UNIVERSIDAD INTERNACIONAL DE LA RIOJA

MÁSTER EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

“Actividad 1. Space Shooter completo”

Alex López

Docentes: Fernando Madrid Recio

Índice

1. Introducción

Este proyecto consiste en el desarrollo de un videojuego tipo "space shooter" en 2D, utilizando Unity y C#.

El objetivo principal es que el jugador controla una nave, elimine oleadas de enemigos y sobreviva recolectando ítems y utilizando poderes, durante el desarrollo se aplicaron conceptos fundamentales del motor Unity, como control de físicas, inputs, colisiones, UI, animaciones básicas y generación procedural de enemigos y objetos.

2. Enlaces

Demo jugable en Itch.io:
https://frobenyus.itch.io/spaceshooter

Repositorio del proyecto (GitHub):
https://github.com/AlexRose97/SpaceShooter

3. Listado de implementaciones extra:

1. Menú principal completo: pantalla de inicio, información, rankings y configuración.
2. Sistema de dificultad: selector entre Fácil, Moderado y Difícil que afecta cantidad de enemigos, oleadas y drop de ítems.
3. Guardado de puntuaciones con JSON: se guarda nombre, puntuación y fecha de cada partida.
4. Visualización de rankings: visualización del JSON de las partidas guardadas.
5. Controles táctiles para móvil: botones de movimiento y disparo funcionales al ejecutar en navegador desde el teléfono.
6. Poderes e ítems: ítems aleatorios con efectos como vida extra
7. Música de fondo y efectos de sonido: integrados y controlados desde scripts.

4. Desarrollo del proyecto:

El desarrollo se dividió en las siguientes etapas:

Mecánicas base (Player.cs):

Movimiento con teclado táctil

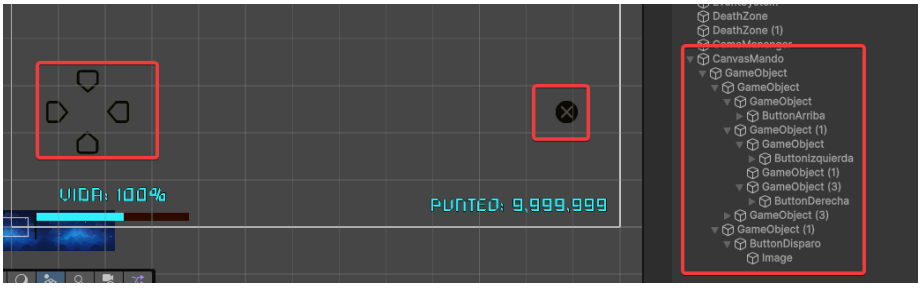
En esta función se encuentra la lógica para realizar el movimiento de la nave(player).

```
/// <summary>
/// Funcion para realizar el movimiento del player, con AWSD o Touch
/// </summary>
Frequently called  1 usage  AlexRose97
void Movimiento()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    // Si hay input táctil, lo usamos en vez del teclado
    if (_inputTouch != Vector2.zero)
    {
        horizontal = _inputTouch.x;
        vertical = _inputTouch.y;
    }
    Vector2 direccion = new Vector2(horizontal, vertical).normalized;
    transform.Translate(translation: direccion * (moveSpeed * Time.deltaTime));
}
```

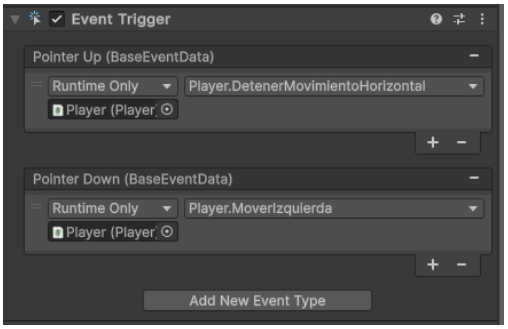
Además se agrega una función que evita que el jugador se salga del espacio delimitado en pantalla utilizando un “Clamp” que valida los límites permitidos.

```
Frequently called  1 usage  AlexRose97
void DelimintarMovimiento()
{
    /*** Delimitar movimiento en la pantalla ***/
    float xClamp = Mathf.Clamp(transform.position.x, min: -8f, max: 8f);
    float yClamp = Mathf.Clamp(transform.position.y, min: -3.2f, max: 3.2f);
    transform.position = new Vector3(xClamp, yClamp, transform.position.z);
}
```

