

Manual de Ingeniería CyberWar

I. Gestión de Escenas y Navegación

Scripts que controlan el flujo del juego (menú, selección de nivel y pausa).

1. menuproyecto1.cs

```
public class menuproyecto1 : MonoBehaviour
{
    public void StartGame()
    {
        // Carga la escena llamada "SampleScene"
        SceneManager.LoadScene("ElegirMundo");
    }

    public void StartGame1()
    {
        // Carga la escena llamada "SampleScene"
        SceneManager.LoadScene("Historia");
    }

    public void ExitGame()
    {
        Debug.Log("Saliendo del juego...");
        Application.Quit();
    }

    public void OpenOptions()
    {
        Debug.Log("Abriendo menú de opciones...");
    }
}
```

- **Propósito:** Controla la carga inicial de escenas desde el menú principal.
- **Funcionalidad Clave:**
 - StartGame(): Carga la escena de selección de mundo ("ElegirMundo").
 - StartGame1(): Carga la escena de introducción o historia ("Historia").
 - ExitGame(): Cierra la aplicación (Application.Quit()).

2. ElegirMundo.cs

```
public class ElegirMundo : MonoBehaviour
{
    public void JugarMundo()
    {
        SceneManager.LoadScene("Mundo1");
    }

    public void VolverAlMenu()
    {
        SceneManager.LoadScene("menuInicial");
    }

    public void Mundo2()
    {
        SceneManager.LoadScene("Mundo2");
    }
}
```

- **Propósito:** Gestiona la selección y carga de niveles específicos.
- **Funcionalidad Clave:**
 - JugarMundo() / Mundo2(): Carga los niveles de juego ("Mundo1", "Mundo2").
 - VolverAlMenu(): Regresa a la escena inicial ("menuInicial").

3. PausaJuego.cs

```
public class PausaJuego : MonoBehaviour
{
    public static bool GamePaused = false;

    public GameObject panelPausa;

    public void Pausar()
    {
        GamePaused = !GamePaused;

        if (GamePaused)
        {
            panelPausa.SetActive(true);
            Time.timeScale = 0f;
        }
        else
        {
            panelPausa.SetActive(false);
            Time.timeScale = 1f;
        }
    }

    public void VolverAlMenu()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene("menuInicial");
    }
}
```

- **Propósito:** Implementa la funcionalidad de pausa.
- **Funcionalidad Clave:**
 - Utiliza una variable estática (GamePaused) para rastrear el estado.
 - Pausar(): Alterna la visibilidad del panelPausa y establece **Time.timeScale = 0f** (pausa) o **1f** (reanudar).

4. GameOverManager.cs (Singleton)

```

public class GameOverManager : MonoBehaviour
{
    public static GameOverManager Instance { get; private set; }

    public GameObject gameOverPanel;

    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(gameObject);

        gameOverPanel.SetActive(false);
    }

    public void ShowGameOver()
    {
        Time.timeScale = 0f; // Congelar el juego
        gameOverPanel.SetActive(true);
    }

    public void Retry()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }

    public void GoToMenu()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene("MenuInicial");
    }
}

```

- **Propósito:** Controla el estado y la interfaz de la pantalla de "Game Over".
- **Funcionalidad Clave:**
 - Implementa el patrón **Singleton**.
 - ShowGameOver(): Activa el gameOverPanel y congela el juego (Time.timeScale = 0f).
 - Ofrece Retry() (recargar escena actual) y GoToMenu() (volver al menú).

5. VictoryManager.cs (Singleton)

```
public class VictoryManager : MonoBehaviour
{
    public static VictoryManager Instance { get; private set; }

    public GameObject victoryPanel;

    private void Awake()
    {
        if (Instance == null) Instance = this;
        else Destroy(gameObject);
    }

    private void Start()
    {
        if (victoryPanel != null)
            victoryPanel.SetActive(false);
    }

    public void ShowVictory()
    {
        if (victoryPanel != null)
        {
            victoryPanel.SetActive(true);
            Time.timeScale = 0f; // Pausa el juego
        }
    }

    // Botón de volver al menú
    public void ReturnToMenu()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene("MenuInicial");
    }
}
```

- **Propósito:** Controla la interfaz final de victoria.
- **Funcionalidad Clave:**
 - Implementa el patrón **Singleton**.
 - ShowVictory(): Activa el victoryPanel y congela el juego (Time.timeScale = 0f).

