



RECREACIÓN DE SUPER MARIO BROS EN UNITY

2D

GRAFICACIÓN

Luis Yael
Méndez
Sánchez

BUAP FCC
Primavera
2025

666

INTEGRANTES



JOSE EMILIANO FLORES
FLORES



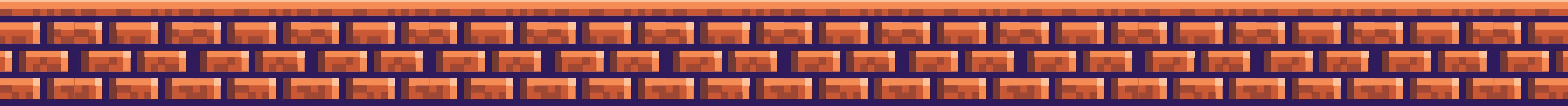
URIEL LEMUS PÉREZ



SANTIAGO SILVERIO FLORES



MAX RAMOS MARTINEZ



INTRODUCCIÓN

En este proyecto nos propusimos recrear una versión básica del clásico Super Mario Bros. utilizando el motor de videojuegos Unity. Para ello, nos guiamos por una serie de video tutoriales que explican paso a paso cómo construir un juego de plataformas 2D desde cero.

A lo largo del desarrollo, aprendimos a implementar elementos fundamentales del género como el movimiento del personaje, la física del salto, la detección de colisiones, diseño de niveles, e interacción con objetos del entorno.

Nuestro objetivo no solo fue replicar una experiencia similar a la del juego original, sino también comprender y aplicar conceptos clave del desarrollo de videojuegos en Unity, como el uso de sprites, animaciones, controladores de física y scripts en C#.

METODOLOGÍA:

Para el desarrollo de este proyecto, seguimos una metodología basada en el aprendizaje práctico guiado. Utilizamos una serie de video tutoriales que nos guiaron paso a paso en la construcción de un videojuego de plataformas en 2D, inspirado en el clásico Super Mario Bros. A lo largo del proceso, aplicamos los conocimientos adquiridos en programación con C# y diseño de juegos en Unity, implementando las mecánicas más representativas de este género: movimiento del personaje, detección de colisiones, física, interacción con objetos y enemigos, así como la creación de niveles.

El enfoque fue iterativo: desarrollábamos una funcionalidad por vez, la probábamos y luego pasábamos a la siguiente. Esto nos permitió entender mejor cómo se conectan entre sí los distintos sistemas dentro de un videojuego, como el motor de físicas, la lógica de scripts y el diseño visual.

PROPÓSITO DEL TRABAJO

El propósito principal de este proyecto fue adquirir experiencia práctica en el desarrollo de videojuegos utilizando Unity, entendiendo no solo la parte técnica, sino también el proceso creativo y lógico detrás de un juego funcional. Además, nos propusimos desarrollar habilidades de resolución de problemas, trabajo en equipo, y pensamiento estructurado, todo dentro de un contexto que simula el desarrollo de un producto real. El enfoque fue iterativo: desarrollábamos una funcionalidad por vez, la probábamos y luego pasábamos a la siguiente. Esto nos permitió entender mejor cómo se conectan entre sí los distintos sistemas dentro de un videojuego, como el motor de físicas, la lógica de scripts y el diseño visual.

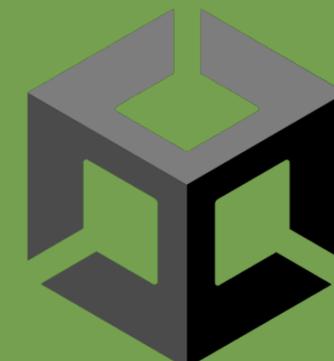


HERRAMIENTAS UTILIZADAS



Para este proyecto utilizamos Unity, un motor de desarrollo de videojuegos ampliamente reconocido por su versatilidad en la creación de juegos tanto en 2D como en 3D. La versión específica que usamos fue [Unity 2021.3.45f1](#), que forma parte de la rama LTS (Long Term Support). Esta versión está diseñada para ofrecer estabilidad y soporte a largo plazo, lo que la hace ideal para proyectos educativos y de producción.