Para soporte móvil (touch) se agrega un canvas con los botones necesarios a mostrar en esta plataforma.



y a cada botón se le agregan los triggers necesarios para ejecutar las acciones de movimiento o disparo, por ejemplo:




```
public void MoverArriba()    { _inputTouch.y =  1; }
1 asset usage  AlexRose97
public void MoverAbajo()     { _inputTouch.y = -1; }
1 asset usage  AlexRose97
public void MoverIzquierda() { _inputTouch.x = -1; }
1 asset usage  AlexRose97
public void MoverDerecha()   { _inputTouch.x =  1; }
2 asset usages  AlexRose97
public void DetenerMovimientoHorizontal() { _inputTouch.x = 0; }
2 asset usages  AlexRose97
public void DetenerMovimientoVertical()   { _inputTouch.y = 0; }
```

Disparos

Se agrega la siguiente función que nos permite generar los proyectiles de la nave del jugador.

```
private void IntentarDisparo()
{
    if (_timer > ratioBullet)
    {
        Instantiate(bulletPrefab, spawnPoint1.transform.position, Quaternion.identity);
        Instantiate(bulletPrefab, spawnPoint2.transform.position, Quaternion.identity);
        ReproduceSound(audioBullet);
        _timer = 0;
    }
}
```

donde dichas función necesita de las posiciones y configuraciones para la nave.

```
[Header("Player Info")]
[SerializeField] private float moveSpeed; //velocidad de movimiento  "5"
[SerializeField] private float ratioBullet; //frecuencia de los disparos  "0.5"
[Header("Prefab Bala")]
[SerializeField] private GameObject bulletPrefab; //UI disparo  DisparoPlayer
[SerializeField] private GameObject spawnPoint1; //posicion bala1  SpawnPoint1
[SerializeField] private GameObject spawnPoint2; //posicion bala2  SpawnPoint2
```

Además se crea una función ejecutada al presionar el teclado o al presionar el botón del UI.

```
public void DispararTouch()
{
    IntentarDisparo();
}
/// <summary>
/// Funcion para realizar el disparo desde el teclado
/// </summary>
Frequently called  1 usage  AlexRose97
void Disparar()
{
    _timer += 1 * Time.deltaTime; //contador de tiempo
    /*se realiza un disparo cuando se preciona la tecla "espacio solo si ya se cumplio el tiempo minimo"*/
    if (Input.GetKeyDown(KeyCode.Space) && _timer > ratioBullet)
    {
        IntentarDisparo();
    }
}
```

Colisiones

Se utiliza la función OnTriggerEnter2D que nos permite detectar interacciones con otros componentes, en esta función se agrega la logica para la colisión de proyectiles y enemigos, dado que cada uno tiene distintos comportamientos

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("BulletEnemy")){...}

    if (other.CompareTag("Enemy")){...}
}
```

una colisión con un proyectil le resta vida al jugador.

```
float damage = DamageConstants.GetDamage(other.tag);
TakeDamage(damage);
Destroy(other.gameObject);
ReproduceSound(audioImpact);
UpdateUIValues();
```

y una colisión con un enemigo le resta vida al jugador pero también le permite ganar puntos.

```
float damage = DamageConstants.GetDamage(enemy.tag, enemy.Nivel);
TakeDamage(damage);
AddPoints(enemy);
Destroy(other.gameObject);
ReproduceSound(audioImpact);
```

Reproducir sonidos

se crea la función que nos permite reproducir efectos de sonido "de un solo uso" sin necesidad de tener un AudioSource permanente, dado que sería complicado ya que las naves pueden destruirse y generar null pointers al tratar de generar el sonido.

```
void ReproduceSound(AudioClip clip)
{
    GameObject tempAudio = new GameObject(name:"TempAudio");
    tempAudio.transform.position = transform.position;

    AudioSource tempSource = tempAudio.AddComponent<AudioSource>();
    tempSource.clip = clip;
    tempSource.volume = 1.0f; // 🔊 volumen deseado
    tempSource.pitch = 1.0f;
    tempSource.spatialBlend = 0f; // 0 = 2D, 1 = 3D

    tempSource.Play();
    Destroy(tempAudio, clip.length); // eliminar al terminar
}
```

Daño y gestión de vida

La lógica de la vida en este juego se maneja de la siguiente forma:

1. Se tiene capacidad de 100 puntos de daños a recibir por cada vida
2. La nave tiene 3 vidas de la nave
3. Cuando una vida se acaba se reproduce un sonido y animación de explosion.
4. Al perder las 3 vidas se acaba el juego.