- ReturnToMenu(): Reanuda el tiempo y regresa a la escena inicial.
-

II. Mecánicas de Combate

Scripts que gestionan la puntería, el disparo y la interacción del proyectil.

1. GunAim.cs

```
public class GunAim : MonoBehaviour
{
    void Update()
    {
        Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector2 direction = mousePos - transform.position;

        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        transform.rotation = Quaternion.Euler(0, 0, angle);
    }
}
```

- **Propósito:** Rota un objeto (el arma) para seguir la posición del ratón.
- **Funcionalidad Clave:**
 - Calcula el ángulo (Mathf.Atan2) entre la posición del arma y la posición del cursor en el mundo, aplicando la rotación en el eje Z.

2. Bullet.cs

```

public class Bullet : MonoBehaviour
{
    public float speed = 12f;
    public float lifetime = 2f;

    void Start()
    {
        Destroy(gameObject, lifetime);
    }

    void Update()
    {
        transform.Translate(Vector2.right * speed * Time.deltaTime);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        Debug.Log("Bala golpeó: " + collision.name);

        if (collision.CompareTag("Enemy"))
        {
            Destroy(collision.gameObject); // Destruye enemigo

            // Notificar al RoundManager
            if (RoundManager.Instance != null)
            {
                RoundManager.Instance.OnEnemyKilled();
            }
        }

        Destroy(gameObject); // Destruye bala siempre
    }
}

```

- **Propósito:** Define el comportamiento de los proyectiles.
- **Funcionalidad Clave:**
 - **Movimiento:** Se mueve continuamente usando transform.Translate.
 - **Colisión:** Al colisionar con un objeto con el tag "Enemy", destruye al enemigo y llama a RoundManager.Instance.OnEnemyKilled().
 - **Autodestrucción:** Se destruye después de un lifetime definido o tras cualquier colisión.

3. PlayerShooting.cs (Alternativa simple)

```
public class PlayerShooting : MonoBehaviour
{
    [Header("Shooting Settings")]
    public GameObject bulletPrefab;           // Prefab de la bala
    public Transform firePoint;                // Lugar donde sale la bala
    public float bulletSpeed = 10f;            // Velocidad de la bala
    public float fireRate = 0.25f;             // Tiempo entre cada disparo

    private float nextFireTime = 0f;

    void Update()
    {
        // Disparar con clic izquierdo o con la tecla Espacio
        if ((Input.GetMouseButton(0) || Input.GetKey(KeyCode.Space)) && Time.time >= nextFireTime)
        {
            Shoot();
            nextFireTime = Time.time + fireRate;
        }
    }

    void Shoot()
    {
        // Crear bala
        GameObject bullet = Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);

        // Agregar velocidad
        Rigidbody2D rb = bullet.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            rb.linearVelocity = firePoint.right * bulletSpeed;
        }

        // Destruir bala a los 3 segundos
        Destroy(bullet, 3f);
    }
}
```

- **Propósito:** Implementa un sistema de disparo básico con tasa de fuego.
- **Funcionalidad Clave:**
 - Controla el tiempo entre disparos (fireRate) usando Time.time.
 - Instancia el bulletPrefab y le aplica una velocidad fija.

4. PlayerAimAndShoot.cs

```
public class PlayerAimAndShoot : MonoBehaviour
{
    [Header("Shooting")]
    public GameObject bulletPrefab;
    public Transform firePoint;
    public float bulletSpeed = 10f;

    private Vector3 originalScale;
    private Collider2D playerCollider;
    private Animator anim;

    private PlayerMovement movement; // <-- IMPORTANTE

    void Start()
    {
        originalScale = transform.localScale;

        playerCollider = GetComponent<Collider2D>();
        anim = GetComponent<Animator>();
        movement = GetComponent<PlayerMovement>(); // <-- CONEXIÓN
    }

    void Update()
    {
        FlipTowardsMouse();

        // Disparo con clic derecho
        if (Input.GetMouseButtonUp(1))
        {
            Shoot();
        }
    }
}
```

```

void FlipTowardsMouse()
{
    Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
    mousePos.z = 0f;

    Vector3 direction = mousePos - transform.position;

    if (direction.x >= 0.01f)
        transform.localScale = new Vector3(originalScale.x, originalScale.y, originalScale.z);
    else if (direction.x <= -0.01f)
        transform.localScale = new Vector3(-originalScale.x, originalScale.y, originalScale.z);

    float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
    firePoint.rotation = Quaternion.Euler(0f, 0f, angle);
}

void Shoot()
{
    // Activar animaci n
    anim.SetBool("Shoot", true);
    movement.isShooting = true;

    StartCoroutine(StopShootAnimation());
}

// Instanciar bala
GameObject bullet = Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);

// Ignorar colisi n con jugador
Collider2D bulletCol = bullet.GetComponent<Collider2D>();
if (bulletCol != null && playerCollider != null)
    Physics2D.IgnoreCollision(bulletCol, playerCollider);

// Aplicar velocidad
Rigidbody2D rb = bullet.GetComponent<Rigidbody2D>();
rb.linearVelocity = firePoint.right * bulletSpeed;
}

private IEnumerator StopShootAnimation()
{
    yield return new WaitForSeconds(0.15f); // Ajusta al largo de tu animaci n
    anim.SetBool("Shoot", false);
    movement.isShooting = false;
}
}