Unity nos permitió trabajar con herramientas específicas para juegos en 2D, como Tilemaps, animaciones basadas en sprites, sistemas de físicas 2D, y un entorno visual amigable para diseñar niveles e interfaces. Además, el motor se integra perfectamente con C#, el lenguaje en el que programamos todas las funcionalidades del juego, desde el movimiento del personaje hasta la interacción con el entorno.



Unity

LENGUAJE DE PROGRAMACIÓN C#

Para programar las funcionalidades del juego utilizamos **C# (C Sharp)**, el lenguaje oficial de scripting en Unity.

En el contexto de Unity, C# se usa para controlar todos los aspectos del comportamiento del juego: desde el movimiento del personaje, interacciones con enemigos y objetos, hasta el manejo de la interfaz del usuario y la lógica de puntuación, tiempo y vidas.

Durante el desarrollo del juego, usamos C# para:

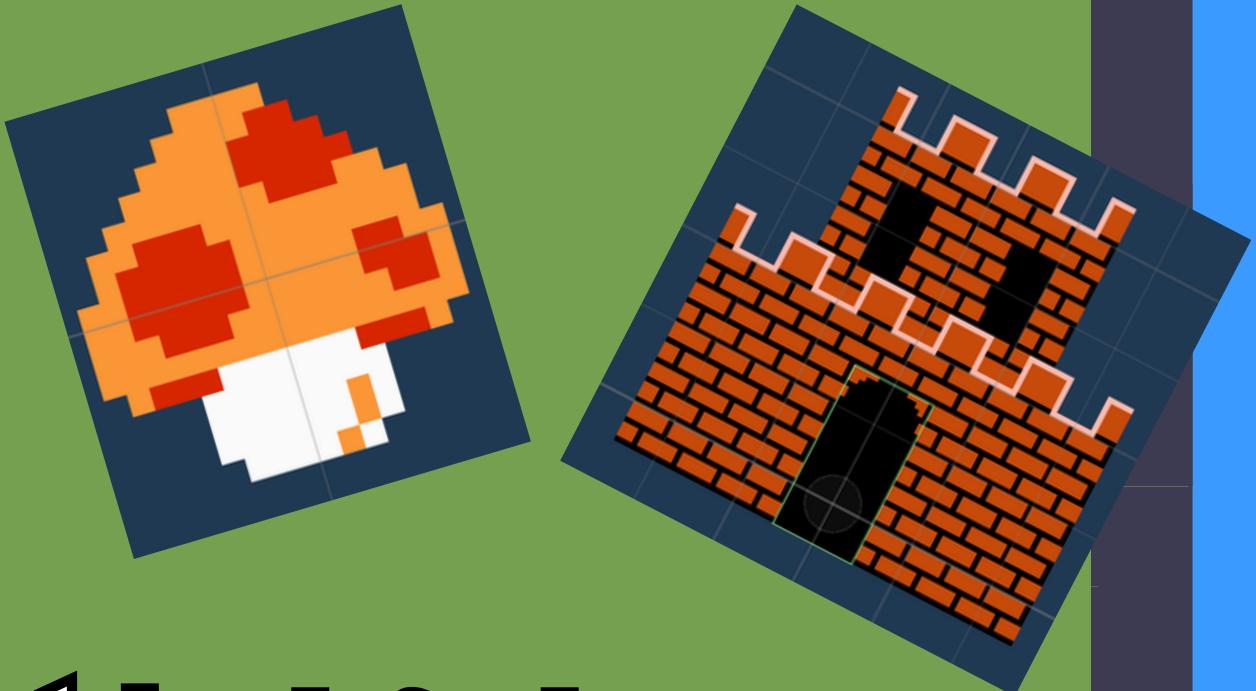
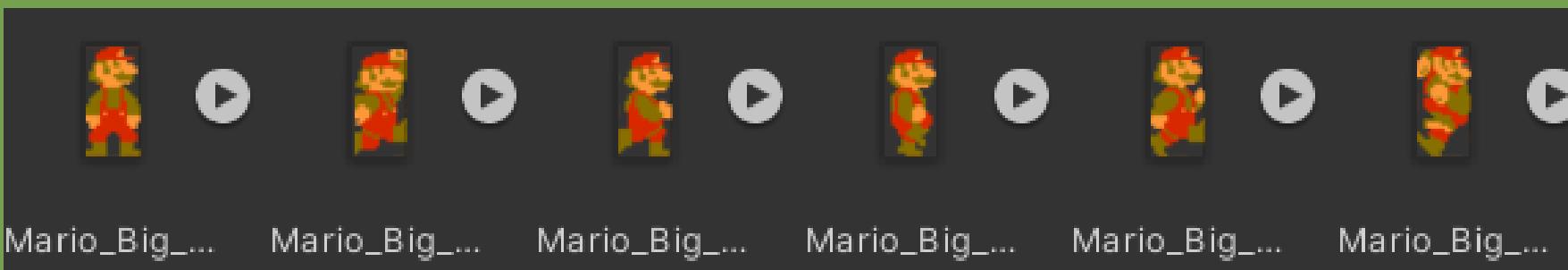
- Detectar colisiones y aplicar físicas.
- Crear animaciones controladas por código.
- Administrar estados del personaje (caminar, saltar, morir).
- Implementar elementos del HUD (tiempo, monedas, puntaje).
- Gestionar la lógica del Menú y la escena de **GAME OVER**.



