al jugador y hasta dónde puede desplazarse.

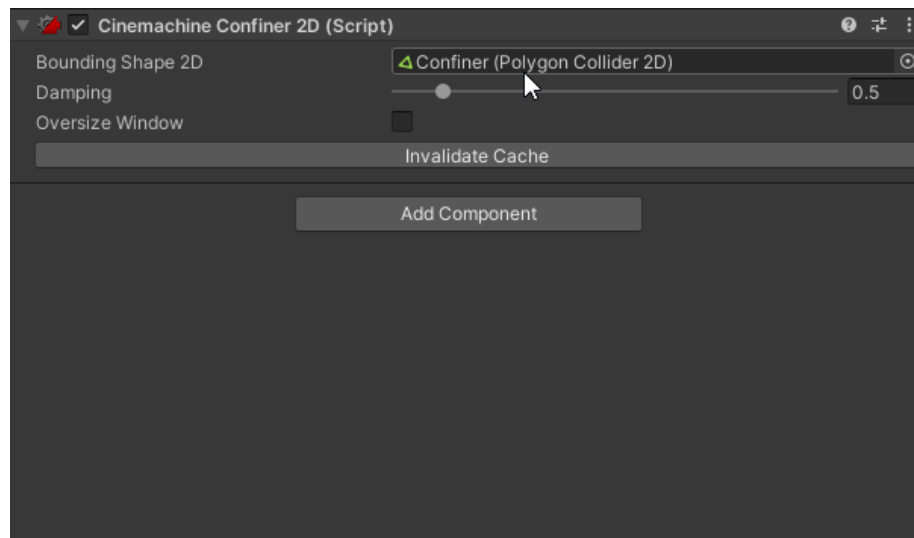


Ilustración 90: Añadiendo el objeto "Confiner" para los límites de la cámara

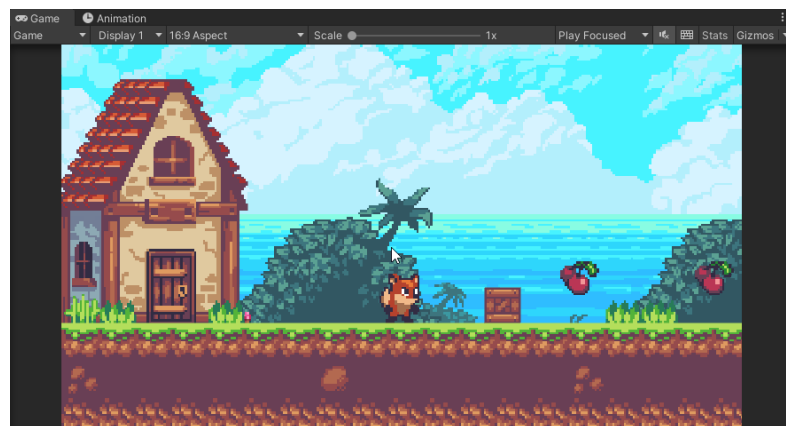


Ilustración 91: Límites aplicados a la cámara

En resumen, Cinemachine es una herramienta poderosa para gestionar cámaras en Unity, lo que permite a los desarrolladores crear efectos visuales cinematográficos y mejorar la jugabilidad de manera eficiente. Esta herramienta es especialmente valiosa en juegos donde la calidad de la cinematografía y la presentación visual son esenciales. Además, la extensión "Confiner" es una valiosa adición que facilita el control y la restricción del movimiento de la cámara virtual en un juego de Unity, lo que resulta útil en juegos de plataformas, aventuras y otros escenarios donde se requiere un control preciso sobre la vista de la cámara dentro de límites específicos en la escena.

4.8 Obstáculos, Daño y Sistema de Salud

Para permitir que el jugador interactúe con el mundo virtual de un videojuego, es esencial incorporar interacciones físicas, y una de ellas consiste en agregar obstáculos en los niveles. Esto se realiza con el propósito de brindar al jugador un desafío al jugar y controlar al personaje mientras navega por los niveles del videojuego. Comenzamos añadiendo un obstáculo simple al escenario.

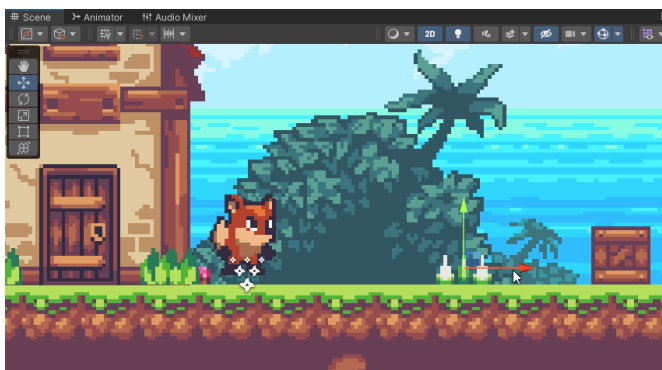


Ilustración 92: Añadiendo un obstáculo al videojuego

Primero, arrastramos la imagen del obstáculo deseado, en este caso, espinas. Para que el jugador pueda interactuar con las espinas, añadimos el componente Box Collider 2D. Con este componente añadido, el jugador puede interactuar con las espinas y recibir daño. Aseguramos ajustar el tamaño de la caja de colisiones y marcamos la casilla "Is Trigger". Si no realizamos este paso, las espinas se comportarán como si fueran una plataforma más. Queremos que el jugador pueda interactuar con las espinas cuando las cajas del personaje y las espinas colisionen entre sí.

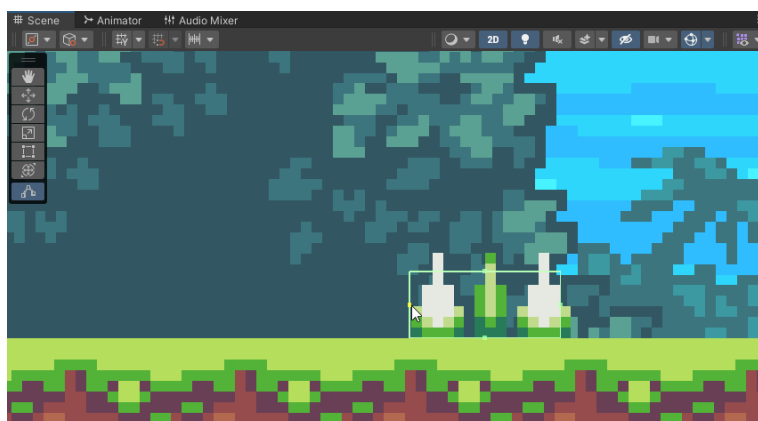


Ilustración 93: Ajustando el tamaño de la caja de colisiones del obstáculo

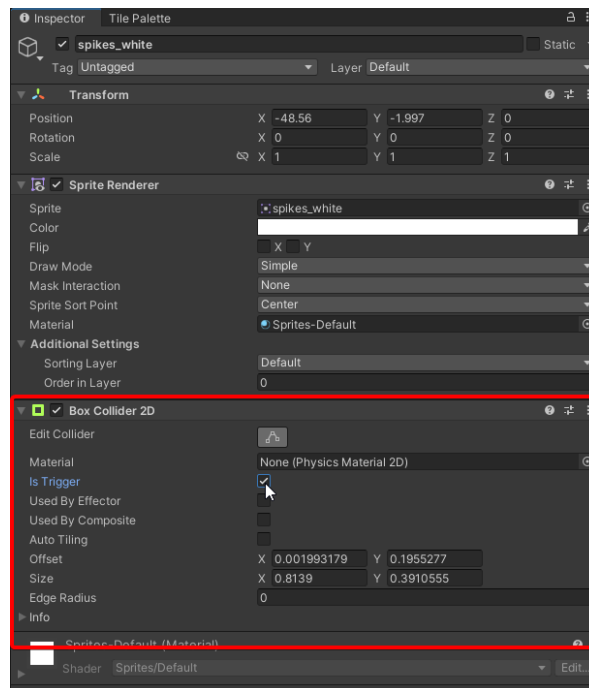


Ilustración 94: Configurando la caja de Colisiones en el inspector

4.8.1 Creando el sistema de salud

Se crea un script llamado "Player Health Controller" que se encarga principalmente de gestionar la vida del jugador, estableciendo variables para la vida actual y la cantidad máxima de salud. Dentro se crea una función para reducir la salud del jugador llamada "Deal Damage", cuyo objetivo es verificar si la salud actual del jugador llega a cero, entonces hace desaparecer al personaje, simulando que éste ha perdido.

```
public void DealDamage()
{
    currentHealth--;

    if (currentHealth <= 0)
    {
        currentHealth = 0;
        gameObject.SetActive(false);
    }
}
```

Ilustración 95: Función "Deal Damage" en el script Player Health Controller

Ahora, el jugador puede perder salud y desaparecer cuando su salud llega a 0. Sin embargo, este código solo se encarga de reducir la vida del jugador. Todavía necesitamos un objeto que dañe al jugador y funcione junto con la función "Deal Damage". Para eso, se crea un script adjunto a las espinas, ya que serán las responsables

de dañar al jugador, y para que funcione correctamente se asigna una etiqueta al jugador que permite verificar con qué objeto específico están colisionando las espinas al escribir el código para dañar al jugador.

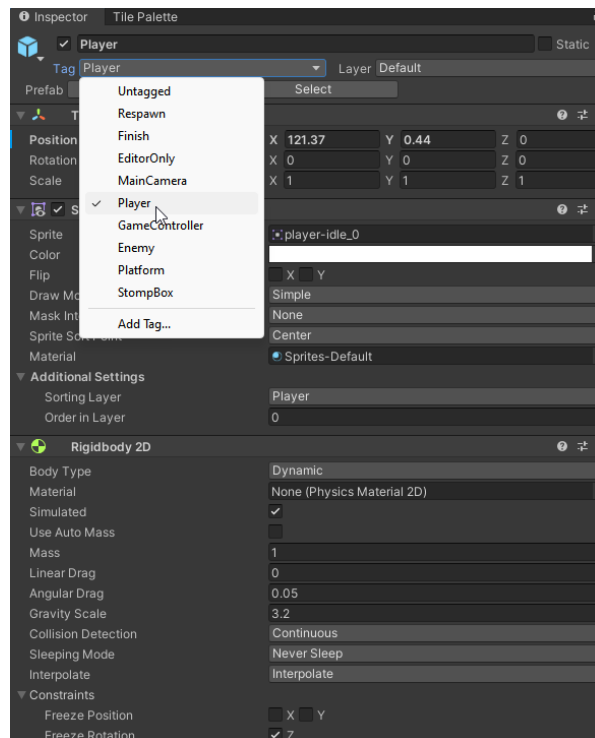


Ilustración 96: Asignando etiquetas al Jugador

Luego se instancia el script "Player Health Controller". Esto significa que permitimos que otros scripts accedan a las funciones de "Player Health Controller" de manera más sencilla. Dado que solo utilizaremos un script para administrar la salud del jugador, instanciamos este script. Al principio del código, creamos una variable pública de tipo "static" que proviene de "Player Health Controller" y llamamos a esta variable "instance".

Definir una variable como "static" implica crear una variable que pertenece a la clase en lugar de a una instancia particular. La variable es compartida entre todas las instancias y se puede acceder a ella sin necesidad de crear una específica para la clase. En otras palabras, solo habrá un "Player Health Controller" al que cada componente o script de Unity podrá acceder en lugar de crear múltiples controladores para la salud del jugador.

```

public class PlayerHealth : MonoBehaviour
{
    public static PlayerHealth instance;

    Mensaje de Unity | 0 referencias
    void Awake()
    {
        instance = this;
    }
}

```

Ilustración 97: Instanciando el script Player Health Controller

4.8.2 Dañando al Jugador

Se crea un script llamado "Damage Player" que se adjunta a las espinas, y utiliza la función de Unity "On Trigger Enter 2D" para detectar la colisión con el jugador, accediendo al script "Player Health Controller" por medio de la instancia creada anteriormente para aplicar daño al jugador.

```

Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        PlayerHealth.instance.DealDamage();
    }
}

```

Ilustración 98: Función para dañar al jugador y restarle puntos de salud

Existen diversas formas de abordar la programación de cómo se le provoca daño al jugador, y la estrategia empleada en esta ocasión contribuye significativamente al rendimiento del juego.

4.8.3 Interfaz del Sistema de Salud

Se crea un objeto de tipo UI Canvas para gestionar la interfaz del juego. Este contendrá todo lo que respecta a interfaces: menús, cantidad de objetos coleccionados, interfaz de salud del jugador, etc. Se añade un objeto "Image" para representar la salud del jugador mediante corazones.

