

«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 13



Clase N° 13 | Pools

Temario:

- Objects Pools

Objetivos de la clase

En esta clase, los estudiantes aprenderán el concepto de Object Pooling en Unity y cómo este contribuye a optimizar el rendimiento de los juegos. Se analizarán sus ventajas en la gestión eficiente de objetos, reduciendo la sobrecarga causada por la creación y destrucción constante de instancias. Mediante una implementación práctica, se desarrollará un sistema reutilizable de manejo de objetos que incluirá un Spawner capaz de generar enemigos de forma eficiente. Además, se programará un comportamiento básico de patrullaje para los enemigos, aplicando Object Pooling en un entorno funcional que simule un escenario real de videojuego.

¿Qué es un Object Pool en videojuegos?

El **Object Pool** (Patrón de Pool de Objetos) es un patrón de diseño que optimiza la gestión de instancias de objetos en memoria. En lugar de crear y destruir objetos constantemente (lo que puede ser costoso en términos de rendimiento y consumo de memoria), se mantiene un **grupo de objetos reutilizables**.

Cuando se necesita un objeto, en lugar de instanciar uno nuevo, se "toma" del pool (si hay disponible). Cuando ya no se necesita, se "devuelve" al pool en lugar de destruirlo. Esto es útil en juegos para manejar elementos como balas, enemigos, partículas y otros objetos que se crean y destruyen repetidamente.

Ejemplos de uso del Object Pooling en Unity 3D

1. Disparos en un juego tipo shooter

En un juego de disparos, cada vez que el jugador presiona el botón de disparo, normalmente se instancia un nuevo proyectil. Sin embargo, crear y destruir balas constantemente puede generar problemas de rendimiento, especialmente si el ritmo de disparo es alto.

✓ Solución:

Implementar un Object Pool que almacene una cantidad fija de balas y las reutilice. En lugar de crear una nueva instancia cada vez, se toma una bala inactiva del Pool, se activa y se reposiciona. Una vez que la bala impacta o sale del área de juego, se desactiva y vuelve al Pool.

2. Generación de enemigos en un Endless Runner

En un Endless Runner, los enemigos aparecen continuamente en la pantalla y desaparecen a medida que el jugador avanza. Si se crean y destruyen constantemente, esto puede afectar el rendimiento del juego en dispositivos móviles o con recursos limitados.

✓ Solución:

Utilizar un Pool de enemigos. Cuando un enemigo deja de ser visible, en lugar de destruirlo, se desactiva y se guarda en una lista de objetos inactivos. Cuando se necesita generar un nuevo enemigo, se reactiva uno del Pool con una nueva posición y estado.

Ejemplo sencillo aplicado a videojuegos:

Creemos un Pool de enemigos con un Spawner que los irá invocando. Los enemigos tendrán un movimiento de patrullaje muy sencillo y al tocar a nuestro personaje "desaparecerán".

Enemy Pool:

```
public class EnemyPool : MonoBehaviour
{
    public static EnemyPool Instance;
    public GameObject enemyPrefab;
    public int poolSize = 10;

    private List<GameObject> enemyList = new List<GameObject>();

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

    private void Start()
    {
        for (int i = 0; i < poolSize; i++)
        {
            GameObject enemy = Instantiate(enemyPrefab);
            enemy.SetActive(false);
            enemyList.Add(enemy);
        }
    }

    public GameObject GetEnemy(Vector3 position, Quaternion rotation)
    {
        foreach (GameObject enemy in enemyList)
        {
            if (!enemy.activeInHierarchy)
            {
                enemy.transform.position = position;
                enemy.transform.rotation = rotation;
                enemy.SetActive(true);
                return enemy;
            }
        }
    }
}
```

```


    }
}

GameObject newEnemy = Instantiate(enemyPrefab, position,
rotation);
enemyList.Add(newEnemy);
return newEnemy;
}

public void ReturnEnemy(GameObject enemy)
{
    enemy.SetActive(false);
}
}

```

Singleton para Acceso Global:

 Un **Singleton** es un patrón de diseño que garantiza que una clase tenga una única instancia en toda la aplicación y proporciona un punto de acceso global a ella.

En **C#** se implementa comúnmente utilizando una **clase estática** o una instancia controlada dentro de la propia clase.

```
public static EnemyPool Instance;
```

- Se declara una variable Instance estática para implementar el patrón Singleton.
- Esto permite que cualquier otro script acceda a la pool de enemigos sin necesidad de referencias manuales.