OTROS RECURSOS UTILIZADOS

Para enriquecer la experiencia visual y sonora del juego,
incorporamos:

- **Sprites:** Personajes, enemigos, bloques, fondos.



- **Sonidos:** Efectos de salto, monedas, colisiones y música de fondo.
hasta la interacción con el entorno.



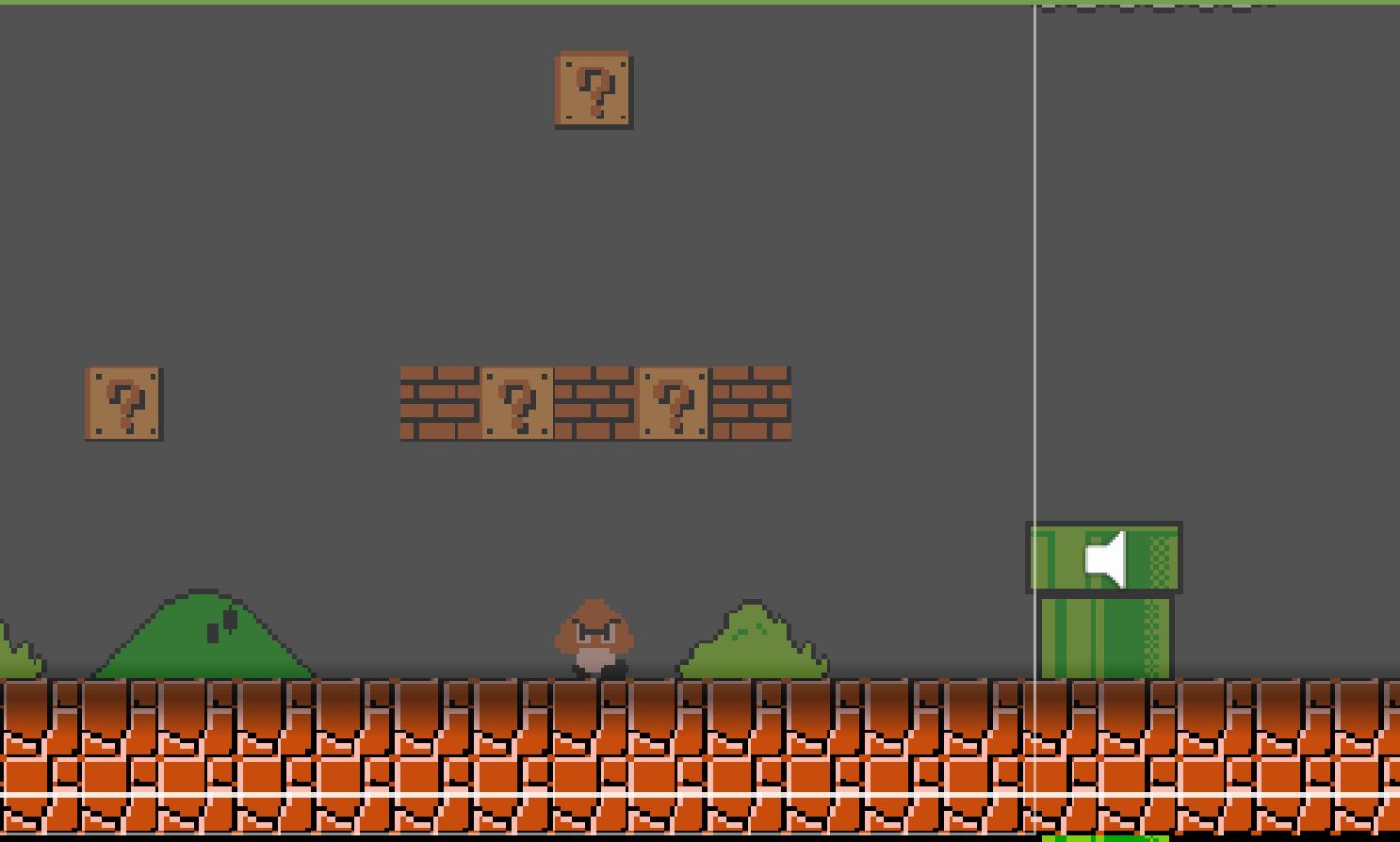


ETAPAS DEL DESARROLLO

DISEÑO DEL ESCENARIO

El nivel del juego se diseñó utilizando la herramienta de **Tilemap** de Unity, que permite construir escenarios en 2D de forma modular, rápida y precisa. Esta herramienta nos facilitó colocar bloques, plataformas y elementos decorativos en una cuadrícula, como si se tratara de un rompecabezas.

Cada bloque del entorno (suelo, ladrillos, tuberías, etc.) se creó a partir de tiles o piezas gráficas reutilizables, lo que permitió mantener consistencia visual y optimizar el rendimiento del juego.



MOVIMIENTO DEL PERSONAJE

El movimiento del personaje fue implementado de forma fluida y responsiva, utilizando un script en C# que permite desplazarse lateralmente, saltar y caer con una sensación realista. Se usaron componentes de física 2D como Rigidbody2D y Collider2D, junto con controles manuales de gravedad y velocidad para lograr un mejor control del salto y la caída.

Características principales:

- **Desplazamiento lateral:** basado en la entrada horizontal del jugador, con suavizado progresivo en la aceleración.
- **Salto controlado:** con altura y duración configurables, usando fórmulas físicas reales para una curva de salto natural.
- **Caída acelerada:** el jugador cae más rápido al soltar el botón de salto, lo que mejora el game feel.
- **Rebote sobre enemigos:** si el jugador cae sobre un enemigo, rebota automáticamente.
- **Detección de colisiones:** ajustada para identificar impactos desde arriba, abajo o los lados.