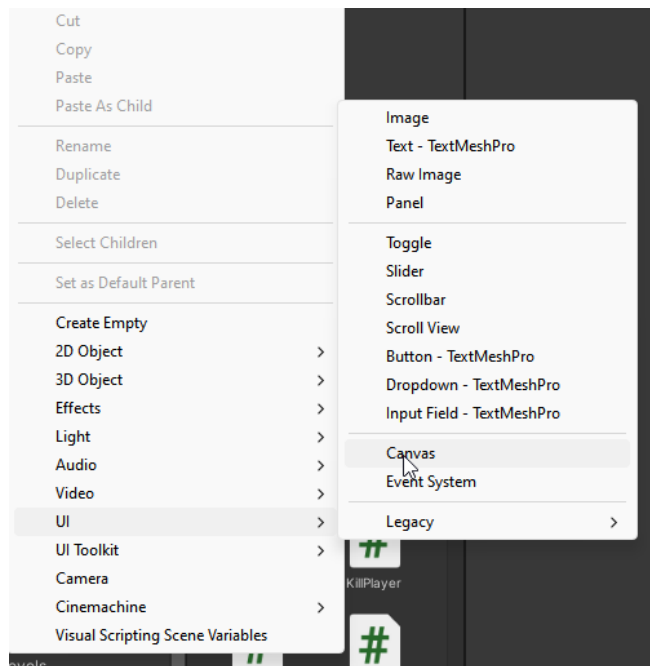


Ilustración 99: Añadiendo un game object de tipo Interfaz

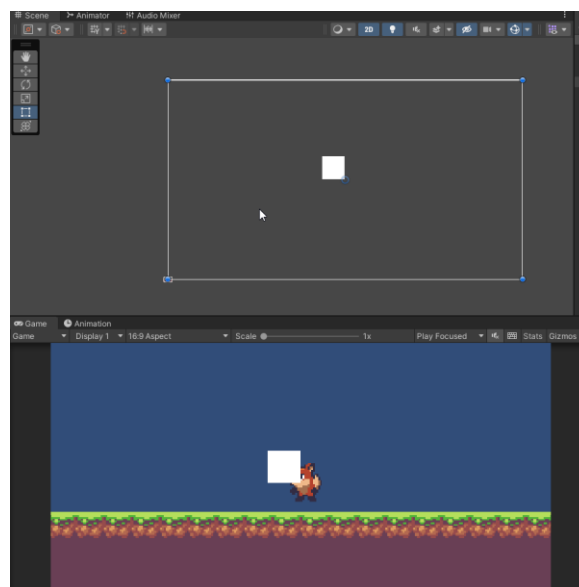


Ilustración 100: Interfaz de tipo imagen en Unity

Para cambiar la apariencia del cuadrado, se selecciona (en este caso) la imagen del corazón y nos dirigimos al apartado que dice "Source Image". Al hacer clic en el círculo junto a él en el Inspector, buscamos el gráfico que deseamos utilizar para el sistema de salud. En este caso, usamos un gráfico de corazón.

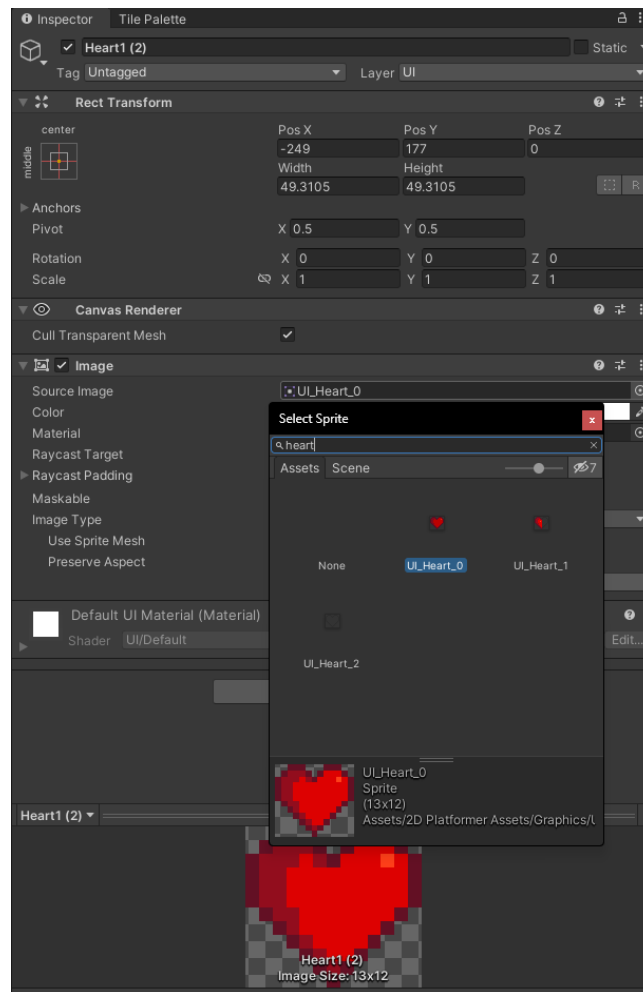


Ilustración 101: Cambinado la imagen del cuadrado a la de un corazón

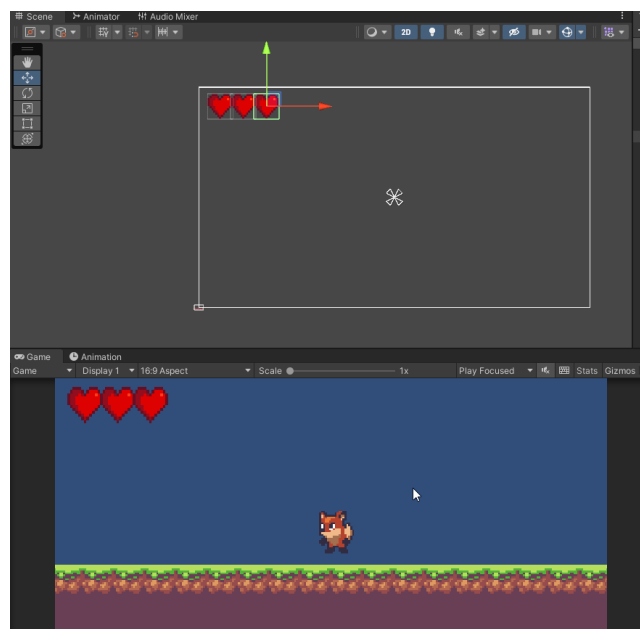


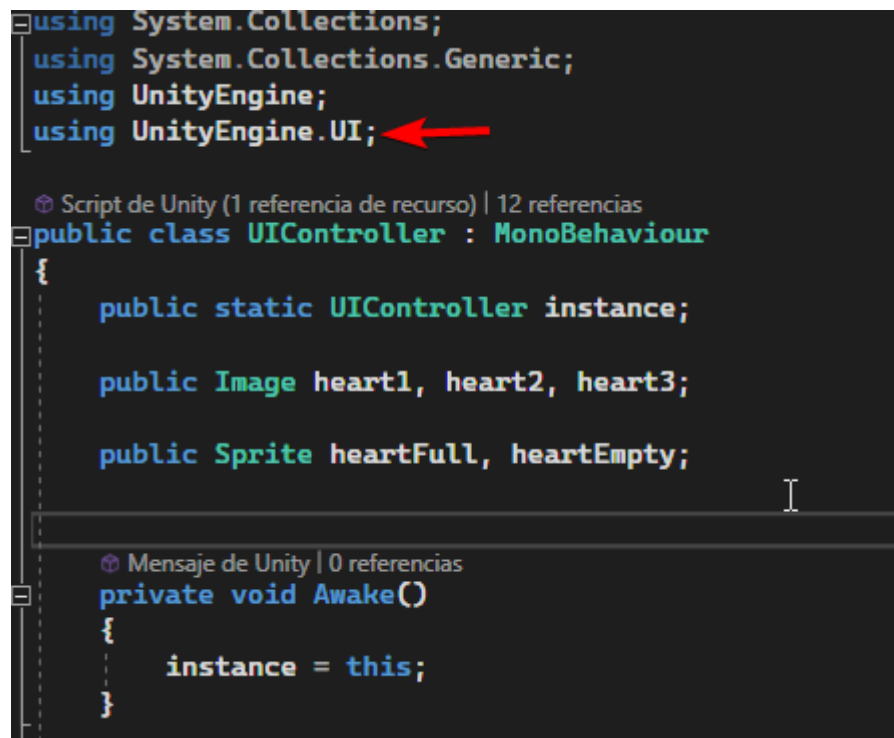
Ilustración 102: Posicionando los corazones en el juego

Luego se crea un script llamado "UIController" para gestionar la actualización visual de la interfaz de salud.

4.8.4 Actualizando el Sistema de Salud

Se utiliza un script para cambiar visualmente la interfaz de salud cuando el jugador recibe daño, ajustando los corazones en la interfaz para reflejar la cantidad de salud actual a través de una función llamada "Update Health Display" que utiliza una estructura de control de flujo llamada "switch". También se procede a instanciar el script "UIController" para actualizar la interfaz del sistema de salud en el momento en que el jugador reciba daño.

Es importante que para poder utilizar componentes de la interfaz de Unity se instruya en el inicio del código se utilizan los componentes de "UnityEngine.UI".



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

Script de Unity (1 referencia de recurso) | 12 referencias
public class UIController : MonoBehaviour
{
    public static UIController instance;

    public Image heart1, heart2, heart3;

    public Sprite heartFull, heartEmpty;

    private void Awake()
    {
        instance = this;
    }
}
```

Ilustración 103: "UnityEngine.UI" y variables en el script UI Controller

En el inspector, los objetos de tipo imagen llamados "Heart1", "Heart2" y "Heart3" se asocian con las variables en el apartado del inspector, mientras que "heartFull" y "heartEmpty" se les asocian con los gráficos del corazón lleno y del corazón vacío.

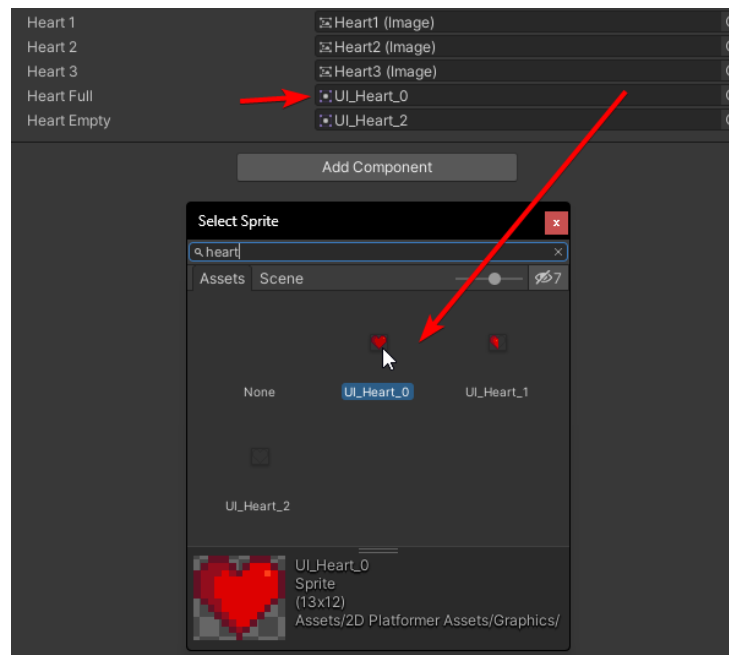


Ilustración 104: Asignando corazón lleno en el inspector

Luego, en el código, se utiliza la estructura de control de flujo llamada "switch" mencionada anteriormente, la cual toma decisiones basadas en el valor de una expresión. Dentro del switch, se utiliza el valor de la vida actual ("currentHealth") para determinar cuántos puntos de salud tiene el jugador. Si su valor es tres (indicando salud completa), los sprites o avatares de cada corazón utilizarán la imagen de un corazón completo. En el caso de que sea dos, solo un corazón cambiará su imagen a uno vacío "heartEmpty", indicando que el jugador recibió daño, quedando solo dos corazones completos, y así sucesivamente hasta que el jugador no tenga corazones, indicando que ha perdido (ver ilustración 105).

Se asigna un valor por defecto a los corazones en la estructura de switch para que todos sean corazones vacíos. Esto se realiza para el caso en que el valor de la vida actual tenga un valor que no sea tres, dos, uno o cero, evitando problemas inesperados.

```

3 referencias
public void UpdateHealthDisplay()
{
    switch (PlayerHealth.instance.currentHealth)
    {
        case 3:
            heart1.sprite = heartFull;
            heart2.sprite = heartFull;
            heart3.sprite = heartFull;

            break;

        case 2:
            heart1.sprite = heartFull;
            heart2.sprite = heartFull;
            heart3.sprite = heartEmpty;

            break;

        case 1:
            heart1.sprite = heartFull;
            heart2.sprite = heartEmpty;
            heart3.sprite = heartEmpty;

            break;

        case 0:
            heart1.sprite = heartEmpty;
            heart2.sprite = heartEmpty;
            heart3.sprite = heartEmpty;

            break;

        default:
            heart1.sprite = heartEmpty;
            heart2.sprite = heartEmpty;
            heart3.sprite = heartEmpty;

            break;
    }
}

```

Ilustración 105: Estructura switch en el script UI Controller

Finalmente, se actualiza la interfaz de los corazones cada vez que el jugador recibe daño, utilizando la instancia del script "UIController" para poder utilizar la función de actualización de la interfaz del sistema de salud.