Variables de la Pool

```

public GameObject enemyPrefab;
public int poolSize = 10;
private List<GameObject> enemyList = new List<GameObject>();

```

- **enemyPrefab**: Prefab del enemigo que se usará para instanciar copias.
- **poolSize**: Cantidad inicial de enemigos que se crean y guardan.
- **enemyList**: Lista donde se almacenan los enemigos inactivos para ser reutilizados.

Configuración del Singleton en Awake:

Recordemos, ¿Qué es Awake()?

Awake() es un método especial de Unity que se ejecuta una sola vez, justo cuando el objeto se carga en la escena (antes de Start()).

Se usa principalmente para **inicializar variables y configurar referencias** antes de que el juego comience a ejecutarse.

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

¿Qué hace?

- Si todavía no existe una instancia (Instance == null), este objeto se convierte en la instancia única.
- Si ya existe otra, se destruye este objeto para evitar duplicados.

✅ **Resultado:** Solo hay una única **EnemyPool** activa en todo el juego.

Creación inicial de enemigos en Start

```
private void Start()
{
    for (int i = 0; i < poolSize; i++)
    {
        GameObject enemy = Instantiate(enemyPrefab);
        enemy.SetActive(false);
        enemyList.Add(enemy);
    }
}
```

💡 ¿Qué pasa acá?

- Se crea una cantidad inicial (poolSize) de enemigos.
- Cada uno se desactiva inmediatamente (SetActive(false)).
- Se guardan en la lista enemyList para su uso posterior.

Ventaja: No instanciamos enemigos cada vez que los necesitamos. Los tenemos preparados y listos para usar.

Obtener un Enemigo Disponible

```
public GameObject GetEnemy(Vector3 position, Quaternion rotation)
{
    foreach (GameObject enemy in enemyList)
    {
        if (!enemy.activeInHierarchy)
        {
            enemy.transform.position = position;
            enemy.transform.rotation = rotation;
            enemy.SetActive(true);
            return enemy;
        }
    }
    // Si llegamos aquí, no hay enemigos disponibles
    GameObject newEnemy=Instantiate(enemyPrefab,position, rotation);
    enemyList.Add(newEnemy);
    return newEnemy;
}
```

Busca un enemigo inactivo dentro de la lista enemyList, lo reactiva y lo devuelve listo para usar en la escena.

Declaración del Método:

- public: El método puede ser llamado desde otros scripts.
- GameObject: Retorna un objeto de tipo GameObject, que representa un enemigo.
- GetEnemy(Vector3 position, Quaternion rotation):
 - Recibe una posición (Vector3 position) donde se quiere colocar el enemigo.
 - Recibe una rotación (Quaternion rotation) para orientarlo correctamente.

Buscar un Enemigo Inactivo en la Pool:

- Se recorre toda la lista enemyList de enemigos.
- if (!enemy.activeInHierarchy):
 - Verifica si el enemigo está inactivo (Es decir, el enemigo está "guardado en la pool" y listo para reutilizarse).
- Si se encuentra un enemigo inactivo:

```
enemy.transform.position = position;
enemy.transform.rotation = rotation;
enemy.SetActive(true);
return enemy;
```

1. Se mueve el enemigo a la posición deseada.
2. Se le da la orientación (rotación) correcta.
3. Se activa con `SetActive(true)` para que aparezca en el juego.
4. **Se retorna el enemigo** inmediatamente (`return enemy;`).

Si no hay enemigos disponibles, se crea uno nuevo:

Ese caso se maneja después de este fragmento, fuera del `foreach`. Si no se encuentra ningún enemigo libre, el método creará uno nuevo.

- Instancia un nuevo enemigo.
- Lo agrega a la lista.
- Lo devuelve.

¿Por qué es útil esto?

Porque evita tener que instanciar un nuevo enemigo cada vez, lo cual consume memoria y tiempo de procesamiento. En cambio, se reutiliza uno que ya fue creado.

Retornar un Enemigo a la Pool

```
public void ReturnEnemy(GameObject enemy)
{
    enemy.SetActive(false);
}
```

Desactiva el enemigo para que pueda ser reutilizado en el futuro.