- **Sonido dinámico:** incluye sonido de salto, colisión con bloques y enemigos.

AJUSTES DE GAME FEEL:

- Se aplicó doble gravedad al caer para hacer el salto más rápido al descender, logrando una sensación más precisa y satisfactoria.
- Se detecta si el jugador cambia de dirección rápidamente (**sliding**) y se ajusta la animación o el control en consecuencia.
- Se limitó el movimiento horizontal para que el jugador no pueda salir del marco visible de la cámara.

```
0 references
private void Update()
{
    // Se procesa el movimiento horizontal basado en la entrada del jugador
    HorizontalMovement();

    // Se verifica si el jugador está tocando el suelo usando un raycast hacia abajo
    grounded = rb.Raycast(Vector2.down);

    // Si está en el suelo, se permite saltar y se ajusta la velocidad vertical
    if (grounded)
    {
        GroundedMovement();
    }

    // Se aplica la gravedad manualmente para controlar mejor el salto y caída
    ApplyGravity();

    // Control del sonido de pasos: se reproduce si está corriendo y en el suelo
    if (grounded && running && !audioSteps.isPlaying)
    {
        if (pasosClip != null)
        {
            audioSteps.clip = pasosClip;
            audioSteps.Play();
        }
    }
    // Se detiene el sonido de pasos si no está corriendo o no está en el suelo
    else if (!(running || !grounded) && audioSteps.isPlaying)
    {
        audioSteps.Stop();
    }
}
```

MOVIMIENTO DEL PERSONAJE

El movimiento del personaje fue implementado de forma fluida y responsiva, utilizando un script en C# que permite desplazarse lateralmente, saltar y caer con una sensación realista. Se usaron componentes de física 2D como Rigidbody2D y Collider2D, junto con controles manuales de gravedad y velocidad para lograr un mejor control del salto y la caída.