```

- **Prop sito:** Integra la punter a, el disparo y la sincronizaci n de animaci n.
- **Funcionalidad Clave:**
 - `FlipTowardsMouse()`: Voltea el *sprite* del personaje para que mire al cursor.
 - `Shoot()`: Instancia el proyectil, utiliza **Physics2D.IgnoreCollision** para evitar colisi n con el propio jugador, y activa el estado `isShooting` en `PlayerMovement.cs` para sincronizar animaciones.

- Utiliza una corutina para desactivar el estado de disparo después de un breve tiempo.
-

III. Gestión de Enemigos y Dificultad

Scripts que definen el comportamiento de las unidades enemigas y el sistema de aparición.

1. FlyingEnemy.cs

```
public class FlyingEnemy : MonoBehaviour
{
    [Header("Follow Settings")]
    public Transform player;
    public float speed = 2f;
    public float followRange = 15f;

    private Animator animator;
    private Vector3 originalScale;

    void Start()
    {
        originalScale = transform.localScale;

        Rigidbody2D rb = GetComponent<Rigidbody2D>();
        if (rb == null)
            rb = gameObject.AddComponent<Rigidbody2D>();

        rb.gravityScale = 0;
        rb.freezeRotation = true;
        rb.bodyType = RigidbodyType2D.Kinematic;

        // IMPORTANTE: evita que el collider frene al enemigo
        rb.collisionDetectionMode = CollisionDetectionMode2D.Discrete;
    }
}
```

```

void Update()
{
    if (player == null) return;

    float distance = Vector2.Distance(transform.position, player.position);

    if (distance < followRange)
    {
        // Movimiento preciso hasta tocar al jugador
        transform.position = Vector2.MoveTowards(
            transform.position,
            player.position,
            speed * Time.deltaTime
        );

        animator?.SetBool("isFlying", true);

        // Voltear sprite
        if (player.position.x > transform.position.x)
            transform.localScale = originalScale;
        else
            transform.localScale = new Vector3(-originalScale.x, originalScale.y, originalScale.z);
    }
    else
    {
        animator?.SetBool("isFlying", false);
    }
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        PlayerHealth health = collision.GetComponent<PlayerHealth>();

        if (health != null)
            health.TakeDamage();

        // El enemigo desaparece al tocar
        Destroy(gameObject);
    }
}

```

- **Propósito:** Define la lógica de un enemigo volador rastreador.

- **Funcionalidad Clave:**

- Usa un Rigidbody2D **Kinematic** con gravityScale = 0.
- **Seguimiento:** Utiliza Vector2.MoveTowards para moverse hacia el jugador (player) si está dentro del followRange.
- **Ataque:** Al colisionar con el "Player", llama a PlayerHealth.TakeDamage() y se autodestruye.