```
private void TakeDamage(float amount)
{
    _currentHealth -= amount;
    if (_currentHealth <= 0)
    {
        _totalLives--;
        AddOrRemoveHeartIcon(false);
        ReproduceSound(audioDestroy); //Reproducir sonido
        //Animar explosion al perder una vida
        GameObject explosionPlayer =
            Instantiate(explosionPlayerPrefab, gameObject.transform.position, Quaternion.identity);
        Destroy(explosionPlayer, t:0.18f); //destruir prefab de explosion
        if (_totalLives <= 0)
        {
            Time.timeScale = 0f; // detener el juego
            gameOverContainer.SetActive(true);
        }
        else
        {
            _currentHealth = _healthPerLife; //Reinicia el % de vida
        }
    }
}
```

Gestión de enemigos:

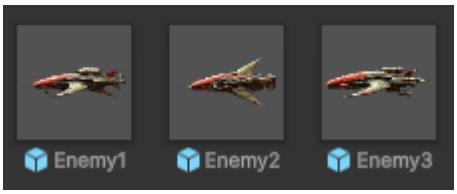
Sistema de Niveles:

se creó el siguiente Diccionario con los distintos punto y daño que puede generar cada uno de los enemigos en el juego dependiendo de su nivel:

DamageConstants.cs

```
private static readonly Dictionary<(string tag, int nivel), EnemyStats> StatsByTypeAndLevel = new()
{
    { ("Enemy", 1), new EnemyStats( damage: 10f,  points: 100) },
    { ("Enemy", 2), new EnemyStats( damage: 20f,  points: 150) },
    { ("Enemy", 3), new EnemyStats( damage: 30f,  points: 200) },
    { ("Boss", 1), new EnemyStats( damage: 30f,  points: 1000) },
    { ("Boss", 2), new EnemyStats( damage: 40f,  points: 1500) },
    { ("Boss", 3), new EnemyStats( damage: 50f,  points: 2000) },
    { ("BulletEnemy", 1), new EnemyStats( damage: 5f,  points: 0) },
    { ("BulletEnemy", 2), new EnemyStats( damage: 15f,  points: 0) },
    { ("BulletEnemy", 3), new EnemyStats( damage: 25f,  points: 0) },
    { ("BulletBoss", 1), new EnemyStats( damage: 25f,  points: 0) },
    { ("BulletBoss", 2), new EnemyStats( damage: 35f,  points: 0) },
    { ("BulletBoss", 3), new EnemyStats( damage: 45f,  points: 0) },
};
```

se crearon distintos Prefabs para cada enemigo distinto:



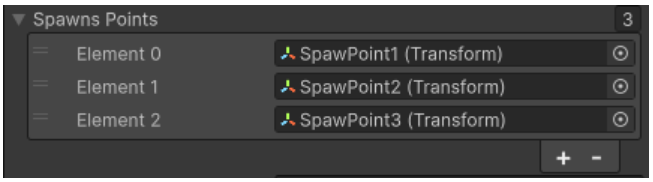
y dependiendo del nivel actual de la partida se genera cada uno de ellos en pantalla:

Spawner.cs

```
switch (i)
{
    case 0:
        Instantiate(enemyPrefab1, randomPosition, Quaternion.identity);
        break;
    case 1:
        Instantiate(enemyPrefab2, randomPosition, Quaternion.identity);
        break;
    case 2:
        Instantiate(enemyPrefab3, randomPosition, Quaternion.identity);
        break;
}
```

Distintos enemigos y disparos

Lo enemigos cuentan con un arreglo de posiciones donde pueden generar los proyectiles:



Enemy.cs

```
[SerializeField] private Transform[] spawnsPoints;
```

de tal forma que al momento de realizar los disparos el enemigo los genera de forma aleatoria en cada uno de esos puntos.