```

public void DealDamage()
{
    currentHealth--;

    if (currentHealth <= 0)
    {
        currentHealth = 0;
        gameObject.SetActive(false);
    }

    UIController.instance.UpdateHealthDisplay();
}

```

Ilustración 106: Actualizando la interfaz en el script Player Health Controller

4.8.5 Añadiendo invulnerabilidad al jugador

Cuando el jugador recibe daño, es crucial que disponga de un periodo de invulnerabilidad para tener la oportunidad de escapar o superar el obstáculo que le causó daño. La ausencia de este sistema de invulnerabilidad podría resultar en un daño continuo, generando una experiencia de juego injusta. En el caso de *Star Forest*, la

penalización por no eludir los obstáculos se limita a perder únicamente un corazón, y es aquí donde entra en juego el sistema de invulnerabilidad.

En el script de "PlayerHealthController", se crean dos variables:

- **invincibleLength:** Representa la duración de la invulnerabilidad del jugador tras recibir daño.
- **invincibleCounter:** Este contador está vinculado a la duración de la invulnerabilidad. Se utiliza en funciones que requieren verificar cuántos segundos de invulnerabilidad le quedan al jugador antes de que pueda recibir daño nuevamente. Dado que es un valor de tipo contador, y no queremos que su valor sea modificado inadvertidamente, se ha definido como una variable privada.

```
public float invincibleLength;  
private float invincibleCounter;
```

Ilustración 107: Variables para el sistema de invulnerabilidad en el script Player Health Controller

Dentro de la función "DealDamage" en el mismo script, cada vez que la salud del jugador no sea cero, se establece que el valor de "invincibleCounter" sea igual al de "invincibleLength". Esta acción asegura que cada vez que el jugador reciba daño, "invincibleCounter" alcance el valor de "invincibleLength", que podría ser, por ejemplo, un segundo. Luego, para garantizar que el jugador deje de recibir daño, se verifica que cada vez que el contador de invulnerabilidad sea mayor a cero. Si "invincibleCounter" es menor o igual a cero, se ejecutará la sección correspondiente del código.

```

public void DealDamage()
{
    if (invincibleCounter <= 0)
    {
        currentHealth--;

        if (currentHealth <= 0)
        {
            currentHealth = 0;

            gameObject.SetActive(false);
        }
        else
        {
            invincibleCounter = invincibleLength;
        }

        UIController.instance.UpdateHealthDisplay();
    }
}

```

Ilustración 108: Añadiendo verificaciones con las variables del sistema de invulnerabilidad

Al ejecutar el juego, el sistema de invulnerabilidad funciona correctamente, pero ahora el jugador parece no poder recibir daño, ya que permanece invulnerable de forma permanente. Para solucionar esto, en el método "Update" de "PlayerHealthController", se verifica si el contador de invulnerabilidad es mayor que cero (o sea, está en estado vulnerable). En caso afirmativo, se resta al contador la cantidad de tiempo transcurrido hasta el siguiente fotograma, utilizando "Time.deltaTime".

```

void Update()
{
    if (invincibleCounter > 0)
    {
        invincibleCounter -= Time.deltaTime;
    }
}

```

Ilustración 109: Verificando si el jugador se encuentra en estado de invulnerabilidad

Con estas modificaciones, el sistema de invulnerabilidad está completamente funcional.

4.8.6 Retroalimentación visual del sistema de invulnerabilidad

Aunque el sistema de invulnerabilidad está operativo, observamos que el jugador no recibe ninguna indicación visual de que ha dejado de ser invulnerable, lo que podría

generar confusión al recibir daño. Existen diversas maneras de demostrar de manera visual que el jugador está en estado de invulnerabilidad, y una de las que se implementan en *Star Forest* es mediante la modificación de la transparencia del avatar. Simplemente se ajusta la opacidad del personaje para indicar visualmente cuando está en estado de invulnerabilidad.

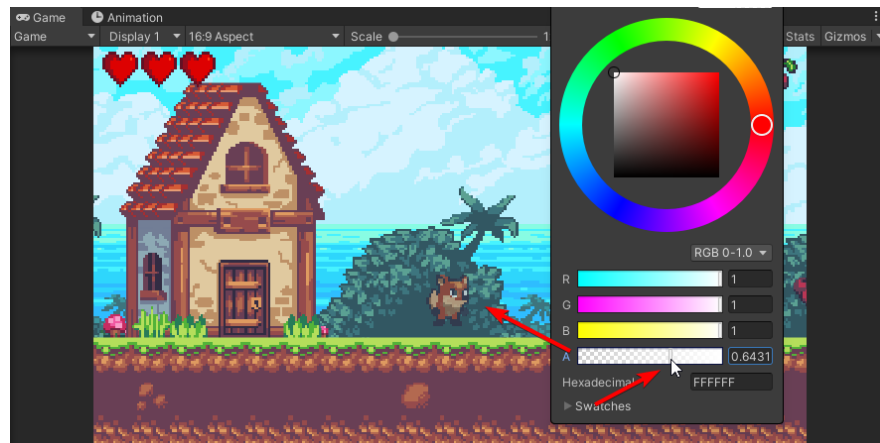


Ilustración 110: Representación visual del sistema de invulnerabilidad

Dentro del script de "PlayerHealthController", se referencia el componente que hace visible al personaje en el juego para poder cambiar la opacidad de éste, y en la función "Deal Damage", se le reduce la opacidad al jugador en el momento en que sea dañado y todavía le queden puntos de salud.

```
public void DealDamage()
{
    if (invincibleCounter <= 0)
    {
        currentHealth--;

        if (currentHealth <= 0)
        {
            currentHealth = 0;

            gameObject.SetActive(false);
        }
        else
        {
            invincibleCounter = invincibleLength;
            sr.color = new Color(sr.color.r, sr.color.g, sr.color.b, .7f);
        }

        UIController.instance.UpdateHealthDisplay();
    }
}
```

Ilustración 111: Cambiando la opacidad del personaje en el script Player Health Controller

En el método "Update" de "Player Health Controller", se realiza una verificación cada vez que el contador de invulnerabilidad sea menor o igual a cero (indicando que el jugador no está en dicho estado o aún no ha recibido daño). Se le asigna un valor de

opacidad al color al que tenía por defecto (es decir, el valor de 1). Esto es principalmente para que el jugador no se mantenga con una opacidad baja permanente luego de recibir daño.

```
void Update()
{
    if (invincibleCounter > 0)
    {
        invincibleCounter -= Time.deltaTime;

        if (invincibleCounter <= 0)
        {
            sr.color = new Color(sr.color.r, sr.color.g, sr.color.b, 1f);
        }
    }
}
```

Ilustración 112: Restaurando la opacidad del jugador en el método Update

Ahora, el jugador tendrá una indicación visual clara de cuándo entra y sale del estado de invulnerabilidad cada vez que recibe daño, sin ningún problema.

4.9 Puntos de Control y Reparición

En la actualidad del videojuego, el jugador carece de la capacidad de continuar jugando después de perder todos los corazones. La intención es que pueda reaparecer en el nivel después de haber perdido, con el objetivo de intentar superar los obstáculos. En este subcapítulo, se introducen los puntos de control y el sistema de reaparición.

4.9.1 Añadiendo un punto de control

Primero, se incorpora un objeto que representa un punto de control en el juego. Este objeto interactúa al entrar en su caja de colisiones. Se agrega arrastrando el avatar o sprite deseado a la escena y se le añade una caja de colisiones circular ("Circle Collider 2D"), configurado como trigger para colisionar con el jugador.

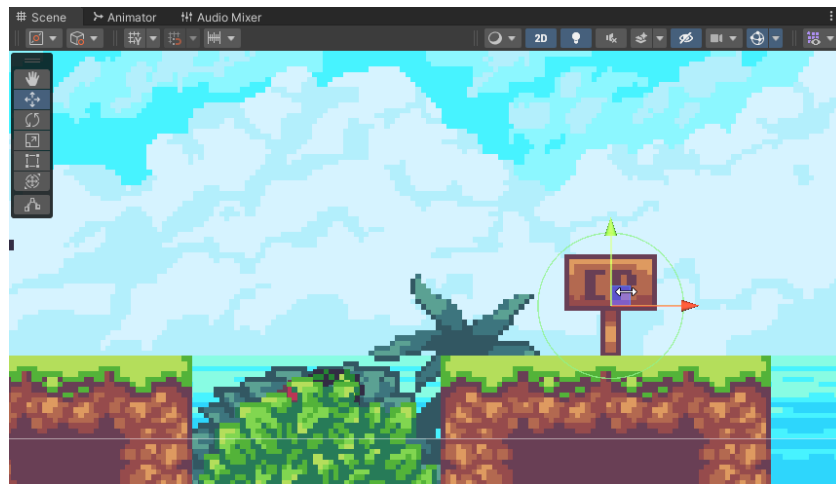


Ilustración 113: Añadiendo punto de control en el proyecto

Se crea un nuevo script llamado "Checkpoint" que cambia el gráfico del punto de control al colisionar con el jugador, indicando que está activado. Este script tiene tres variables, una para obtener los gráficos o visuales del punto de control y otras dos para los gráficos activado y desactivado.

```
public SpriteRenderer sr;  
public Sprite cpOn, cpOff;
```

Ilustración 114: Variables del script Checkpoint

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        sr.sprite = cpOn;
    }
}
```

Ilustración 115: Activando el punto de control cuando colisiona con el jugador

Se preparan los objetos en el inspector con sus respectivos gráficos y objetos, y se tiene un punto de control funcional. Sin embargo, este aún no está totalmente operativo.

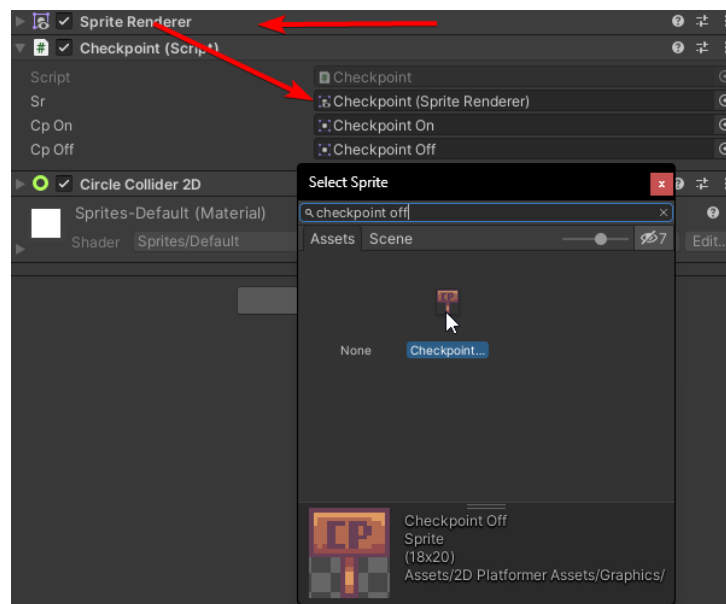


Ilustración 116: Configurando las imágenes del punto de control en el inspector

Aunque al probar el juego el punto de control funciona cuando el jugador colisiona con él, al duplicar otro punto de control, ambos estarán activados.



Ilustración 117: Puntos de control activados simultáneamente

La idea del punto de control es que solo uno esté activado y no varios simultáneamente. Por lo tanto, en el siguiente subcapítulo se explora la forma de desactivarlos y evitar tener varios activados al mismo tiempo. Pero antes, se añade una función pública en el script de "Checkpoint" llamada "ResetCheckpoint", que simplemente cambiará el gráfico del punto de control de activado a desactivado.

```
public void ResetCheckpoint()
{
    sr.sprite = cpOff;
}
```

Ilustración 118: Función "ResetCheckpoint" en el script UI Controller

4.9.2 Desactivando puntos de control

Se necesita un nuevo script, "CheckpointController", para gestionar los puntos de control mediante código. Se crea un objeto llamado "Checkpoint Controller" sin componentes y se asocia el script homónimo. Este script instancia un array de puntos de control, obteniéndolos en el método "Start".

```
public class CheckpointController : MonoBehaviour
{
    public static CheckpointController instance;

    private Checkpoint[] checkpoints;

    // Mensaje de Unity | 0 referencias
    private void Awake()
    {
        instance = this;
    }

    // Start is called before the first frame update
    // Mensaje de Unity | 0 referencias
    void Start()
    {
        checkpoints = FindObjectsOfType<Checkpoint>();
    }
}
```

Ilustración 119: Código base del script "Checkpoint Controller"

Por cada punto de control que colisione con el jugador, se desactivan todos los puntos de control presentes excepto con el que colisionó el jugador.

```
public void DeactivateCheckpoints()
{
    for (int i = 0; i < checkpoints.Length; i++)
    {
        checkpoints[i].ResetCheckpoint();
    }
}
```

Ilustración 120: Recorriendo todos los puntos de control presentes para desactivarlos

Luego, esta función se llama en el script "Checkpoint", utilizando una instancia del script "Checkpoint Controller".

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        CheckpointController.instance.DeactivateCheckpoints();

        sr.sprite = cp0n;
    }
}
```

Ilustración 121: Llamando la función para desactivar puntos de control



Ilustración 122: Puntos de control funcionales

4.9.3 Almacenando el punto de reaparición

Se configura un sistema de reaparición estableciendo un punto de reaparición para el jugador cada vez que colisiona con un punto de control. Se crea una variable "spawnPoint" en el script "CheckpointController" y una función "SetSpawnPoint" que asigna la posición actual en el momento en que el jugador colisiona con un punto de control. Entonces, en el momento en que el jugador entre en contacto con un punto de control, se llama a esta función, obteniendo así la posición del jugador.