2. EnemySpawnManager.cs (Singleton)

```
public class EnemySpawnManager : MonoBehaviour
{
    public static EnemySpawnManager Instance { get; private set; }

    public GameObject enemyPrefab;           // Prefab del enemigo
    public List<Transform> spawnPoints;     // Lista de spawners (tipo abstracto)
    public float spawnInterval = 3f;

    private float currentSpawnInterval;
    private bool isSpawning = true;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }

    void Start()
    {
        currentSpawnInterval = spawnInterval;
        InvokeRepeating(nameof(SpawnEnemy), 1f, currentSpawnInterval);
    }
}
```

```

void SpawnEnemy()
{
    if (!isSpawning)
        return;

    if (spawnPoints.Count == 0 || enemyPrefab == null)
    {
        Debug.LogWarning("Faltan spawners o prefab del enemigo.");
        return;
    }

    // Spawner aleatorio
    int randomIndex = Random.Range(0, spawnPoints.Count);
    Transform chosenSpawner = spawnPoints[randomIndex];

    GameObject enemy = Instantiate(enemyPrefab, chosenSpawner.position, Quaternion.identity);

    FlyingEnemy flyingEnemy = enemy.GetComponent<FlyingEnemy>();
    if (flyingEnemy != null && flyingEnemy.player == null)
    {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player != null)
        {
            flyingEnemy.player = player.transform;
        }
    }
}

public void PauseSpawning()
{
    isSpawning = false;
    Debug.Log("Enemy spawning paused");
}

public void ResumeSpawning()
{
    isSpawning = true;
    Debug.Log("Enemy spawning resumed");
}

public void IncreaseDifficulty(float multiplier)
{
    currentSpawnInterval = Mathf.Max(0.5f, currentSpawnInterval / multiplier);

    CancelInvoke(nameof(SpawnEnemy));
    InvokeRepeating(nameof(SpawnEnemy), 0.5f, currentSpawnInterval);

    Debug.Log($"Difficulty increased! New spawn interval: {currentSpawnInterval}s");
}
}

```

- **Propósito:** Administra la aparición de enemigos.
- **Funcionalidad Clave:**

- Implementa el patrón **Singleton**.
 - Utiliza InvokeRepeating para llamar repetidamente a SpawnEnemy() en puntos aleatorios (spawnPoints).
 - IncreaseDifficulty(): Reduce el currentSpawnInterval, acelerando la aparición de enemigos para incrementar la dificultad.
-

IV. Mecánicas Centrales del Jugador

Scripts que gestionan el movimiento, la salud y el *feedback* al jugador.

1. PlayerMovement.cs

```
public class PlayerMovement : MonoBehaviour
{
    [Header("Movimiento")]
    public float walkSpeed = 3f;
    public float runSpeed = 6f;
    public float jumpForce = 6f;

    [Header("Componentes")]
    private Rigidbody2D rb;
    private Animator anim;
    private SpriteRenderer sr;

    private bool isGrounded = true;
    private Vector2 moveInput;

    // --- Disparo ---
    [HideInInspector] public bool isShooting = false;
    [HideInInspector] public float shootTimer = 0f;
    private float shootCooldown = 0.25f;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        sr = GetComponent<SpriteRenderer>();
    }
}
```

```

void Update()
{
    // --- CONTROLAR TIEMPO DE DISPARO ---
    if (isShooting)
    {
        shootTimer -= Time.deltaTime;
        if (shootTimer <= 0)
        {
            isShooting = false;
            anim.SetBool("Shoot", false);
        }
    }

    // --- Movimiento horizontal ---
    moveInput.x = Input.GetAxisRaw("Horizontal");
    bool isMoving = Mathf.Abs(moveInput.x) > 0.1f;

    // --- Correr con Shift ---
    bool isPressingShift = Input.GetKey(KeyCode.LeftShift);
    float currentSpeed = isPressingShift ? runSpeed : walkSpeed;

    rb.linearVelocity = new Vector2(moveInput.x * currentSpeed, rb.linearVelocity.y);

    // --- Saltar ---
    if (Input.GetKeyDown(KeyCode.W) && isGrounded)
    {
        rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
        isGrounded = false;
        anim.SetTrigger("Jumping");
    }

    // --- Animaci n de caminar/correr (solo si NO est  disparando) ---
    if (!isShooting)
    {
        anim.SetBool("isRunning", isMoving);
    }

    // --- Rotar sprite seg n direcci n ---
    if (moveInput.x != 0)
        sr.flipX = moveInput.x < 0;
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Ground"))
        isGrounded = true;
}

```

- **Prop sito:** Gestiona el movimiento f sico y las animaciones base.
- **Funcionalidad Clave:**

- Controla la velocidad de caminata/carrera (walkSpeed, runSpeed) con la tecla Shift.
- Controla el salto (jumpForce) y el chequeo de colisión con el suelo (isGrounded).
- Sincroniza la animación de isShooting y el *cooldown* de disparo con PlayerAimAndShoot.cs.

