«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 13

Buenos Aires Ciudad

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 su impacto en la optimización del rendimiento. Se explorarán sus ventajas en la gestión eficiente de objetos, evitando la creación y destrucción constante de instancias. A través de una implementación práctica, se desarrollará un sistema reutilizable para manejar objetos en el juego, incluyendo un **Spawner** que genere enemigos de manera eficiente. Además, se programará un comportamiento básico de patrullaje para los enemigos, aplicando el Object Pooling en un escenario funcional.

¿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 en Unity 3D:

1. Disparos en un shooter

En un juego de disparos, cada vez que el jugador dispara, se podría instanciar un nuevo proyectil. Sin embargo, crear y destruir balas constantemente puede generar problemas de rendimiento.

> ✅ **Solución:** Un Object Pool almacena una cantidad fija de balas y las reutiliza cuando se necesitan, evitando la creación y destrucción repetitiva.

2. Generación de enemigos en un Endless Runner

En un **Endless Runner**, los enemigos aparecen constantemente en la pantalla y desaparecen cuando el jugador avanza. En lugar de destruir los enemigos y crear nuevos, se pueden **desactivar y reactivar desde un Pool**.

> ✅ **Solución:** Se guarda una lista de enemigos inactivos que se van reactivando a medida que se necesitan.

Ejemplo sencillo aplicado a videojuegos:

Creamos 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 “desapareran”.

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:

Recordemos qué es: 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 instanciará.
- `poolSize`: Cantidad inicial de enemigos en la pool.

- `enemyList`: Lista que almacena los enemigos inactivos para reutilizarlos.

Configuración del Singleton en Awake:

¿Qué era el Awake?
`Awake()` es un **método especial de Unity** que se ejecuta cuando un objeto es instanciado en la escena, **antes de que comience el juego** y antes de que se llame a `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);
    }
}
```

- Si `Instance` es `null`, se asigna a este objeto.
- Si ya existe otra instancia, se destruye el objeto duplicado, asegurando que solo haya una `EnemyPool` activa.

Creación de la Pool en Start

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

- Se instancian `poolSize` enemigos inactivos y se almacenan en la lista `enemyList`.

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;
            }
        }
```

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, no está en la escena actualmente).
- Si se encuentra un enemigo inactivo:
    1. **Se actualiza su posición y rotación** para colocarlo en el nuevo punto de spawn.
    2. **Se activa el enemigo (`SetActive(true)`)** para que aparezca en la escena.
    3. **Se retorna el enemigo** inmediatamente (`return enemy;`).

Si No Hay Enemigos Disponibles, Se Crea Uno Nuevo:

- Si **no había enemigos inactivos**, entonces:
    a. Se **crea un nuevo enemigo** usando `Instantiate(enemyPrefab, position, rotation)`, en la posición y rotación especificadas.

b. Se **agrega el nuevo enemigo a la lista (`enemyList.Add(newEnemy)`)** para futuras reutilizaciones.
c. **Se retorna el nuevo enemigo**.

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

```
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.

Control del 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.
- Se obtiene un enemigo de la pool y se le asigna la etiqueta `"Enemy"`.

Enemy: Comportamiento del Enemigo

```
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.
- `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();
    }
```

- Se guarda la posición inicial como `spawnPoint`.
- Se establece un primer objetivo de patrulla.

Patrulla Aleatoria

```
    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

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

`Random.insideUnitCircle` es una propiedad de la clase `Random` en Unity que devuelve un punto aleatorio dentro de un círculo de radio 1 en el plano 2D. El resultado es un `Vector2`, es decir, un par de coordenadas `(x, y)` que están dentro del círculo unitario. Esto también lo podriamos hacer Random.insideUnitSphere que crea una esfera en vez de un círculo.

- `Random.insideUnitCircle * patrolRadius` genera un punto aleatorio dentro del radio.

- Se convierte a `Vector3` para que X y Z cambien mientras Y se mantiene en 0.

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".
- Luego, se retorna el enemigo al pool para ser reutilizado más tarde.

Finalmente, al implementar estos 3 códigos:

- El pool en algún objeto como “Enemy Manager”.
- El Spawn lo pueden colocar en cada zona donde quieran tener enemigos.
- El Enemy, obviamente, en el enemigo.

Lograremos tener un manejo de enemigos utilizando el diseño/sistema de **Object Pool**.

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
Buenos
Aires
Ciudad