```
Frequently called  1 usage  AlexRose97
IEnumerator SpawnBullet()
{
    while (true)
    {
        Transform[] puntosMezclados = spawnsPoints.OrderBy(x => Random.value).ToArray();
        foreach (Transform punto in puntosMezclados)
        {
            Instantiate(bulletPrefab, punto.position, Quaternion.identity);
            yield return new WaitForSeconds(Random.Range(0f, 0.15f)); //Retrazo entre disparo
        }

        ReproduceSound(audioBullet);
        float tiempoEspera = Random.Range(minTimeBullet, maxTimeBullet);
        yield return new WaitForSeconds(tiempoEspera);
    }
}
```

Enemy.cs

Sistema de dificultad:

La dificultad del juego afecta la cantidad de enemigos que se generan y las oleadas por nivel, para realizar esto se añadió un menú de configuración previo al juego.

Esto nos permite llenar los variables globales que pueden ser utilizadas en distintas partes del juego:

```
public class GameGlobalValues : MonoBehaviour
{
    2 usages
    public static string NombreJugador { get; set; }
    Frequently called  3 usages
    public static string Dificultad { get; set; }


    Frequently called  1 usage  AlexRose97
    public static GameDifficultyData ObtenerDatosPorDificultad(string dificultad){...}
}
```

GameGlobalValues.cs

Esta dificultad afecta directamente las oleadas, generación de enemigos y objetos.

```
public static GameDifficultyData ObtenerDatosPorDificultad(string dificultad)
{
    var data = new GameDifficultyData();

    switch (dificultad.ToUpperInvariant())
    {
        case "FACIL":
            data.enemigos = 5;
            data.oleadas = 1;
            data.itemsDrop = 0.25f;
            data.tiempoEnemigo = 1f;
            break;
        case "MODERADO":
            data.enemigos = 10;
            data.oleadas = 3;
            data.itemsDrop = 0.15f;
            data.tiempoEnemigo = 0.75f;
            break;
        case "DIFICIL":
            data.enemigos = 20;
            data.oleadas = 4;
            data.itemsDrop = 0.05f;
            data.tiempoEnemigo = 0.5f;
            break;
    }

    return data;
}
```

GameGlobalValues.cs

la cual es utilizada en la lógica de generación de enemigos.

```
IEnumerator SpawnEnemies()
{
    var datos :GameDifficultyData = GameGlobalValues.ObtenerDatosPorDificultad(GameGlobalValues.Dificultad);
    //Niveles
    for (int i = 0; i < 3; i++)
    {
        //Oleadas
        for (int j = 0; j < datos.oleadas; j++)
        {
            imageBorder.enabled = true;
            textOleada.text = $"Nivel {i + 1} - Oleada {j + 1}";
            yield return new WaitForSeconds(1f); //Tiempo para borrar el texto
            imageBorder.enabled = false;
            textOleada.text = "";
            //Enemigos
            for (int k = 0; k < datos.enemigos; k++){...}

            yield return new WaitForSeconds(3f); //Tiempo entre cada Oleada
        }

        yield return new WaitForSeconds(5f); //Tiempo entre cada Nivel
    }
```

Spawner.cs

Guardado de puntuaciones:

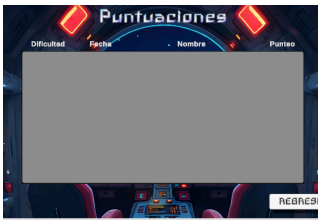
Se crea una función llamada CargarDatos, que permite realizar la lectura de un archivo físico que guardara la información.

```
public void CargarDatos()
{
    if (File.Exists(RutaArchivo))
    {
        string json = File.ReadAllText(RutaArchivo);
        lista = JsonUtility.FromJson<ListaDePartidas>(json);
    }
}
```

Al finalizar la partida el juego muestra un modal, el cual al darle click en ok ejecuta la transición a la pantalla inicial que también ejecuta la función GuardarPartida, que permite adjuntar a la lista de partidas la información de la partida actual.