2. PlayerHealth.cs

```
public class PlayerHealth : MonoBehaviour
{
    [Header("Lives")]
    public int maxLives = 3;
    private int currentLives;

    [Header("UI Hearts")]
    public Image[] heartImages;           // Asigna los corazones del Canvas
    public Sprite fullHeart;
    public Sprite emptyHeart;

    private HeartNode head;              // Lista enlazada de corazones

    [Header("Invincibility")]
    public float invincibleTime = 1f;
    private bool isInvincible = false;

    private void Start()
    {
        currentLives = maxLives;
        BuildLinkedList();
        UpdateHearts();
    }

    // Construye la lista enlazada basada en el array del inspector
    private void BuildLinkedList()
    {
        HeartNode prev = null;

        for (int i = 0; i < heartImages.Length; i++)
        {
            HeartNode newNode = new HeartNode(heartImages[i]);

            if (head == null)
                head = newNode;
            else
                prev.next = newNode;

            prev = newNode;
        }
    }
}
```

```
public void TakeDamage()
{
    if (isInvincible)
        return;

    currentLives--;

    if (currentLives <= 0)
    {
        currentLives = 0;
        UpdateHearts();
        Die();
        return;
    }

    UpdateHearts();
    StartCoroutine(Invincibility());
}

private IEnumerator Invincibility()
{
    isInvincible = true;
    yield return new WaitForSeconds(invincibleTime);
    isInvincible = false;
}

private void Die()
{
    Debug.Log("El jugador murió");

    if (GameOverManager.Instance != null)
        GameOverManager.Instance.ShowGameOver();
    else
        Debug.LogError("NO HAY GameOverManager en la escena.");
}
```

```

private void UpdateHearts()
{
    HeartNode current = head;
    int index = 0;

    while (current != null)
    {
        if (index < currentLives)
            current.heartImage.sprite = fullHeart;
        else
            current.heartImage.sprite = emptyHeart;

        current = current.next;
        index++;
    }
}

// Nodo de lista enlazada para manejar corazones
public class HeartNode
{
    public Image heartImage;
    public HeartNode next;

    public HeartNode(Image img)
    {
        heartImage = img;
        next = null;
    }
}

```

- **Propósito:** Administra el sistema de vida, daño e invencibilidad del jugador.
- **Estructura de Datos:** Utiliza una **lista enlazada** de nodos (HeartNode) para gestionar la interfaz de corazones (Image[] heartImages).
- **Funcionalidad Clave:**
 - TakeDamage(): Decrementa currentLives y, si el jugador sigue vivo, inicia la **corrutina Invincibility()**.
 - Invincibility(): Establece el estado isInvincible durante un tiempo definido para evitar daño repetido.
 - Die(): Llama a GameOverManager.Instance.ShowGameOver().

V. Flujo de Juego y Desafío

Scripts que orquestan el progreso por rondas y gestionan el desafío final.

1. RoundManager.cs (Singleton)

```
public class RoundManager : MonoBehaviour
{
    public static RoundManager Instance;

    [Header("Round Settings")]
    public int enemiesPerRound = 10;           // Valor base usado si no hay cola
    public int totalRounds = 3;
    public float difficultyIncreaseRate = 1.2f;

    // 🔳 NUEVA ESTRUCTURA DE DATOS
    private Queue<int> enemiesQueue = new Queue<int>();

    [Header("Current Round Info")]
    public int currentRound = 1;
    private int enemiesKilledThisRound = 0;
    private int requiredEnemiesThisRound; // valor extraido de la cola

    [Header("UI")]
    public TextMeshProUGUI roundText;
    public TextMeshProUGUI killsText;

    [Header("Events")]
    public bool pauseSpawningBetweenRounds = true;
    public float delayBetweenRounds = 3f;

    [Header("Panels")]
    public GameObject victoryPanel;

    private bool isRoundActive = true;

    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
        {
            Destroy(gameObject);
            return;
        }

        if (victoryPanel != null)
            victoryPanel.SetActive(false);
    }
}
```

```
private void Start()
{
    BuildRoundQueue();

    requiredEnemiesThisRound = enemiesQueue.Dequeue(); // SE USA LA COLA
    UpdateUI();
}

private void BuildRoundQueue()
{
    for (int i = 1; i <= totalRounds; i++)
    {
        int enemies = Mathf.RoundToInt(enemiesPerRound * Mathf.Pow(difficultyIncreaseRate, i - 1));
        enemiesQueue.Enqueue(enemies);
    }
}

public void OnEnemyKilled()
{
    enemiesKilledThisRound++;

    UpdateUI();

    if (enemiesKilledThisRound >= requiredEnemiesThisRound)
    {
        CompleteRound();
    }
}
```

```
private void CompleteRound()
{
    Debug.Log($"Round {currentRound} completada.");

    if (currentRound >= totalRounds)
    {
        Debug.Log("Juego completado. Mostrando pregunta final...");
        QuestionManager.Instance.ShowRandomQuestion();
        return;
    }

    if (pauseSpawningBetweenRounds)
    {
        isRoundActive = false;
        if (EnemySpawnManager.Instance != null)
            EnemySpawnManager.Instance.PauseSpawning();
    }

    Invoke(nameof(StartNextRound), delayBetweenRounds);
}

private void StartNextRound()
{
    currentRound++;
    enemiesKilledThisRound = 0;
    isRoundActive = true;

    requiredEnemiesThisRound = enemiesQueue.Dequeue(); // SIGUIENTE VALOR DE LA COLA

    Debug.Log($"Iniciando ronda {currentRound}...");

    if (EnemySpawnManager.Instance != null)
    {
        EnemySpawnManager.Instance.IncreaseDifficulty(difficultyIncreaseRate);
        EnemySpawnManager.Instance.ResumeSpawning();
    }

    UpdateUI();
}
```

```

private void UpdateUI()
{
    if (roundText != null)
        roundText.text = $"Round {currentRound}/{totalRounds}";

    if (killsText != null)
        killsText.text = $"Kills: {enemiesKilledThisRound}/{requiredEnemiesThisRound}";
}

public bool IsRoundActive()
{
    return isRoundActive;
}

public void ShowFinalVictory()
{
    Debug.Log("Jugador ha ganado con éxito. Mostrando panel final...");

    if (victoryPanel != null)
        victoryPanel.SetActive(true);

    Time.timeScale = 0f;
}