```
public void SetSpawnPoint(Vector3 newSpawnPoint)
{
    spawnPoint = newSpawnPoint;
}
```

Ilustración 123: Función para almacenar la posición del jugador

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        CheckpointController.instance.DeactivateCheckpoints();

        sr.sprite = cpOn;

        CheckpointController.instance.SetSpawnPoint(transform.position);
    }
}
```

Ilustración 124: Almacenando la posición del jugador cuando colisiona con un punto de control

Con esto, ahora al colisionar con cada punto de control, se almacena un nuevo punto de reaparición. Sin embargo, aún se requiere la capacidad de permitir al jugador reaparecer.

4.9.4 Sistema de reaparición

Se crea un objeto de juego llamado "Level Manager" con un script homónimo. Este administra los puntos de control y controlará la desactivación y reactivación del jugador. Se crea una variable ("waitToRespawn") que da un breve lapso de tiempo antes de reaparecer. Se crea la corrutina "RespawnCo" para desactivar y reactivar el jugador después de un tiempo y luego hacerlo reaparecer dado un punto o posición establecido, seguido de eso se actualiza la interfaz de la salud para evitar el problema de cuando el jugador, al reaparecer, no posea corazones o puntos de salud. Para poder activar esta corrutina, se crea una función llamada "RespawnPlayer" que la ejecuta.

```
2 referencias
public void RespawnPlayer()
{
    StartCoroutine(RespawnCo());
}

1 referencia
private IEnumerator RespawnCo()
{
    PlayerController.instance.gameObject.SetActive(false);

    yield return new WaitForSeconds(waitToRespawn);

    PlayerController.instance.gameObject.SetActive(true);

    PlayerController.instance.transform.position = CheckpointController.instance.spawnPoint;

    PlayerHealth.instance.currentHealth = PlayerHealth.instance.maxHealth;
    UIController.instance.UpdateHealthDisplay();
}
```

Ilustración 125: Código base del script Level Manager

En "PlayerHealthController", cuando la salud es cero, se llama a "RespawnPlayer".

```

public void DealDamage()
{
    if (invincibleCounter <= 0)
    {
        currentHealth--;

        if (currentHealth <= 0)
        {
            currentHealth = 0;

            LevelManager.instance.RespawnPlayer();
        }
        else
        {
            invincibleCounter = invincibleLength;
            sr.color = new Color(sr.color.r, sr.color.g, sr.color.b, .7f);

            PlayerController.instance.KnockBack();
        }

        UIController.instance.UpdateHealthDisplay();
    }
}

```

Ilustración 126: Haciendo reaparecer al jugador cuando su punto de salud llega a cero

Finalmente, en el script "CheckpointController", en el método "Start", se asigna la posición actual al punto de reaparición inicial, dado que, si no se hace esto, el jugador puede reaparecer, por ejemplo, bajo la plataforma.

```

void Start()
{
    checkpoints = FindObjectsOfType<Checkpoint>();

    spawnPoint = PlayerController.instance.transform.position;
}

```

Ilustración 127: Estableciendo punto de reaparición inicial

Con estas modificaciones, el sistema de reaparición y los puntos de control funcionan sin ningún problema en el juego.

4.10 Coleccionables y Efectos visuales

Con todos los sistemas que se han ido añadiendo, el juego va tomando forma de a poco, pero aún faltan ciertos elementos claves. De momento, el jugador puede controlar al personaje, interactuar con los obstáculos, interactuar con puntos de control, hacer que el jugador reaparezca al fallar el nivel y también tiene la capacidad de obtener una retroalimentación en vivo al momento de perder puntos de salud y de perder el juego. Pero al momento de jugar un videojuego de plataformas 2D, se requiere más que saltar y esquivar objetos; el jugador busca poner sus habilidades en práctica, y no

necesariamente lo desea hacer con los obstáculos que el juego presenta. Es allí donde entran los coleccionables como las monedas, objetos que restauran la salud y todo esto va acompañado de un efecto visual.

El jugador, al igual que salta solo por el deseo de hacerlo, busca coleccionar objetos solo por el simple hecho de hacerlo. Por lo que el siguiente objetivo es empezar a añadir coleccionables.

4.10.1 Añadiendo coleccionables

Se inicia añadiendo un objeto que representa un coleccionable al escenario, en este caso, una cereza, luego se le añade una caja de colisiones de tipo circular ("Circle Collider 2D) y se activa la casilla "Is Trigger" en el inspector de este objeto para que pueda interactuar con el jugador sólo cuando éste colisione con el otro.

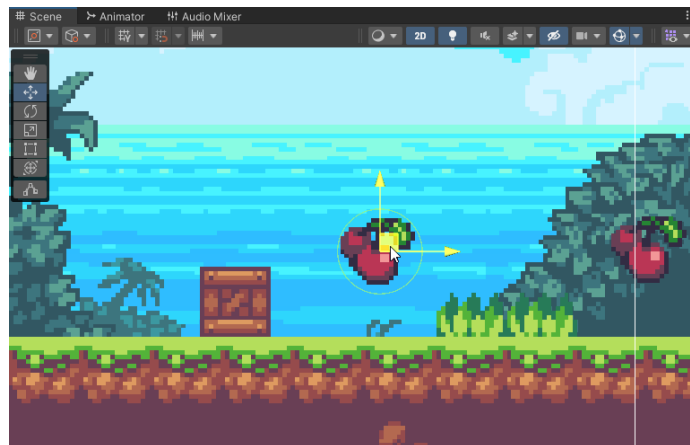
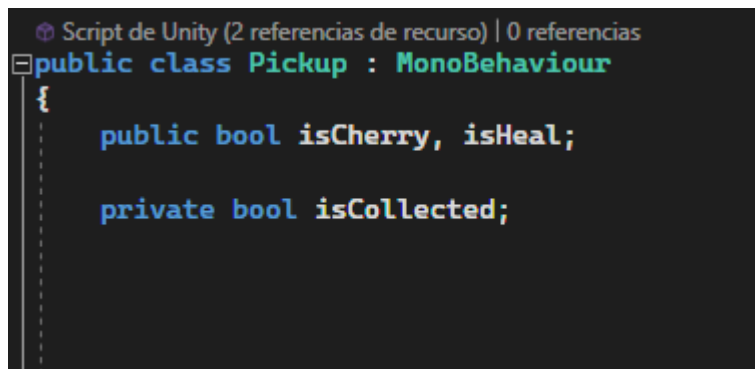


Ilustración 128: Añadiendo objeto coleccionable en el proyecto

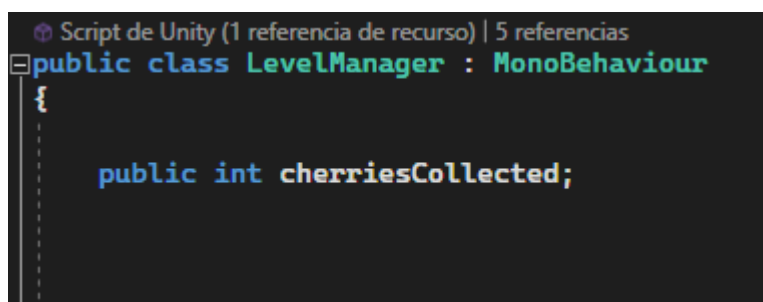
Se crea un script llamado "Pickup" para gestionar la interacción de los coleccionables con el jugador, definiendo variables para identificar el tipo de coleccionable, ya sea una cereza o un corazón que le devuelva puntos de salud al jugador, una variable que indique que el objeto ha sido coleccionado y finalmente una variable que almacena la cantidad de cerezas recolectadas que se crea en el script "Level Manager".



```
Script de Unity (2 referencias de recurso) | 0 referencias
public class Pickup : MonoBehaviour
{
    public bool isCherry, isHeal;

    private bool isCollected;
}
```

Ilustración 129: Variables en el script Pickup

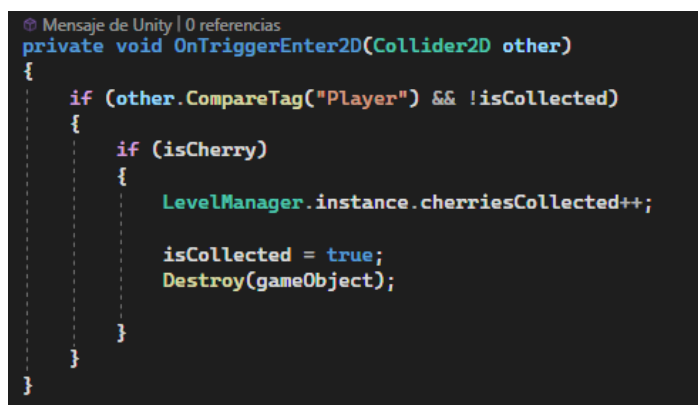


```
Script de Unity (1 referencia de recurso) | 5 referencias
public class LevelManager : MonoBehaviour
{
    public int cherriesCollected;
}
```

Ilustración 130: Variable de recolección en el script Level Manager

Se verifica la colisión del jugador con el coleccionable y se actualiza la interfaz del juego. En caso de ser una cereza, por cada vez que el objeto colisione con el jugador y si no ha sido coleccionado, entonces se le suma 1 utilizando la instancia del script “LevelManager” con la variable “cherriesCollected” (cerezas coleccionadas).

Dado que el jugador colisionó con el objeto, eso quiere decir que este objeto ha sido coleccionado, por lo que la variable “isCollected”, que se encarga de decir si el objeto ha sido recolectado o no, se le asigna el valor de verdadero o “true”, y una vez recolectado, se destruye este objeto.



```
Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player") && !isCollected)
    {
        if (isCherry)
        {
            LevelManager.instance.cherriesCollected++;

            isCollected = true;
            Destroy(gameObject);
        }
    }
}
```

Ilustración 131: Código de recolectando cereza

4.10.2 Coleccionable de Salud

Se añade el coleccionable de salud (corazón) con un manejo diferente al de la cereza. Primero se crea una función en el script "PlayerHealthController" para restaurar la salud del jugador y luego se verifica la colisión con el coleccionable de salud y se actualiza la interfaz de salud.

```
if(isHeal)
{
    if(PlayerHealthController.instance.currentHealth != PlayerHealthController.instance.maxHealth)
    {
        PlayerHealthController.instance.HealPlayer();

        isCollected = true;
        Destroy(gameObject);
    }
}
```

Ilustración 132: Código de recolectando corazón

```
1 referencia
public void HealPlayer()
{
    currentHealth++;

    if (currentHealth > maxHealth)
    {
        currentHealth = maxHealth;
    }

    UIController.instance.UpdateHealthDisplay();
}
```

Ilustración 133: Función "Heal Player" en el script Player Health Controller

La forma en que funciona es que, si al jugador le faltan puntos de salud o corazones, el jugador puede interactuar con coleccionables, permitiéndole colisionar con estos coleccionables. Entonces por cada vez que colisione con objetos coleccionables de tipo corazón, la salud del jugador se regenera por solo un punto y se actualiza la interfaz.

4.10.3 Interfaz para las cerezas

Se utiliza un objeto de tipo interfaz que utiliza el componente de texto para mostrar la cantidad de cerezas recolectadas. Se ajusta el tamaño, fuente y posición del texto para que sea visible y luego se crea un script "UIController" para que gestione la interfaz y mostrar la cantidad de cerezas recolectadas.

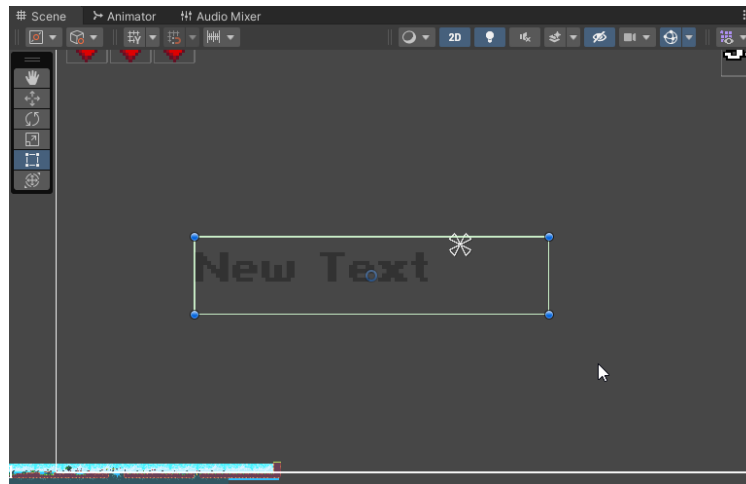


Ilustración 134: Añadiendo texto en el proyecto

Para que el texto se vea más visible, se le añade un componente llamado “Outline”, que dibuja una especie de sombra para el texto.