Características principales:

- **Desplazamiento lateral:** basado en la entrada horizontal del jugador, con suavizado progresivo en la aceleración.
- **Salto controlado:** con altura y duración configurables, usando fórmulas físicas reales para una curva de salto natural.
- **Caída acelerada:** el jugador cae más rápido al soltar el botón de salto, lo que mejora el game feel.
- **Rebote sobre enemigos:** si el jugador cae sobre un enemigo, rebota automáticamente.
- **Detección de colisiones:** ajustada para identificar impactos desde arriba, abajo o los lados.
- **Sonido dinámico:** incluye sonido de salto, colisión con bloques y enemigos.

COLISIÓN CON ENEMIGOS BLOQUES

Se detecta en `PlayerMovement.OnCollisionEnter2D(Collision2D collision)` comprobando la capa del objeto colisionado (`collision.gameObject.layer`).

Enemigos

- Si la colisión viene de arriba (`transform.DotTest(..., Vector2.down)`), el jugador rebota (se le da la mitad de la fuerza de salto) y suena `golpeEnemigoClip`.
- Si la colisión no es desde arriba, se reproduce `muerteClip`. (Aquí podrías llamar a `GameManager.Instance.ResetLevel()` para gestionar la vida y el reinicio.)

Bloques

- Si el jugador golpea un bloque desde abajo (`transform.DotTest(..., Vector2.up)`), se anula su velocidad vertical y se reproduce `golpeBloqueClip`.
- Las colisiones laterales quedan preparadas para lógica adicional (p. ej. detener el avance si toca una pared).

CAPTURA DE CÓDIGO:

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    int otherLayer = collision.gameObject.layer;

    // Si colisiona con un enemigo
    if (otherLayer == LayerMask.NameToLayer("Enemy"))
    {
        // Si cae sobre el enemigo desde arriba
        if (transform.DotTest(collision.transform, Vector2.down))
        {
            velocity.y = jumpForce / 2f; // Rebote al saltar sobre enemigo
            jumping = true;

            // Se reproduce sonido de golpe a enemigo
            if (golpeEnemigoClip != null)
                audioSource.PlayOneShot(golpeEnemigoClip);
        }
        else
        {
            // Si colisiona con enemigo desde otro ángulo, se reproduce sonido de muerte
            if (muerteClip != null)
                audioSource.PlayOneShot(muerteClip);
        }
    }
    // Si colisiona con otros objetos que no sean power-ups
    else if (otherLayer != LayerMask.NameToLayer("PowerUp"))
    {
        // Si golpea un bloque desde abajo
        if (transform.DotTest(collision.transform, Vector2.up))
        {
            velocity.y = 0f; // Se detiene la velocidad vertical
        }
    }
}
```

RECOLECCIÓN DE MONEDAS O ITEMS

Toda la lógica de puntuación y actualización de HUD está centralizada en
GameManager:

Monedas

- GameManager.AddCoin() incrementa el contador, reproduce coinClip y, al llegar a 100 monedas, resetea el contador y llama a AddLife() para otorgar vida extra.
- CollectCoinItem() suma 100 pts además de llamar a AddCoin().

Power-ups e ítems especiales

- CollectPowerUp() suma 1 000 pts y puede invocar AddHongoNormal(), AddStar(), etc.
- Collect1UpGreen() usa AddLife() para dar una vida extra.
- AddStar(), AddHongoNormal() y AddHongoEspecial() reproducen sus respectivos clips y actualizan UI; estrella activa un estado de invencibilidad temporal.

CAPTURA DE CÓDIGO:

```
0 references
public void CollectCoinItem()
{
    coins++;
    AddScore(200);
    AddCoin(); // Maneja sonido, vidas y HUD
}

/// <summary>
/// Recoleistar un power-up.
/// Añade puntos y puede llamar a métodos para efectos especiales.
/// </summary>
0 references
public void CollectPowerUp()
{
    AddScore(1000);
    // Aquí puedes llamar AddHongoNormal(), AddStar(), etc.
}

/// <summary>
/// Recoleistar un 1-Up verde (vida extra).
/// </summary>
0 references
public void Collect1UpGreen()
{
    AddLife(); // No suma puntos, solo vida extra
}