```
public void GuardarPartida()
{
    Player player = FindFirstObjectByType<Player>();
    lista.partidas.Add(new PartidaData
    {
        nombreJugador = GameGlobalValues.NombreJugador,
        dificultad = GameGlobalValues.Dificultad,
        puntuacion = player.Score,
        fecha = DateTime.Now.ToString( format: "yyyy-MM-dd HH:mm:ss"),
    });
    string json = JsonUtility.ToJson(lista,  prettyPrint: true);
    File.WriteAllText(RutaArchivo,  contents: json);
}
```

De igual forma en la pantalla de punteos se realiza la lectura del archivo que posee el listado de partidas guardadas, por lo que acá lo que se realiza es un recorrido para mostrarlas con ayuda de un prefab en orden descendente.



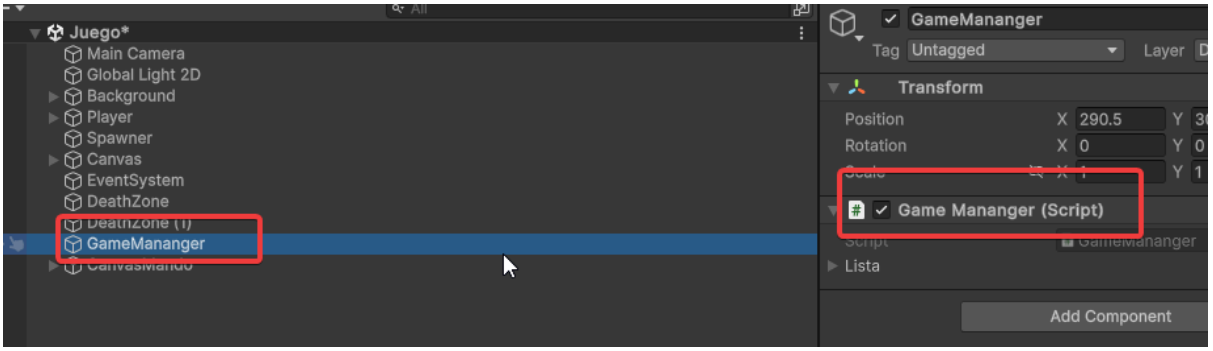

```
public void MostrarRanking()
{
    // Limpia el contenido anterior
    foreach (Transform hijo in contenedorItems)
    {
        Destroy(hijo.gameObject);
    }

    // Ordenar por puntuación descendente
    var partidasOrdenadas :List<PartidaData>  = lista.partidas // List<PartidaData>
        .Where(p :PartidaData  => p.puntuacion > 0) // IEnumerable<PartidaData>
        .OrderByDescending(p :PartidaData  => p.puntuacion) // IOrderedEnumerable<PartidaData>
        .ToList();

    foreach (var partida in partidasOrdenadas)
    {
        GameObject item = Instantiate(itemUIPrefab, contenedorItems);
        var ui = item.GetComponent<ItemUIPunteos>();
        ui.Configurar(partida.nombreJugador, partida.puntuacion, partida.fecha, partida.dificultad);
    }
}
```

MenuMananger.cs

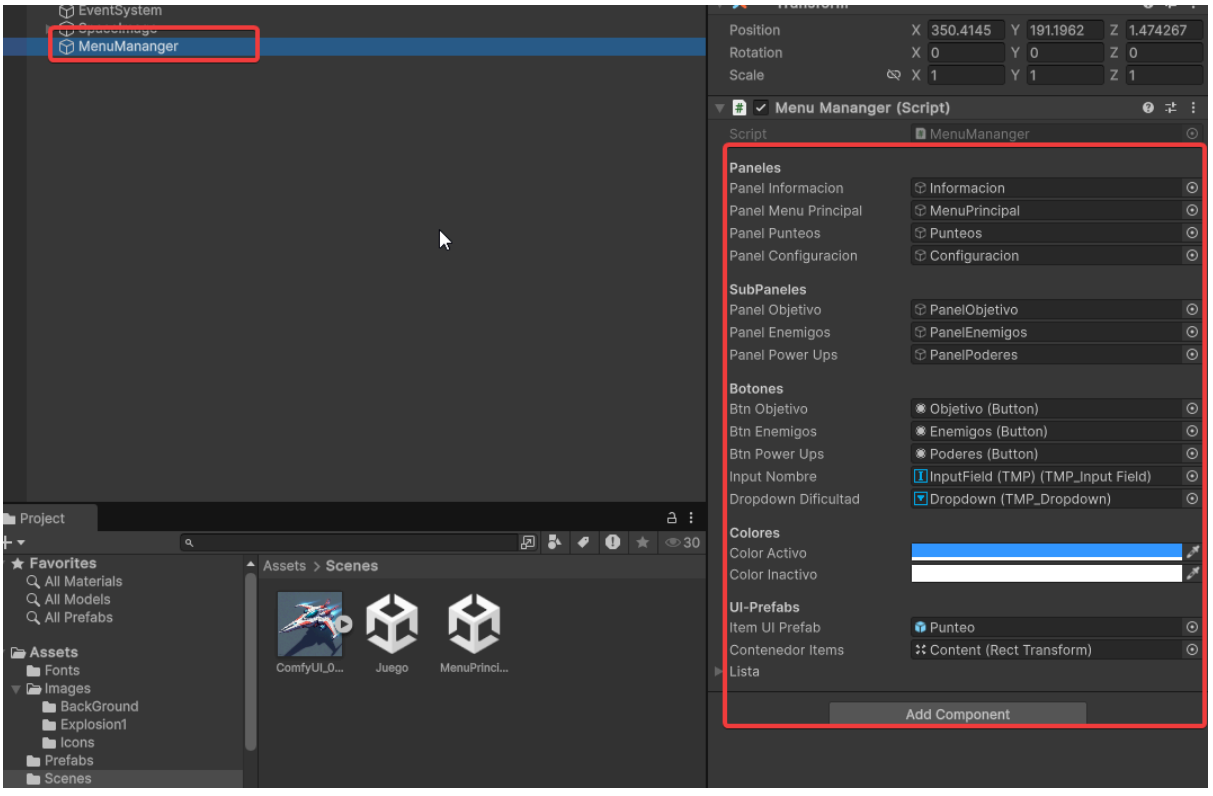
UI:

La interfaz cuenta con distintos paneles que muestran informacion al usuario.

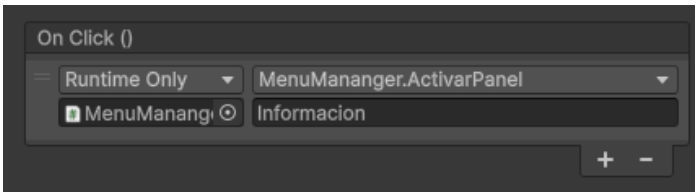
Para poder realizar las transiciones y mostrar los distintos modales se utilizan los siguientes componentes y scripts.



MenuMananger.cs



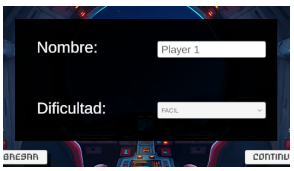
y en los botones se agrega el llamado a la función ActivarPanel, que nos permite mostrar el panel correspondiente.