```

- **Propósito:** Controla la progresión del juego a través de rondas.
- **Estructura de Datos:** Usa una **cola genérica (Queue<int>)** para almacenar el número de enemigos requeridos en cada ronda.
- **Funcionalidad Clave:**
 - OnEnemyKilled(): Se invoca al matar un enemigo; si se alcanza el objetivo, llama a EndRound().
 - StartNextRound(): Incrementa currentRound, extrae el nuevo objetivo de la cola y llama a EnemySpawnManager.Instance.IncreaseDifficulty().
 - Si se completa la última ronda, invoca a QuestionManager.Instance.ShowRandomQuestion().

2. QuestionManager.cs (Singleton)

```
public class QuestionManager : MonoBehaviour
{
    public static QuestionManager Instance;

    [Header("UI")]
    public GameObject questionPanel;
    public TextMeshProUGUI questionText;

    private Question currentQuestion;

    [System.Serializable]
    public class Question
    {
        public string text;
        public bool answer; // true = verdadero, false = falso
    }

    [Header("Preguntas de Ciberseguridad")]
    public Question[] questions = new Question[5];

    void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(gameObject);

        questionPanel.SetActive(false);
    }

    public void ShowRandomQuestion()
    {
        int index = Random.Range(0, questions.Length);
        currentQuestion = questions[index];

        questionText.text = currentQuestion.text;
        questionPanel.SetActive(true);

        Time.timeScale = 0f; // Pausar el juego mientras responde
    }
}
```

```

public void AnswerTrue()
{
    CheckAnswer(true);
}

public void AnswerFalse()
{
    CheckAnswer(false);
}

void CheckAnswer(bool playerAnswer)
{
    questionPanel.SetActive(false);
    Time.timeScale = 1f;

    if (playerAnswer == currentQuestion.answer)
    {
        Debug.Log("✓ Respuesta correcta!");
        RoundManager.Instance.ShowFinalVictory(); // 🎉 MOSTRAR PANEL DE VICTORIA
    }
    else
    {
        Debug.Log("✗ Respuesta incorrecta. Reiniciando...");
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}

```

- **Propósito:** Presenta un desafío final de verdadero/falso.
- **Funcionalidad Clave:**
 - ShowRandomQuestion(): Selecciona una pregunta, activa el questionPanel y **pausa el juego** (Time.timeScale = 0f).
 - CheckAnswer(): Si la respuesta es correcta, llama a RoundManager.Instance.ShowFinalVictory() (o VictoryManager.ShowVictory()). Si es incorrecta, reinicia la escena.

VI. Customización y Sistemas Auxiliares

Scripts para la gestión de la interfaz de opciones, el audio, los *skins* y la cuenta regresiva.