/// <summary>
/// Recoleistar un hongo secreto que permite saltos especiales.
/// </summary>
0 references
```

MECÁNICAS DE DAÑO Y MUERTE

La vida y el reinicio de nivel están en GameManager:

- **Perder vida:** ResetLevel() decrementa lives y recarga la escena; si lives == 0, llama a GameOver().
- **Game Over:** pausa el juego (Time.timeScale = 0), muestra el panel gameOverPanel, reproduce gameOverClip en bucle y, tras 5 s, reinicia la partida entera.
- Reproducir efectos de sonido de daño o muerte se desacopla del manejo de vidas, facilitando que la capa de presentación (PlayerMovement) solo “notifique” y que GameManager decida la lógica de estado global.



ELEMENTOS DE UI Y HUI

- **Marcadores clave:** Puntuación, monedas y vidas se muestran siempre actualizados al instante, permitiendo al jugador evaluar rápidamente su estado y decidir si arriesgarse a explorar más o conservar recursos.
- **Pausa interactiva:** Al pulsar P, se detiene todo el juego y aparece un menú centrado con opciones para reanudar o salir, asegurando que al volver la partida quede exactamente donde se dejó.
- **Game Over suave:** Al agotar las vidas, se muestra un panel de fin de juego con efecto sonoro y una breve pausa, para luego reiniciar automáticamente el nivel, ofreciendo feedback claro y fluido.



CAPTURA DE CÓDIGO:

```
public class PauseManager : MonoBehaviour
{
    [Header("Panel raíz del menú de pausa")]
    // Referencia al objeto UI que contiene el menú de pausa
    4 references
    public GameObject pauseMenuUI;

    // Variable que indica si el juego está actualmente pausado
    3 references
    private bool isPaused = false;

    /// <summary>
    /// Método llamado cada frame por Unity.
    /// Detecta la pulsación de teclas para pausar, reanudar o salir del juego.
    /// </summary>
    0 references
    private void Update()
    {
        // Si se presiona la tecla P, alterna entre pausar y reanudar
        if (Input.GetKeyDown(KeyCode.P))
        {
            if (isPaused)
                Resume(); // Reanuda el juego si está pausado
            else
                Pause(); // Pausa el juego si está activo
        }

        // Si se presiona la tecla Escape, se sale del juego
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            ExitGame();
        }
    }
}
```

PULIDO FINAL

- **Animaciones suaves:** Afinar transiciones y añadir pequeños “anticipos” para que cada salto, carrera o impacto se sienta natural.
- **Feedback audiovisual:** Animaciones y efectos sonoros en saltos, golpes y recolección de objetos para reforzar cada acción.
- **Optimización:** Reducir draw calls con sprite atlases y ajustar físicas para mantener 60 FPS constantes.
- **Balance:** Ajustar la velocidad de enemigos, la colocación de ítems y los tiempos de invencibilidad para un reto justo.
- **Pruebas y corrección:** Sesiones de playtest para eliminar glitches, colisiones erróneas y asegurar una experiencia pulida.

OFICULTADES ENFRENTADAS

- **Menú de pausa complejo:** fue lo más difícil. Los botones no respondían y teníamos dos Canvas superpuestos que interferían entre sí.
 - **Solución:** Optamos por hacer el menú mediante teclas y ya no tanto por botones.
-
- **Actualización de puntuación inconsistente:** cada acción (moneda, enemigo, ítem) disparaba métodos en lugares distintos, provocando retrasos o valores erróneos.
 - **Solución:** Centralizamos todo en un único método de “añadir puntos” que modifica la variable y refresca el HUD al instante, garantizando que cada acción se refleje correctamente en pantalla.

CONCLUSIÓN

Desarrollar este proyecto de Mario Bros en Unity nos permitió integrar y aplicar diversos conceptos fundamentales de la programación orientada a objetos, el desarrollo de videojuegos y la arquitectura de software en C#. Entendimos la importancia del patrón Singleton para controlar el flujo global del juego de forma centralizada, evitando duplicación de instancias y facilitando el acceso a datos clave como vidas, puntuación y progreso del jugador. Además, trabajar con corutinas, manejo de escenas, y sistemas de audio nos ayudó a comprender mejor cómo se sincronizan eventos y acciones en un entorno de juego interactivo. También nos enfrentamos a desafíos técnicos reales como la gestión del HUD dinámico, la reutilización de componentes, y la lógica de progresión entre niveles, lo que fortaleció nuestra capacidad para resolver problemas y estructurar código limpio y mantenible.

EJECUCIÓN