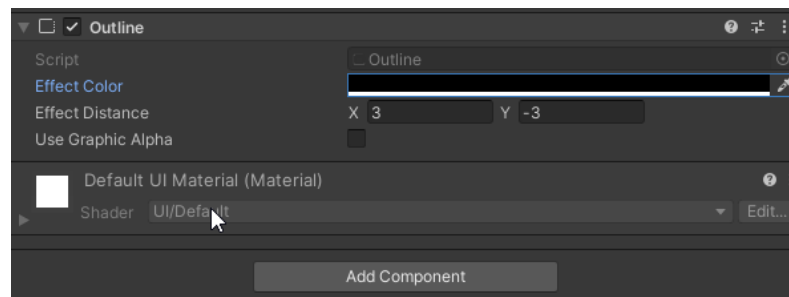


Ilustración 135: Componente "Outline" para el texto

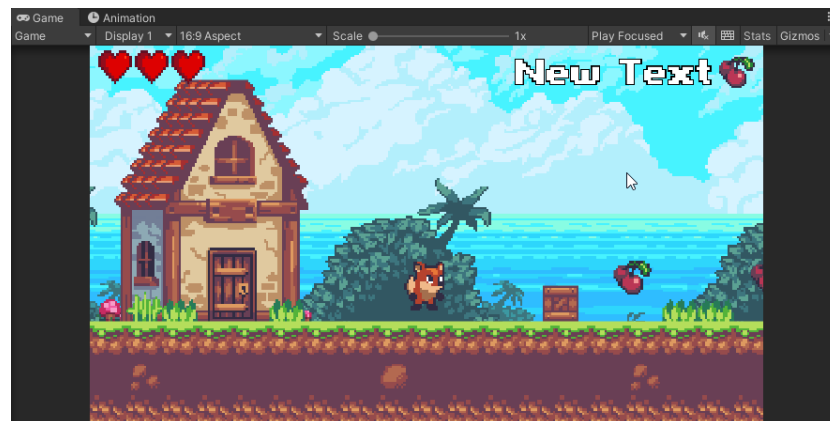


Ilustración 136: Texto con sombra aplicada

```
2 referencias
public void UpdateCherryCount()
{
    ...
    cherryText.text = LevelManager.instance.cherriesCollected.ToString();
}
```

Ilustración 137: Función "Update Cherry Count" en el script "UI Controller"

```

if (isCherry)
{
    LevelManager.instance.cherriesCollected++;

    isCollected = true;
    Destroy(gameObject);

    Instantiate(pickupEffect, transform.position, transform.rotation);

    UIController.instance.UpdateCherryCount();
}

```

Ilustración 138: Actualizando el texto con la cantidad de cerezas coleccionadas

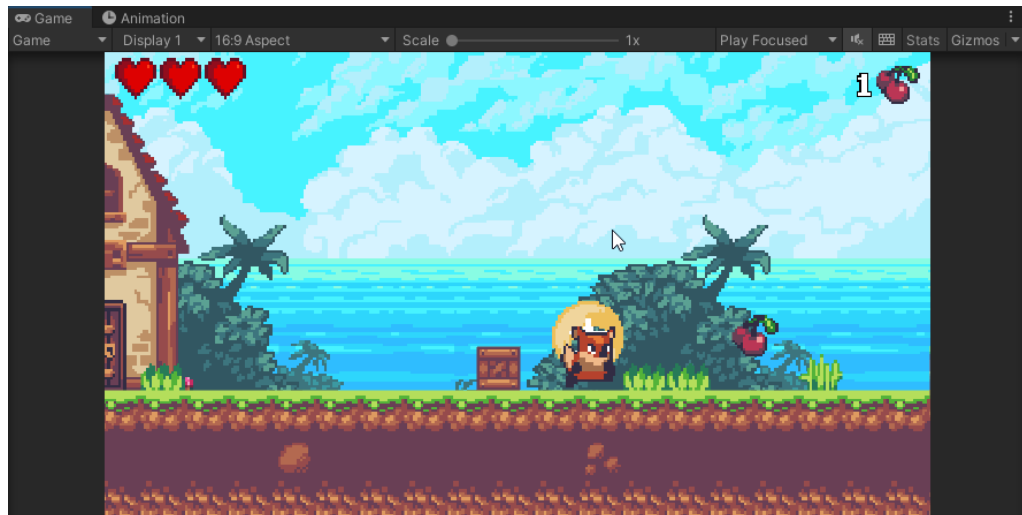


Ilustración 139: Representación visual de cantidad de cerezas coleccionadas en Acción

4.11 Creando al enemigo

En relación con el daño al jugador, se introduce un primer enemigo en el videojuego con patrones de movimiento predefinidos. Se utilizan dos "Game Objects": uno para la imagen gráfica del enemigo y otro vacío para gestionar colisiones y comportamiento. Un script llamado "Enemy Controller" se encarga de mover el enemigo entre puntos definidos, cambiando su dirección y aplicando pausas.

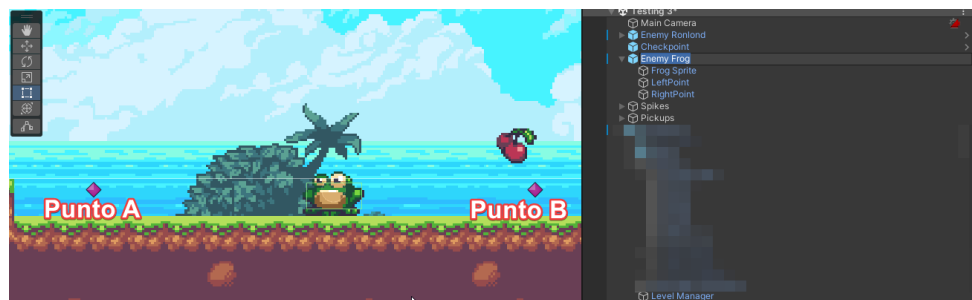


Ilustración 140: Estableciendo puntos de patrullaje del Enemigo

```
void Update()
{
    if (moveCount > 0)
    {
        moveCount -= Time.deltaTime;

        if (movingRight)
        {
            rb.velocity = new Vector2(moveSpeed, rb.velocity.y);

            sr.flipX = true;

            if (transform.position.x > rightPoint.position.x)
            {
                movingRight = false;
            }
        }
        else
        {
            rb.velocity = new Vector2(-moveSpeed, rb.velocity.y);

            sr.flipX = false;

            if (transform.position.x < leftPoint.position.x)
            {
                movingRight = true;
            }
        }

        if (moveCount <= 0)
        {
            waitCount = Random.Range(waitTime * .75f, waitTime * 1.25f);
        }

        anim.SetBool("isMoving", true);
    }

    else if (waitCount > 0)
    {
        waitCount -= Time.deltaTime;
        rb.velocity = new Vector2 (0f, rb.velocity.y);

        if (waitCount <= 0)
        {
            moveCount = Random.Range(moveTime * .75f, waitTime * .75f);
        }
        anim.SetBool("isMoving", false);
    }
}
```

Ilustración 141: Código base de movimiento del Enemigo

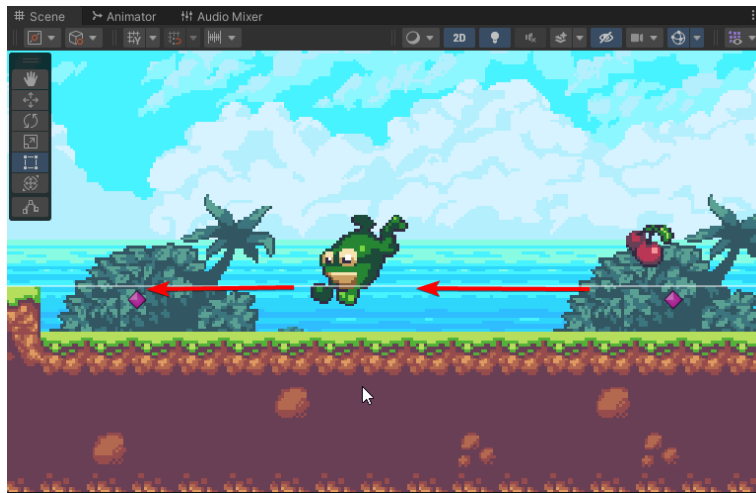


Ilustración 142: Enemigo en movimiento

La destrucción del enemigo se realiza mediante un objeto llamado "StompBox", que utiliza un script para desactivar el sprite del enemigo y aplicar un efecto visual. Además, se añade la capacidad de que el enemigo dañe al jugador mediante un script llamado "Deal Damage" y una caja de colisiones. Con estas implementaciones, el enemigo se vuelve funcional, con animaciones, colisiones y un patrón de movimiento determinado.

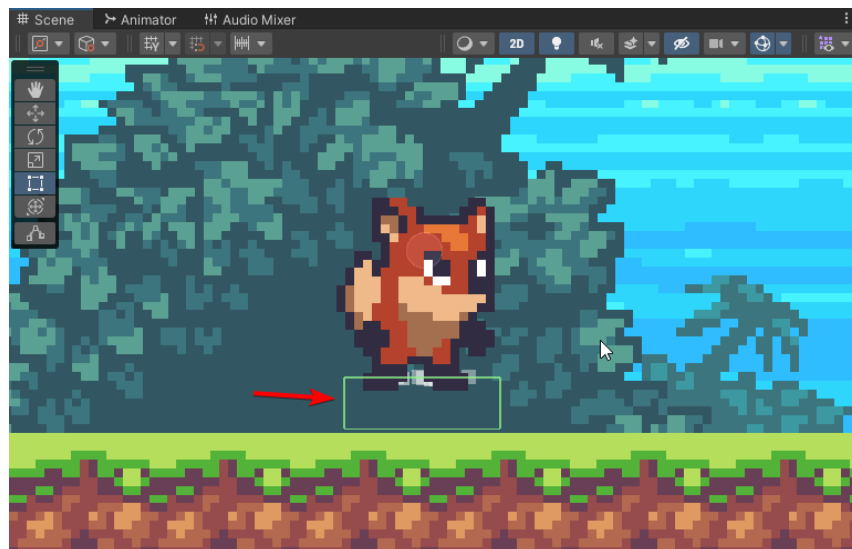


Ilustración 143: Añadiendo el objeto "Stompbox" en el jugador

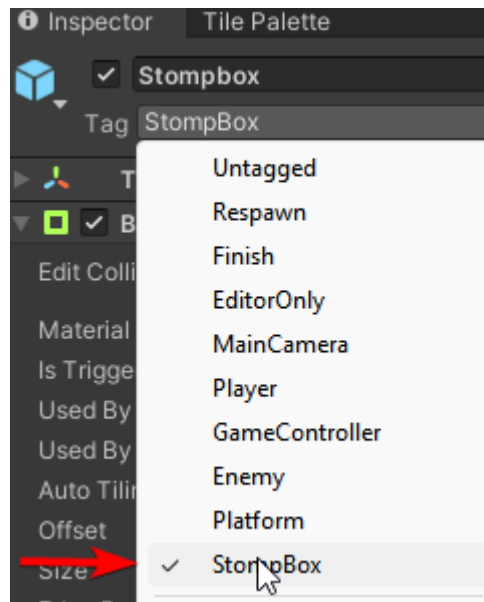


Ilustración 144: Asignando etiqueta al Game Object "StompBox"