```
/// <summary>
/// Funcion para activar los paneles principales de la ventana informacion
/// </summary>
/// <param name="panel">Nombre del panel ej: Informacion</param>
5+ asset usages  AlexRose97
public void ActivarPanel(string panel)
{
    // Activar el panel correspondiente
    if (panel == "Informacion"){...}
    else if (panel == "MenuPrincipal"){...}
    else if (panel == "Punteos"){...}
    else if (panel == "Configuracion"){...}
}
```

MenuMananger.cs

En el caso de la pantalla utilizada para guardar el nombre y seleccionar la dificultad, se utiliza una función llamada ButtonPlay, que permite guardar en variables globales estos valores que son utilizados en el resto del juego.




```
public void ButtonPlay()
{
    GameGlobalValues.NombreJugador = inputNombre.text;
    GameGlobalValues.Dificultad = dropdownDificultad.options[dropdownDificultad.value].text;
    SceneManager.LoadScene("Juego");
}
```

MenuMananger.cs

5. Conclusiones

1. El desarrollo del proyecto permitió reforzar conocimientos clave de Unity y programación en C#.
2. Uno de los mayores aprendizajes fue la necesidad de estructurar bien el código para adaptarlo a distintas plataformas (PC y móvil).
3. Se comprendió cómo detectar colisiones entre objetos utilizando el método OnTriggerEnter2D, esencial para manejar interacciones como daño, recolección de ítems o destrucción de enemigos.
4. Se aprendió a crear y destruir objetos dinámicamente en base a prefabs, permitiendo la aparición controlada de enemigos, ítems y efectos visuales durante el juego. Esto se logró mediante el uso de Instantiate() para generar nuevas instancias y Destroy() para eliminarlas.
5. Se abordó el reto de mantener datos entre escenas (como el nombre del jugador y la dificultad seleccionada). Para resolverlo de forma eficiente se utilizó una clase estática global, que permitió almacenar y recuperar información de manera accesible desde cualquier escena del juego.

Principales retos:

1. Gestionar los controles táctiles para una experiencia fluida en el móvil.
2. Implementar un sistema de dificultad que afecta varios aspectos del gameplay.
3. Organizar correctamente la UI para mantener claridad en pantallas pequeñas.