Enemy Spawner

Vamos ahora a crear un script que nos permita generar enemigos periódicamente, sin sobrecargar el sistema, porque generaremos enemigos cada cierto tiempo, limitando el número máximo de enemigos activos y reutilizarlos con **GetEnemy**.

```
public class EnemySpawner : MonoBehaviour
{
    public float spawnInterval = 3f;
    public int maxEnemies = 5;
    private float timer;
    private void Update()
    {
        timer += Time.deltaTime;
        if (timer >= spawnInterval)
        {
            timer = 0f;
            SpawnEnemy();
        }
    }
}
```



```

    }

    }

    void SpawnEnemy()
    {
        if (GameObject.FindGameObjectsWithTag("Enemy").Length <
maxEnemies)
        {
            Vector3 spawnPosition = new Vector3(Random.Range(-5, 5), 0,
Random.Range(-5, 5));
            Quaternion spawnRotation = Quaternion.identity;
            GameObject enemy =
EnemyPool.Instance.GetEnemy(spawnPosition, spawnRotation);
            enemy.tag = "Enemy";
        }
    }
}

```

Variables

```

public float spawnInterval = 3f;
public int maxEnemies = 5;
private float timer;

```

- **spawnInterval:** Tiempo entre cada spawn de enemigos.
- **maxEnemies:** Cantidad máxima de enemigos activos simultáneamente.
- **timer:** Controla el tiempo transcurrido desde el último spawn.

Controlar el tiempo (Spawn) en Update

```

private void Update()
{
    timer += Time.deltaTime;
    if (timer >= spawnInterval)
    {
        timer = 0f;
        SpawnEnemy();
    }
}

```

- Se incrementa timer con el tiempo transcurrido (Time.deltaTime).
- Cuando el timer alcanza spawnInterval, se resetea y se llama a SpawnEnemy().

Spawn de un Enemigo

```
void SpawnEnemy()
{
    if (GameObject.FindGameObjectsWithTag("Enemy").Length <
maxEnemies)
    {
        Vector3 spawnPosition = new Vector3(Random.Range(-5, 5), 0,
Random.Range(-5, 5));
        Quaternion spawnRotation = Quaternion.identity;
        GameObject enemy =
EnemyPool.Instance.GetEnemy(spawnPosition, spawnRotation);
        enemy.tag = "Enemy";
    }
}
```

- Si la cantidad de enemigos activos (GameObject.FindGameObjectsWithTag("Enemy")) es menor que maxEnemies, se genera un nuevo enemigo.
- Se determina una posición aleatoria dentro de un rango de -5 a 5 en X y Z, manteniendo Y en 0. (Vector3 spawnPosition)
- Se obtiene un enemigo de la pool y se le asigna la etiqueta "Enemy".

Enemy: Comportamiento del Enemigo

Este script define cómo se comporta un enemigo una vez que ha sido activado desde el Pool.

Se encarga de:

- Patrullar aleatoriamente alrededor de su punto de aparición.
- Detectar colisiones con el jugador.
- Retornarse automáticamente al EnemyPool al colisionar

```
public class Enemy : MonoBehaviour
{
    private Vector3 spawnPoint;
    public float patrolRadius = 5f;
    public float speed = 2f;
    private Vector3 targetPosition;

    private void Start()
    {
        spawnPoint = transform.position;
        SetNewTargetPosition();
    }
}
```

```

private void Update()
{
    Patrol();
}

void Patrol()
{
    transform.position = Vector3.MoveTowards(transform.position,
targetPosition, speed * Time.deltaTime);

    if (Vector3.Distance(transform.position, targetPosition) <
0.2f)
    {
        SetNewTargetPosition();
    }
}

void SetNewTargetPosition()
{
    Vector2 randomPoint = Random.insideUnitCircle * patrolRadius;
    targetPosition = spawnPoint + new Vector3(randomPoint.x, 0,
randomPoint.y);
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        Debug.Log("Muere Enemigo");
        EnemyPool.Instance.ReturnEnemy(gameObject);
    }
}
}

```

Variables

```

private Vector3 spawnPoint;
public float patrolRadius = 5f;
public float speed = 2f;
private Vector3 targetPosition;

```

- **spawnPoint:** Guarda la posición inicial del enemigo.(desde donde patrullará)
- **patrolRadius:** Radio en el que el enemigo puede moverse.
- **speed:** Velocidad de movimiento.

- **targetPosition:** Posición a la que el enemigo se moverá.

Definir Punto de Patrulla al Iniciar

```
private void Start()
{
    spawnPoint = transform.position;
    SetNewTargetPosition();
}
```

- Al activarse el enemigo, guarda su posición como spawnPoint.
- Luego llama a SetNewTargetPosition() para establecer un primer destino aleatorio dentro del área de patrulla.

●
📌 Importante: Este método se ejecuta cada vez que el enemigo es activado desde el Pool.