1. MusicManager.cs (Singleton, Persistente)

```
public class MusicManager : MonoBehaviour
{
    [SerializeField] private AudioSource musicSource;
    [SerializeField] private Slider volumeSlider;

    private static MusicManager instance;

    private void Awake()
    {
        // Patrón Singleton
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }
    }

    private void Start()
    {
        if (musicSource == null)
            musicSource = GetComponent<AudioSource>();

        // Recupera el volumen guardado
        float savedVolume = PlayerPrefs.GetFloat("MusicVolume", 0.5f);
        musicSource.volume = savedVolume;

        if (volumeSlider != null)
        {
            volumeSlider.value = savedVolume;
            volumeSlider.onValueChanged.AddListener(ChangeVolume);
        }
    }
}
```

```
private void ChangeVolume(float value)
{
    musicSource.volume = value;
    PlayerPrefs.SetFloat("MusicVolume", value); // Guarda el volumen global
}

public void RegisterSlider(Slider newSlider)
{
    if (newSlider == null) return;

    volumeSlider = newSlider;
    volumeSlider.value = musicSource.volume;
    volumeSlider.onValueChanged.RemoveAllListeners();
    volumeSlider.onValueChanged.AddListener(ChangeVolume);
}

}
```

- **Propósito:** Gestiona la reproducción de música de fondo y el control de volumen.
- **Funcionalidad Clave:**
 - Usa el patrón **Singleton** y **DontDestroyOnLoad** para persistir entre escenas.
 - Guarda y carga el volumen con **PlayerPrefs**.
 - RegisterSlider(Slider newSlider): Sincroniza el gestor con el *slider* de la interfaz de usuario en la escena actual.

2. VolumeSliderLinker.cs

```
public class VolumeSliderLinker : MonoBehaviour
{
    private void Start()
    {
        Slider slider = GetComponent<Slider>();

        if (FindObjectOfType<MusicManager>() != null)
        {
            FindObjectOfType<MusicManager>().RegisterSlider(slider);
        }
        else
        {
            Debug.LogWarning("[!] No se encontró el MusicManager en la escena.");
        }
    }
}
```

- **Propósito:** Script auxiliar para enlazar un Slider de la escena con el MusicManager persistente.
- **Funcionalidad Clave:**
 - En Start(), busca la instancia del MusicManager y llama a su método RegisterSlider(), asegurando que el control de volumen de la UI esté funcional desde el inicio.

3. SettingsController.cs

```
public class SettingsController : MonoBehaviour
{
    [Header("Panels")]
    [SerializeField] private GameObject settingsPanel;
    [SerializeField] private GameObject skinsPanel;
    [SerializeField] private GameObject ayudaPanel;

    // estructura de datos
    private Stack<GameObject> panelHistory = new Stack<GameObject>();
    private bool isOpen = false;

    private void Start()
    {

        if (settingsPanel == null)
            settingsPanel = GameObject.Find("SettingsPanel");

        if (skinsPanel == null)
            skinsPanel = GameObject.Find("PanelSkins");

        if (ayudaPanel == null)
            ayudaPanel = GameObject.Find("AyudaPanel");

        if (settingsPanel != null) settingsPanel.SetActive(false);
        if (skinsPanel != null) skinsPanel.SetActive(false);
        if (ayudaPanel != null) ayudaPanel.SetActive(false);
    }

    public void OpenSettings()
    {
        if (settingsPanel == null) return;

        settingsPanel.SetActive(true);
        panelHistory.Clear();

        Time.timeScale = 0f;
        isOpen = true;
    }
}
```

```
public void CloseSettings()
{
    if (settingsPanel == null) return;

    settingsPanel.SetActive(false);
    Time.timeScale = 1f;
    panelHistory.Clear();
    isOpen = false;
}

private void OpenPanel(GameObject panelToOpen)
{
    if (panelToOpen == null) return;

    if (panelHistory.Count == 0)
    {
        panelHistory.Push(settingsPanel);
    }
    else
    {
        panelHistory.Push(GetCurrentPanel());
    }

    GetCurrentPanel()?.SetActive(false);

    panelToOpen.SetActive(true);
}
```

```

public void GoBack()
{
    if (panelHistory.Count == 0)
    {
        CloseSettings();
        return;
    }

    GetCurrentPanel()?.SetActive(false);

    GameObject previous = panelHistory.Pop();
    previous.SetActive(true);
}

private GameObject GetCurrentPanel()
{
    if (settingsPanel.activeSelf) return settingsPanel;
    if (skinsPanel.activeSelf) return skinsPanel;
    if (ayudaPanel.activeSelf) return ayudaPanel;

    return null;
}

public void OpenSkins()
{
    OpenPanel(skinsPanel);
}

public void OpenAyuda()
{
    OpenPanel(ayudaPanel);
}

```

- **Propósito:** Gestiona la navegación jerárquica de los paneles de opciones.
- **Estructura de Datos:** Utiliza un **Stack (Stack<GameObject>)** para guardar el orden de los paneles abiertos.
- **Funcionalidad Clave:**

- OpenSettings(): Activa el panel principal y pausa el juego.
- OpenPanel(): Empuja el panel actual al *stack* antes de abrir el nuevo.
- GoBack(): Saca el panel anterior del *stack* para regresar.