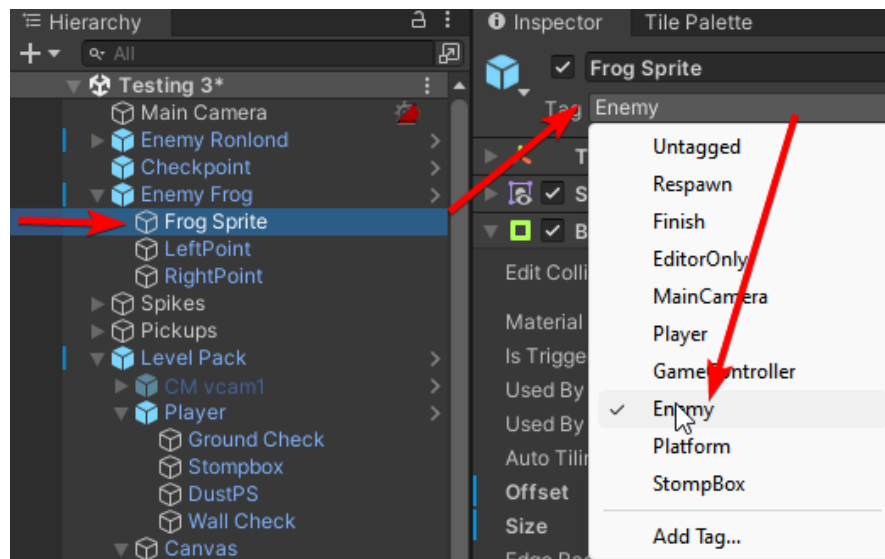


Ilustración 145: Asignando etiqueta al Enemigo

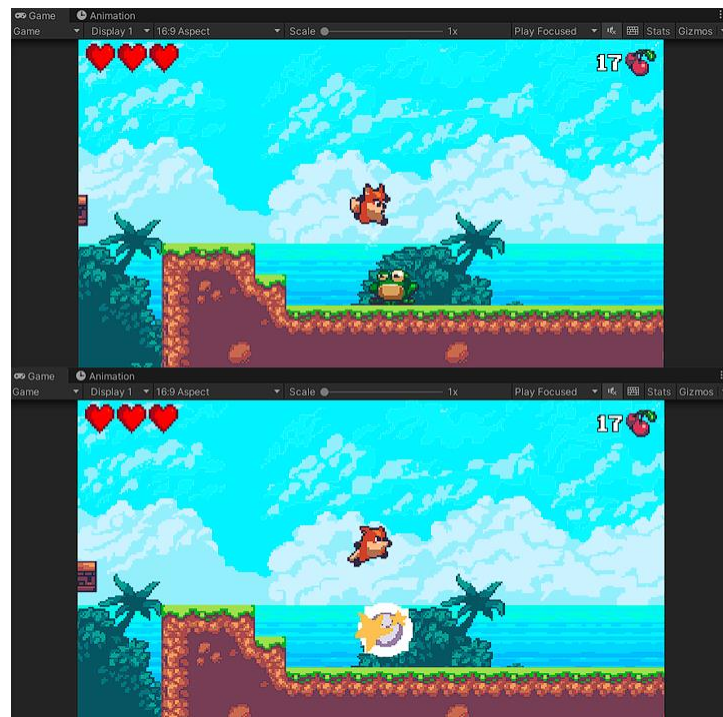


Ilustración 146: Stompbox en Acción

4.12 Añadiendo Sonidos

El audio desempeña un papel crucial en la experiencia del jugador en un videojuego. Para integrar efectos de sonido y música de manera efectiva, se crea un objeto "Audio Manager" que administra todos los sonidos del juego. Este objeto se compone de sonidos como efecto de sonido de salto, recolección de objeto, el personaje siendo dañado, entre otros. Se utiliza un script llamado "AudioManager" para controlar la reproducción y detención de sonidos a través de código. Se destaca la necesidad de desactivar la reproducción automática al inicio para evitar la reproducción simultánea de todos los sonidos. El script incluye una función "PlaySFX" que reproduce efectos de sonido según su posición en un array como, por ejemplo, la reproducción de sonidos al perder salud o derrotar enemigos en scripts específicos. Además, se activan las opciones "Play on Awake" y "Loop" para la música del juego, asegurando que comience automáticamente y se reproduzca de manera continua al inicio.

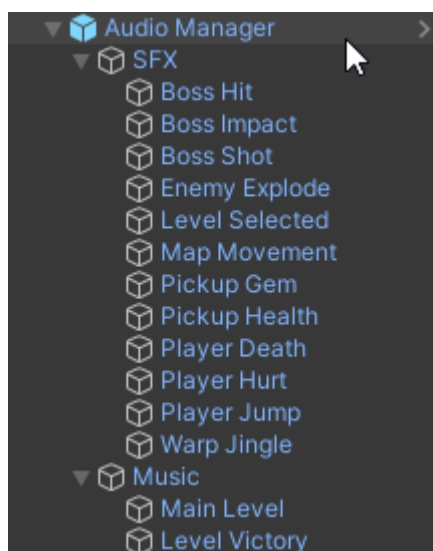


Ilustración 147: Sonidos añadidos en el proyecto

```
public static AudioManager instance;

public AudioSource[] soundEffects;

public AudioSource bgm, levelEndMusic;

// Mensaje de Unity | 0 referencias
private void Awake()
{
    instance = this;
}
```

Ilustración 148: Código base del script Audio Manager

```

public void PlaySFX(int soundToPlay)
{
    soundEffects[soundToPlay].Stop();

    soundEffects[soundToPlay].Play();
}

```

Ilustración 149: Función para reproducir efecto de sonido



Ilustración 150: Almacenando sonidos en un Array o lista

En este punto es donde se inicia la incorporación de efectos de sonido en diferentes scripts para acciones específicas. Por ejemplo, en el script "PlayerHealthController", cada vez que el jugador pierde un punto de salud, se utiliza la función "PlaySFX" del script "AudioManager" para reproducir el sonido correspondiente a la posición 9 (ver ilustración 150).

```

public void DealDamage()
{
    if (invincibleCounter <= 0)
    {
        currentHealth--;

        if (currentHealth <= 0)
        {
            currentHealth = 0;

            gameObject.SetActive(false);

            Instantiate(deathEffect, transform.position, transform.rotation);

            LevelManager.instance.RespawnPlayer();
        }
        else
        {
            invincibleCounter = invincibleLength;
            sr.color = new Color(sr.color.r, sr.color.g, sr.color.b, .7f);

            PlayerController.instance.KnockBack();

            AudioManager.instance.PlaySFX(9);
        }

        UIController.instance.UpdateHealthDisplay();
    }
}

```

Ilustración 151: Reproduciendo el sonido en la función "Deal Damage" del script Player Health Controller

4.13 Menú

Cada videojuego cuenta con un menú de inicio que presenta al jugador las opciones de iniciar o salir del juego. Se incorporan imágenes para visualizar un fondo y el título del juego. A continuación, se crean objetos de interfaz que utilizan componentes de botones, cada uno con sus respectivos gráficos. Estos botones tienen la función de iniciar o salir del videojuego, y son gestionados por un script llamado "Main Menu" o menú principal.



Ilustración 152: Añadiendo Menú de inicio y botones interactivables

Para activar los botones del menú principal, dentro del script "Main Menu" se utiliza una variable que recibe un objeto de tipo escena o "Scene", que básicamente representa los niveles en el proyecto en general. Esta escena representa el primer nivel que se carga cuando el jugador presiona el botón de iniciar el videojuego. Además, para el botón de salir del videojuego, se emplea un método de Unity (Application.Quit) que cierra la aplicación al presionar dicho botón.

```

public class MainMenu : MonoBehaviour
{
    public string startScene;

    // Start is called before the first frame update
    // Mensaje de Unity | 0 referencias
    void Start()
    {
    }

    // Update is called once per frame
    // Mensaje de Unity | 0 referencias
    void Update()
    {
    }

    0 referencias
    public void StartGame()
    {
        SceneManager.LoadScene(startScene);
    }

    0 referencias
    public void QuitGame()
    {
        Application.Quit();
    }
}

```

Ilustración 153: Código base del script Main Menu

Ahora, para que estos botones funcionen correctamente, en el inspector de estos botones, se les asigna la función correspondiente que se creó en el script. En el caso del botón de inicio, se le asigna la función de “StartGame”, y al botón de salida, se le asigna la función “QuitGame”.

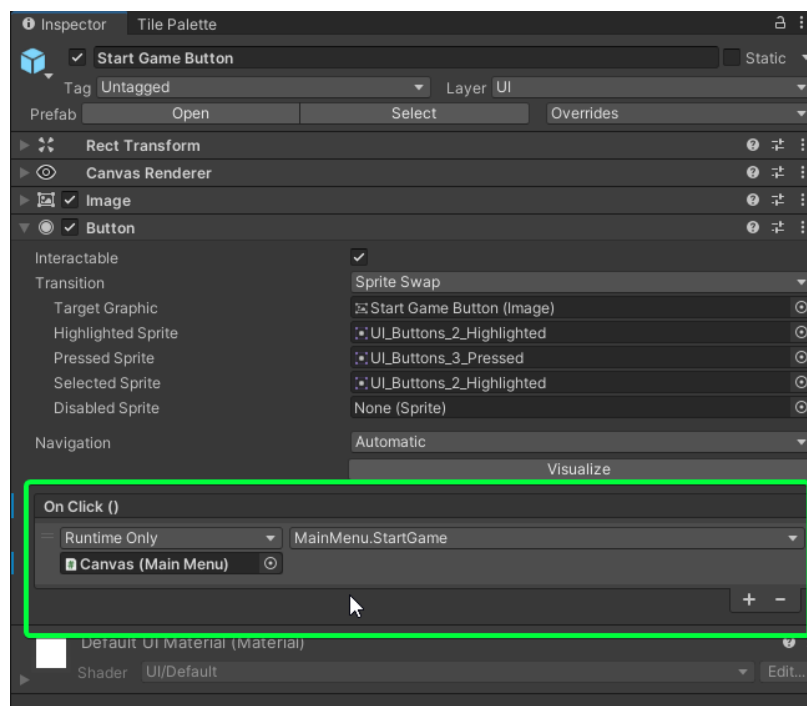


Ilustración 154: Asignando la función de Iniciar juego al botón de Iniciar

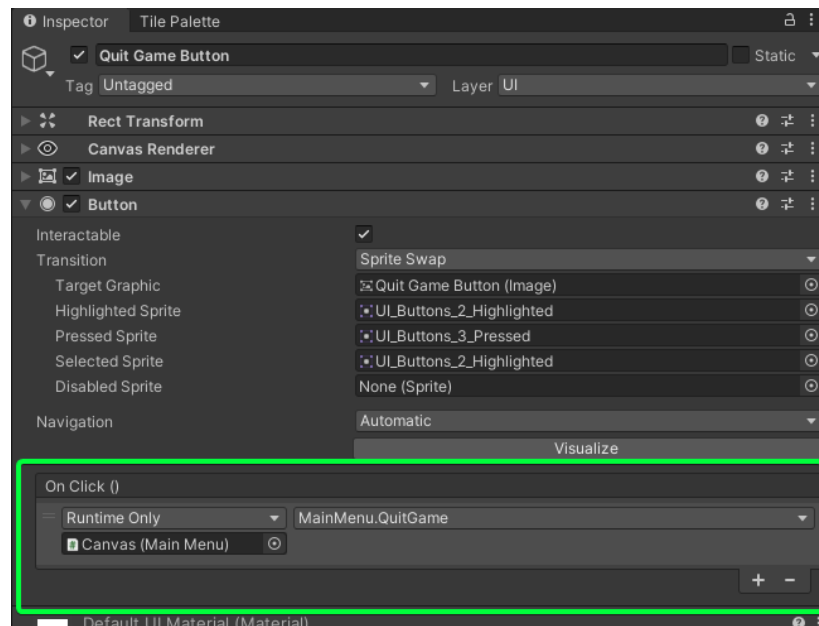


Ilustración 155: Asignando la función de salir del juego al botón de Salir

Posteriormente, para implementar un menú de pausa, se crea una imagen a la que se le ajusta la opacidad, se agrega un texto que indica al jugador que el juego está pausado y se incluyen dos botones: uno para reanudar el juego y otro para regresar al menú principal.



Ilustración 156: Añadiendo menú de pausa

Para que funcione, se crea un script llamado "Pause" o pausa, en el cual se implementa una función que pausa y reanuda el juego según su estado actual. En caso de pausa, se utiliza el método de Unity "time.TimeScale" para detener el tiempo (si es igual a cero) o para reanudarlo (si es igual a uno). Finalmente, para activar la pausa, se indica en el método "Update" que el juego se pause al presionar la tecla de escape.

```

// Update is called once per frame
@ Mensaje de Unity | 0 referencias
void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        PauseUnpause();
    }
}

1 referencia
public void PauseUnpause()
{
    if (isPaused)
    {
        isPaused = false;
        pauseScreen.SetActive(false);
        Time.timeScale = 1f;
    }
    else
    {
        isPaused = true;
        pauseScreen.SetActive(true);
        Time.timeScale = 0f;
    }
}

0 referencias
public void MainMenu()
{
    SceneManager.LoadScene(mainMenu);
    Time.timeScale = 1f;
}
}

```

Ilustración 157: Código base del script Pause

Con esto, se tiene un menú principal y un menú de pausa totalmente funcional en el juego.

4.14 Finalizando el videojuego

En este momento, se ha desarrollado un juego en el que el jugador puede controlar al personaje, con niveles para jugar, sus respectivos coleccionables, tutorial y obstáculos. Por lo tanto, en esta subsección, se transforma el juego *Star Forest* en un software de juego totalmente funcional.

Primero, se hace clic en "File" > "Build Settings".

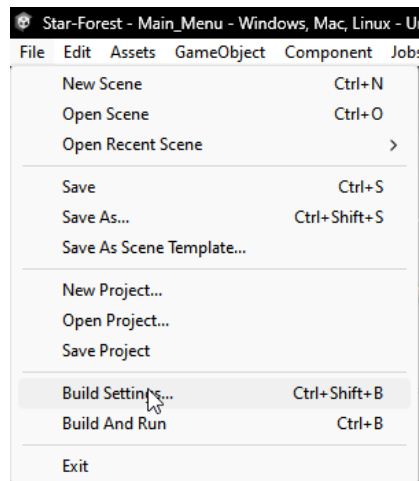


Ilustración 158: Build Settings en Unity

Es en este punto donde definimos el orden de las escenas de nivel y convertimos el proyecto en un software de juego jugable. Se agregan las escenas o niveles, se organizan según el orden deseado y luego se crea el juego para las plataformas específicas, pero en este caso, compilamos el videojuego para la plataforma PC.