Patrulla Aleatoria

```
private void Update()
{
    Patrol();
}
void Patrol()
{
    transform.position = Vector3.MoveTowards(transform.position,
targetPosition, speed * Time.deltaTime);

    if (Vector3.Distance(transform.position, targetPosition) <
0.2f)
    {
        SetNewTargetPosition();
    }
}
```

- Vector3.MoveTowards() mueve al enemigo hacia targetPosition.
- Si el enemigo llega a su destino (distancia menor a 0.2f), se genera un nuevo objetivo.

Generar una nueva posición de patrulla aleatoria

```
void SetNewTargetPosition()
{
    Vector2 randomPoint = Random.insideUnitCircle * patrolRadius;
    targetPosition = spawnPoint + new Vector3(randomPoint.x, 0,
randomPoint.y);
}
```

- Random.insideUnitCircle genera un punto aleatorio dentro de un círculo.
- Se multiplica por el patrolRadius para que quede dentro del área deseada.
- Se convierte a Vector3, usando x y z (porque el juego es en 3D), y se deja y en 0.

📌 Esto genera un movimiento natural y aleatorio sin salir del radio definido.
Detectar al Jugador y Desaparecer

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        Debug.Log("Muere Enemigo");
        EnemyPool.Instance.ReturnEnemy(gameObject);
    }
}
```

- Si el enemigo colisiona con un objeto con la etiqueta "Player", se imprime "Muere Enemigo" y se activa la colisión.
- Luego, se retorna el enemigo al pool para ser reutilizado más tarde.

📌 Esto simula que el enemigo ha "muerto" o sido derrotado.

¿Donde implementamos los 3 códigos?

- El pool lo asignaremos en algún objeto como "Enemy Manager".
- El Spawn se podría colocar en cada zona donde se quiera tener enemigos periódicamente.
- El Enemy, se agrega en el prefab del enemigo.

Con esta implementación lograremos tener un manejo de enemigos utilizando el diseño de **Object Pool**, lo que generará el siguiente comportamiento:

- Cuando el **EnemySpawner** genera un enemigo, no lo instancia: **lo saca del Pool**.
- Cuando el enemigo "muere", no se destruye: **vuelve al Pool**.
- Al activarse nuevamente, el enemigo **retoma su patrulla desde una nueva posición**.

El Sistema de Reinvocación:



*El desarrollo del **proyecto Nexus** avanza con éxito. Los escenarios están tomando forma, las mecánicas están bien integradas y la jugabilidad se siente cada vez más fluida. Sin embargo, el equipo de **TalentoLab** ha encontrado un nuevo problema:*

*Durante las pruebas, han notado que cuando los enemigos aparecen en grandes cantidades, **el rendimiento del juego empieza a verse afectado**. El motor gráfico se sobrecarga cada vez que un nuevo enemigo es instanciado y destruido constantemente, generando **caídas de FPS y un consumo excesivo de memoria**.*

El equipo ha enviado un nuevo desafío:

"No podemos permitir que la simulación de Nexus sufra fallas por un mal manejo de los recursos. Necesitamos encontrar una forma eficiente de generar enemigos sin sobrecargar el sistema. Implementen un método para reciclar entidades en lugar de crearlas y eliminarlas constantemente."

Ejercicios prácticos:



En algunos niveles, hay plataformas móviles y dinámicas que aparecen y desaparecen constantemente. Cada vez que una plataforma se genera desde cero y luego se destruye, se repite el mismo problema que tenían los enemigos: un alto consumo de memoria y procesamiento innecesario.

Roberta te ha enviado una nueva solicitud:

"Si logramos optimizar los enemigos, también podemos hacer lo mismo con las plataformas. Implementa un sistema de Object Pooling para nuestras plataformas dinámicas y asegúrate que el rendimiento del juego siga siendo estable."

Materiales y recursos adicionales.

Object Pooling

<https://learn.unity.com/tutorial/introduction-to-object-pooling#>

Preguntas para reflexionar.

1. Volvamos a pensar. ¿Cuáles son los beneficios de la optimización?
 2. ¿Siempre es obligatorio optimizar?
 3. Piensen más ejemplos para realizar Objects Pools.
-

Próximos pasos.

En la próxima clase veremos una introducción a las Partículas, que nos permitirá añadir más estilos de Feedback a nuestros juegos.



Buenos Aires
aprende
Agencia de Habilidades para el Futuro

BA Buenos
Aires
Ciudad