4. SkinNode.cs

```
public class SkinNode
{
    public Sprite skin;
    public SkinNode next;

    public SkinNode(Sprite s)
    {
        skin = s;
        next = null;
    }
}
```

- **Propósito:** Clase de nodo que define la estructura básica de la lista enlazada para las *skins* (contiene el *Sprite* y la referencia al *next* nodo).

5. SkinManager.cs

```
public class SkinManager : MonoBehaviour
{
    public Sprite[] skinSprites;           // Asignas tus 3 skins
    public SpriteRenderer playerSprite;   // SpriteRenderer del personaje

    private SkinNode head;
    private SkinNode current;

    void Start()
    {
        BuildCircularList();

        // Cargamos la skin seleccionada anteriormente
        int index = SkinSelector.savedSkinIndex;

        current = GetSkinNode(index);

        ApplySkin();
    }

    void BuildCircularList()
    {
        SkinNode prev = null;

        for (int i = 0; i < skinSprites.Length; i++)
        {
            SkinNode newNode = new SkinNode(skinSprites[i]);

            if (head == null)
                head = newNode;
            else
                prev.next = newNode;

            prev = newNode;

            if (i == skinSprites.Length - 1)
                newNode.next = head; // circular
        }
    }
}
```

```

SkinNode GetSkinNode(int index)
{
    SkinNode temp = head;

    for (int i = 0; i < index; i++)
    {
        temp = temp.next;
    }

    return temp;
}

void ApplySkin()
{
    playerSprite.sprite = current.skin;
}

```

- **Propósito:** Construye la estructura de datos de *skins* y aplica la selección.
- **Estructura de Datos:** Construye una **lista enlazada circular** a partir de un *array* de *sprites*.
- **Funcionalidad Clave:**
 - Start(): Carga el índice guardado en SkinSelector.savedSkinIndex y aplica el *sprite* al personaje (playerSprite).

6. SkinSelector.cs

```

public class SkinSelector : MonoBehaviour
{
    public static int savedSkinIndex = 0;

    public void SelectSkin(int index)
    {
        savedSkinIndex = index;
        Debug.Log("Skin seleccionada: " + index);
    }
}

```

- **Propósito:** Script simple para guardar de forma persistente la *skin* seleccionada.
- **Funcionalidad Clave:**

- Utiliza una variable **estática** (savedSkinIndex) para que el valor persista entre las escenas de UI y la escena de juego, sin necesidad de un DontDestroyOnLoad.

7. CuentaRegresiva.cs

```
public class CuentaRegresiva : MonoBehaviour
{
    public GameObject panelCuenta;           // Panel que contiene el texto
    public TextMeshProUGUI textoCuenta;      // Texto de la cuenta regresiva
    public float tiempoEntreNumeros = 1f;     // Tiempo entre cada número

    private void Start()
    {
        // Pausa el juego mientras hace la cuenta
        Time.timeScale = 0f;
        StartCoroutine(IniciarCuenta());
    }

    IEnumerator IniciarCuenta()
    {
        int contador = 3;
        textoCuenta.color = Color.white; // color fijo

        while (contador > 0)
        {
            textoCuenta.text = contador.ToString();
            yield return new WaitForSecondsRealtime(tiempoEntreNumeros);
            contador--;
        }

        textoCuenta.text = "¡GO!";
        yield return new WaitForSecondsRealtime(0.7f);

        // Quita el panel y reanuda el juego
        panelCuenta.SetActive(false);
        Time.timeScale = 1f;
    }
}
```

- **Propósito:** Muestra una cuenta regresiva al inicio de un nivel.
- **Funcionalidad Clave:**
 - **Pausa Inicial:** Establece Time.timeScale = 0f y ejecuta una **corutina**.

- **Tiempo Real:** La corrutina utiliza `yield return new WaitForSecondsRealtime()` para que el conteo se actualice a pesar de que el juego esté pausado.
- Al finalizar, reanuda el juego (`Time.timeScale = 1f`).