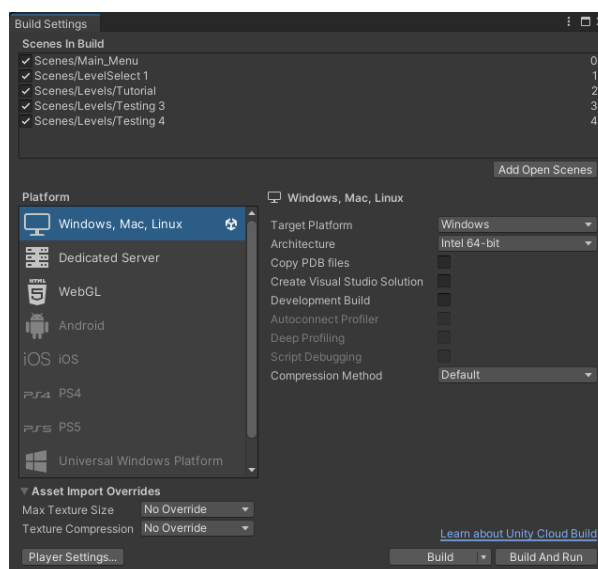


Ilustración 159: Ventana "Build Settings" para crear el juego

Para agregar las escenas creadas para el videojuego (incluyendo el tutorial y el menú principal), simplemente se arrastran a la ventana correspondiente. Una vez completado este paso, se selecciona la plataforma preferida y se hace clic en "Build". Posteriormente, se elige una carpeta para almacenar todos los archivos y se espera a que se complete el proceso de construcción. Al finalizar la compilación, el juego se convierte en un software independiente listo para ser utilizado.

Capítulo 5: Resultados

Se realizó una evaluación de la experiencia de usuario (UX) del juego Star Forest, desarrollado en el marco de este proyecto, con el objetivo de comprender las preferencias y reacciones de los jugadores, proporcionando información crucial para el proceso de diseño del juego. La evaluación se llevó a cabo con la participación de doce personas, cada una de las cuales recibió un formulario para completar después de probar el juego. Los datos recopilados a través de estos formularios fueron utilizados para generar un gráfico que representa el nivel de satisfacción de los jugadores con los diferentes elementos y mecánicas implementadas en el juego.

5.1 Resultados de la Evaluación

Movimiento Básico:

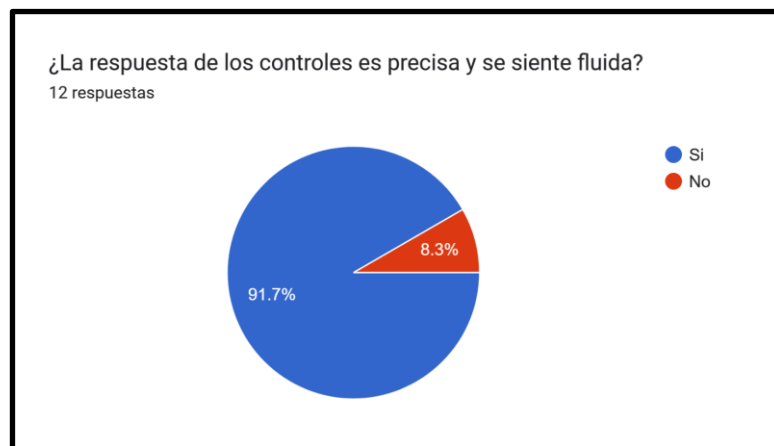


Ilustración 160: ¿La respuesta de los controles es precisa y se siente fluida?

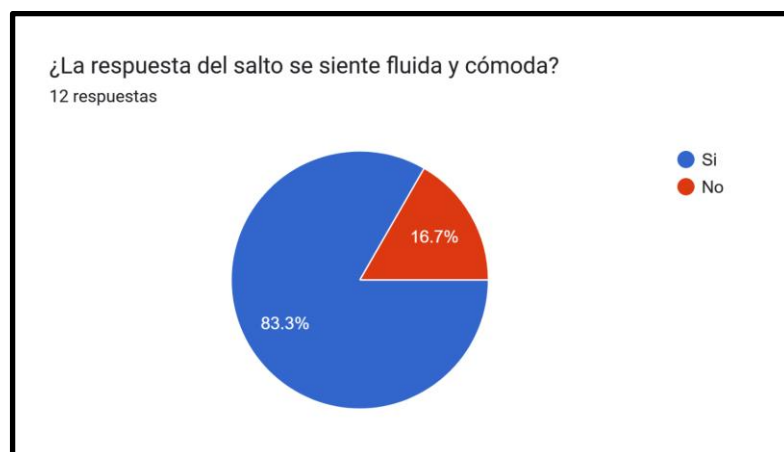


Ilustración 161: ¿La respuesta del salto se siente fluida y cómoda?

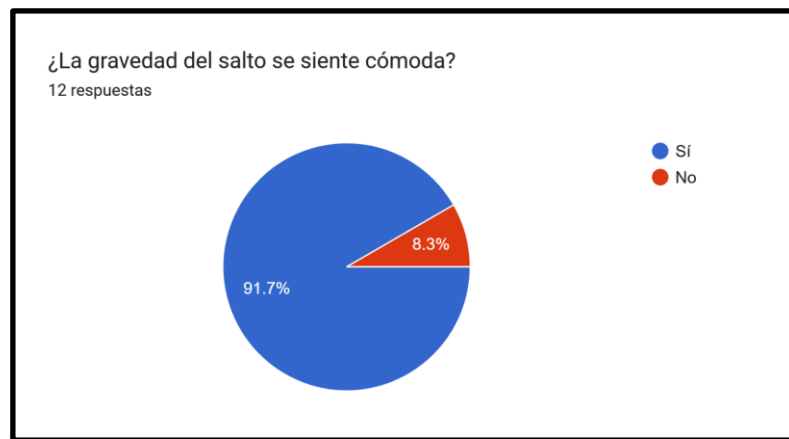


Ilustración 162: ¿La gravedad del salto se siente cómoda?

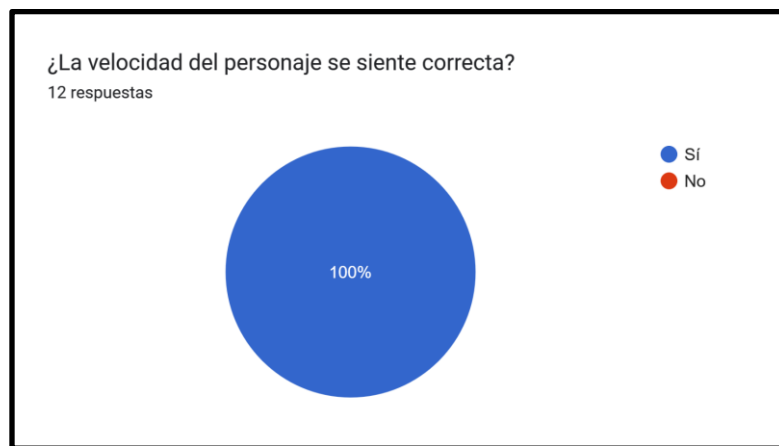


Ilustración 163: ¿La velocidad del personaje se siente correcta?

Entre los resultados obtenidos, los participantes ofrecieron críticas constructivas, señalando aspectos mejorables. Una de las sugerencias destacadas fue la introducción de una ventana de inicio para explicar el juego. Los jugadores expresaron su disposición a comprometerse con la experiencia y manifestaron que apreciarían más información inicial para comprender mejor el contexto del juego.

Mecánicas del Videojuego:

¿La respuesta del salto doble se siente fluida?

12 respuestas

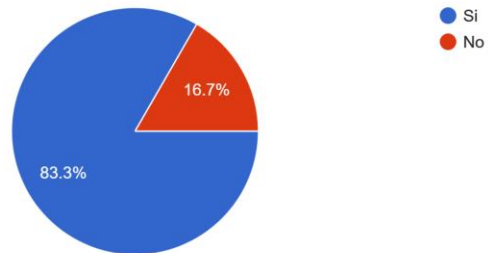


Ilustración 164: ¿La respuesta del salto doble se siente fluida?

¿La respuesta del salto coyote se siente correcta?

12 respuestas

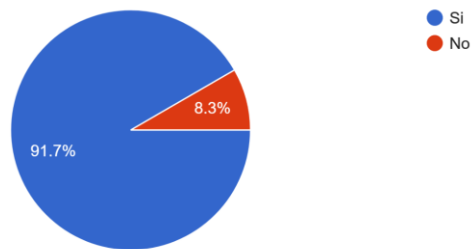


Ilustración 165: ¿La respuesta del salto coyote se siente correcta?

¿La respuesta del impulso o dash se siente correcta?

12 respuestas

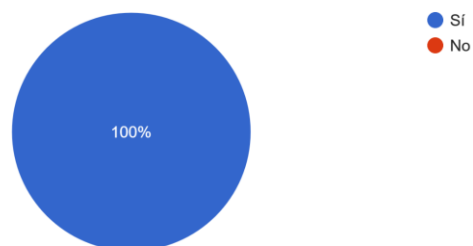


Ilustración 166: ¿La respuesta del impulso o dash se siente correcta?

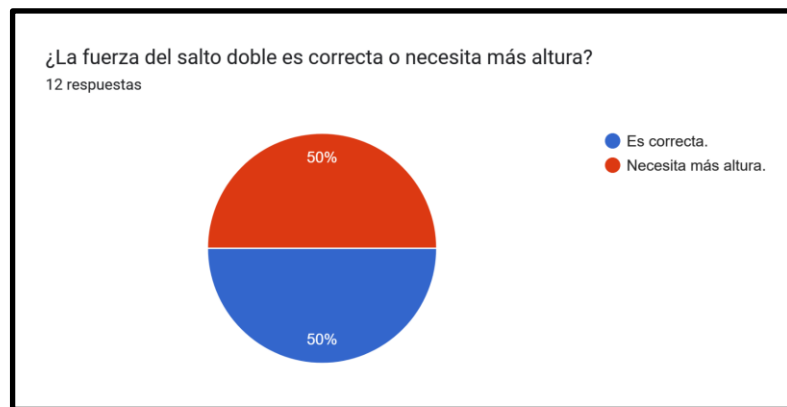


Ilustración 167: ¿La fuerza del salto doble es correcta o necesita más altura?

En relación con las mecánicas del juego, los participantes sugirieron la necesidad de revisar y ajustar la mecánica de salto. Las mejoras propuestas incluyen una respuesta de salto más ágil, corrección de problemas con el hitbox, ajustes en la altura del salto y coherencia en la aplicación del doble salto. Estas recomendaciones tienen como objetivo proporcionar una experiencia de juego más fluida y satisfactoria para los usuarios.

Interfaz, Diseño de Niveles y Sonidos del Juego:

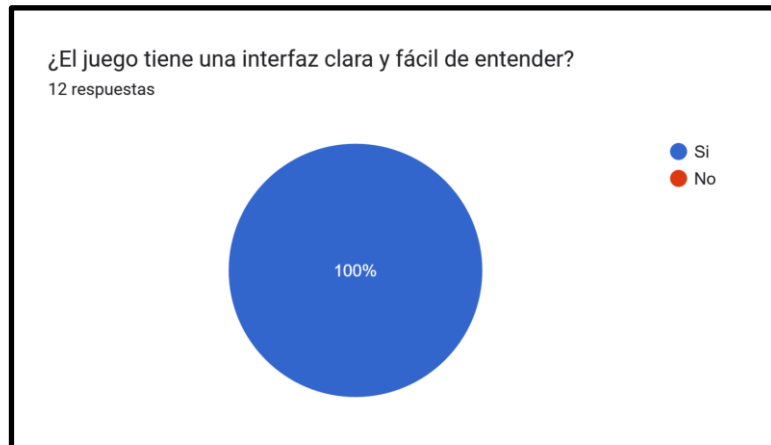


Ilustración 168: ¿El juego tiene una interfaz clara y fácil de entender?

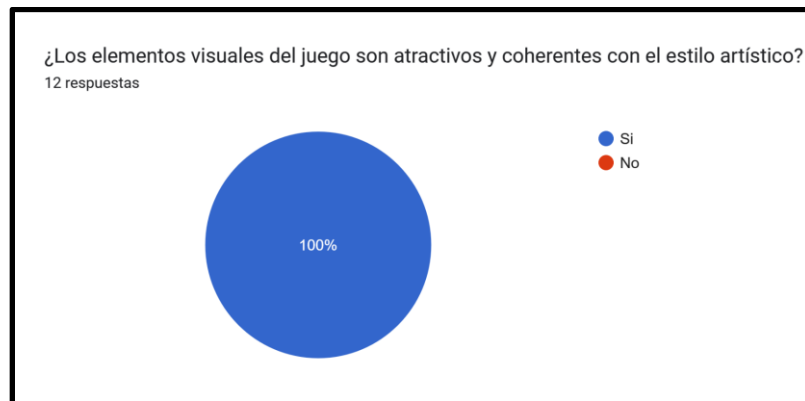


Ilustración 169: ¿Los elementos visuales del juego son atractivos y coherentes con el estilo artístico?



Ilustración 170: ¿El tutorial o instrucciones del juego son claros y te ayudan a comprender las mecánicas?

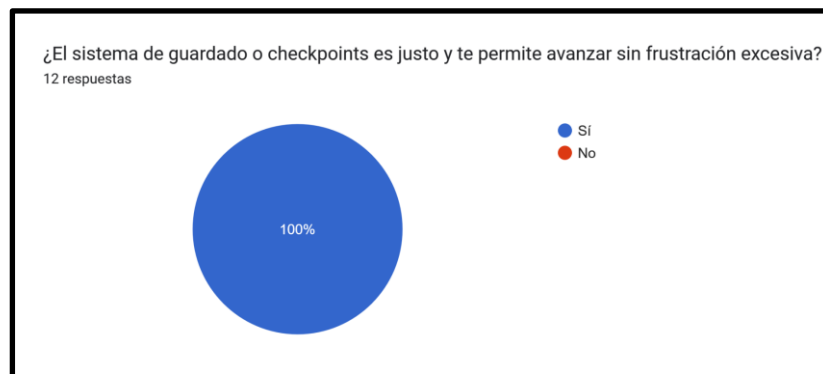


Ilustración 171: ¿El sistema de guardado o checkpoints es justo y te permite avanzar sin frustración excesiva?

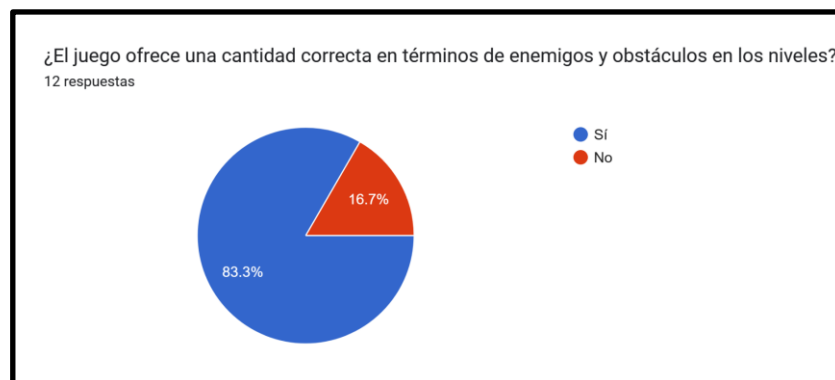


Ilustración 172: ¿El juego ofrece una cantidad correcta en términos de enemigos y obstáculos en los niveles?

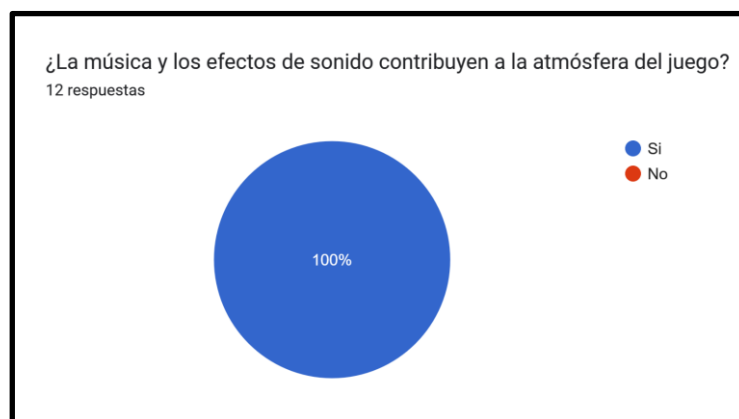


Ilustración 173: ¿La música y los efectos de sonido contribuyen a la atmósfera del juego?

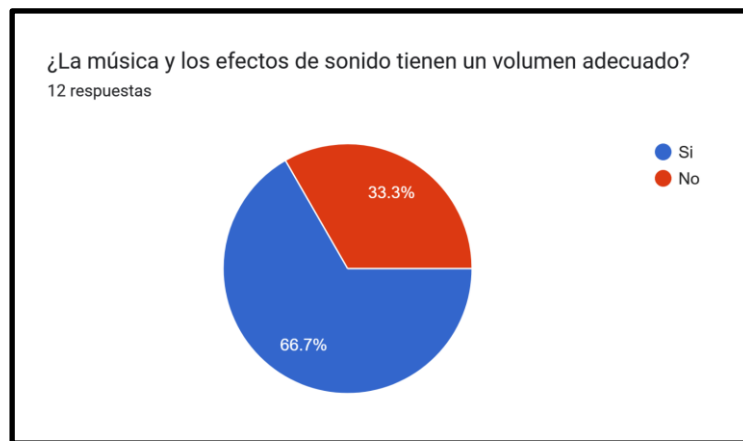


Ilustración 174: ¿La música y los efectos de sonido tienen un volumen adecuado?

En la última categoría de evaluación, los participantes resaltaron la importancia de ajustar el balance del juego en términos de elementos de recuperación, mejorar la visibilidad de obstáculos, considerar las solicitudes específicas de los jugadores y ajustar la dificultad. Estas sugerencias buscan proporcionar una experiencia más atractiva y desafiante, optimizando la interfaz, el diseño de niveles y los aspectos sonoros del juego.

La evaluación UX del juego *Star Forest* proporciona valiosa información para el proceso de diseño. Las sugerencias y críticas constructivas de los participantes han destacado áreas específicas que requieren atención y mejora. La implementación de cambios propuestos en las mecánicas de juego, la introducción de información inicial detallada y la optimización de la interfaz y el diseño de niveles pueden contribuir significativamente a mejorar la satisfacción y la experiencia general de los jugadores. Este tipo de análisis exhaustivo ayuda al desarrollo de los videojuegos en general, ya que asegura una experiencia de usuario más atractiva y gratificante.

Capítulo 6: Conclusión

Esta tesis ha sido concebida con el propósito de facilitar la incorporación de programadores al mundo del desarrollo de juegos a través de Unity, destacando la accesibilidad y simplicidad inherentes a esta plataforma. Se han explorado los diversos elementos que componen un videojuego de plataformas 2D, así como las funciones y herramientas de Unity, ilustrando el proceso de desarrollo y orientando a los programadores en el uso del motor de juegos utilizado en este proyecto. La exhibición de las diversas herramientas para crear un juego en 2D, especialmente para aquellos con habilidades de programación previas, abre la puerta a nuevas posibilidades y fomenta la autoexploración en este campo.

La plataforma Unity, respaldada por su tienda de recursos, destaca como una herramienta valiosa para los desarrolladores. La amplia variedad de activos gratuitos permite ahorrar tiempo y esfuerzo en la creación de juegos, y aunque existen opciones de pago, la abundancia de recursos gratuitos reduce la necesidad de invertir en activos adicionales. La sólida comunidad de soporte de Unity⁸ proporciona a los desarrolladores un recurso valioso cuando enfrentan desafíos durante el proceso de desarrollo. Esta red de apoyo contribuye a la superación de obstáculos y fomenta el intercambio de conocimientos entre la comunidad de desarrolladores de Unity.

La versatilidad de Unity como motor de juegos se destaca por su capacidad para la publicación en múltiples plataformas, ofreciendo a los desarrolladores la flexibilidad de presentar sus creaciones en diversos entornos. Esta característica no solo amplía la visibilidad del juego, sino que también potencia sus posibilidades de éxito y generación de ingresos. La amplia aceptación de Unity en la industria del desarrollo de videojuegos subraya su utilidad y la libertad que brinda a los creadores. Aunque las grandes empresas pueden contar con recursos para desarrollar sus propios motores de juego, la elección de Unity, un motor de juego ya excelente y gratuito, resulta inigualable en un momento en que la industria de los videojuegos experimenta un auge significativo, siendo una opción destacada para individuos que buscan materializar sus propias visiones y proyectos independientes en la industria de los videojuegos.

⁸ Disponible en el foro oficial (<https://forum.unity.com/>)

Referencias

[1] Concepto Definición (s/f) *Videojuegos*. Conceptodefinicion.de.

<https://conceptodefinicion.de/videojuegos/>

[2] CNN (2013), *8 memorable quotes from Nintendo president Satoru Iwata*. Edition.cnn.com.

<https://edition.cnn.com/2015/07/13/asia/gallery/japan-satoru-iwata-quotes/index.html>

[3] WeForum.org (2022), *Gaming is booming and is expected to keep growing*. Weforum.org

<https://www.weforum.org/agenda/2022/07/gaming-pandemic-lockdowns-pwc-growth/>

[4] Wikipedia (s/f) *Videojuego Independiente*. Wikipedia.org. Consultado el 29 de agosto de 2022.

https://es.wikipedia.org/wiki/Videojuego_independiente

[5] Gamespot (2022), *Every Game Released in 2022*. Gamespot.com

<https://www.gamespot.com/articles/every-game-released-in-2022/1100-6499287/>

[6] IGN (2022), *The Game Awards 2022: Everything Announced*. Ign.com

<https://www.ign.com/articles/the-game-awards-2022-everything-announced>

[7] Tarreo (2018), *Estos son los juegos que llegarán a PC, Xbox y Playstation en Julio*.

Tarreo.com

<https://www.tarreo.com/noticias/485116/Estos-son-los-juegos-que-llegaran-a-PC-Xbox-y-Playstation-en-julio>

[8] Videojuegos Fandom (s/f) *Videojuegos de plataformas*. Videojuegos.Fandom.com.

https://videojuegos.fandom.com/es/wiki/Videojuego_de_plataformas

[9] Unity (s/f) *Crear juegos 2D y 3D en Unity*. Unity.com.

<https://unity.com/es/how-to/difference-between-2D-and-3D-games>

[10] Retrogame Deconstruction Zone (2020) *What was the first platformer, Space Panic or Donkey Kong?* RetroGameDeconstructionZone.com. Consultado el 30 de agosto de 2022.

<https://www.retrogamedeconstructionzone.com/2020/07/what-was-first-platformer-space-panic.html>

[11] Begy, J. (2010) *The History and Significance of Jumping in Games*. Jason Begy.

[12] Giant Bomb (s/f) *Scott Rogers*. GiantBomb.com. Consultado el 09 de septiembre de 2022.

<https://www.giantbomb.com/scott-rogers/3040-36702/credits/>

[13] Rogers, S. (2010) *Level Up! The guide to great video game design*. John Wiley & Sons, Ltd, p.126.

[14] Tokio School (2020) *Mecánicas de juego más habituales en los videojuegos*. Tokioschool.com.

<https://www.tokioschool.com/noticias/mecanicas-de-juego-habituales-en-videojuegos/>

[15] Kotaku (2009) *Let's talk about jumping*. Kotaku.com.

<https://kotaku.com/lets-talk-about-jumping-5420545>

[16] Taylor, D. (2017) *14 Hidden Video Game Design Tricks That Will Blow Your Mind*. HowStuffWorks.com.

<https://electronics.howstuffworks.com/14-hidden-video-game-design-tricks-that-will-blow-your-mind.htm>

[17] Manthorp, R. (2019) *Flynn: Advanced Jump Mechanics*. GameMaker.io.

<https://gamemaker.io/es/blog/flynn-advanced-jump-mechanics>

[18] De, S. (2021) *What Are Hitboxes in Gaming?*. MakeUseOf.com.

<https://www.makeuseof.com/hitboxes-gaming/>

[19] Markovic, I. (s/f) *Why Giving Instant Feedback is Important for Effective Learning*. Edume.com.

<https://www.edume.com/blog/role-of-feedback-in-improving-learning>

[20] Star Loop Studios (s/f) *Game UX Design: Why UX Matters When Making A Video Game*. StarLoopStudios.com. Consultado el 26 de diciembre de 2022.

<https://starloopstudios.com/game-ux-design-why-ux-matters-when-making-a-video-game/>

[21] Yu-Kai Chou (2018) *4 Experience Phases in Gamification – Phase 2: The Onboarding Phase*. Yukaichou.com. Consultado el 13 de octubre de 2023.

<https://yukaichou.com/gamification-study/4-experience-phases-gamification-2-onboarding-phase/>

[22] HeadSpin (2022) *Why is Usability Crucial for a Good Gaming Experience?*. Headspin.io. Consultado el 14 de octubre de 2023

<https://www.headspin.io/blog/usability-for-good-gaming-experience>

[23] Shelstad, W. J., Chaparro, B. S., & Keebler, J. R. (2019, November). Assessing the user experience of video games: Relationships between three scales. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* (Vol. 63, No. 1, pp. 1488-1492). Sage CA: Los Angeles, CA: SAGE Publications.

[24] Davidson, S. (2018). *A Multi-dimensional model of enjoyment: Development and validation of an enjoyment scale (ENJOY)*. Embry-Riddle Aeronautical University.

[25] Phan, M. H., Keebler, J. R., & Chaparro, B. S. (2016). The development and validation of the game user experience satisfaction scale (GUESS). *Human factors*, 58(8), 1217-1247.

[26] Schrepp, M., Thomaschewski, J., & Hinderks, A. (2017). Construction of a benchmark for the user experience questionnaire (UEQ).

[27] Norman, K. L. (2013). Geq (game engagement/experience questionnaire): a review of two papers. *Interacting with computers*, 25(4), 278-283.

[28] León Yacelga, Andrés Roberto, Acosta Espinoza, Jorge Lenin, & Díaz Vásquez, Rita

Azucena. (2021). Aplicación de la metodología incremental en el desarrollo de sistemas de información. *Revista Universidad y Sociedad*, 13(5), 175-182. Epub 02 de octubre de 2021.

[29] Isbister, K., & Hodent, C. (Eds.). (2022). *Game Usability: Advice from the Experts for Advancing UX Strategy and Practice in Videogames*. CRC Press.

[30] Martínez, G. (2023). *¿Qué es Unity y para qué sirve?* Ebac.mx.

<https://ebac.mx/blog/que-es-unity-y-para-que-sirve>