Performance Optimisations with Object Pooling

Intermediate

Rob Homewood

Overview

What are we doing here

- What problem are we trying to solve
- How will we solve the problem
- Example code for solution
- Caveats to keep in mind
- Final considerations

The Problem

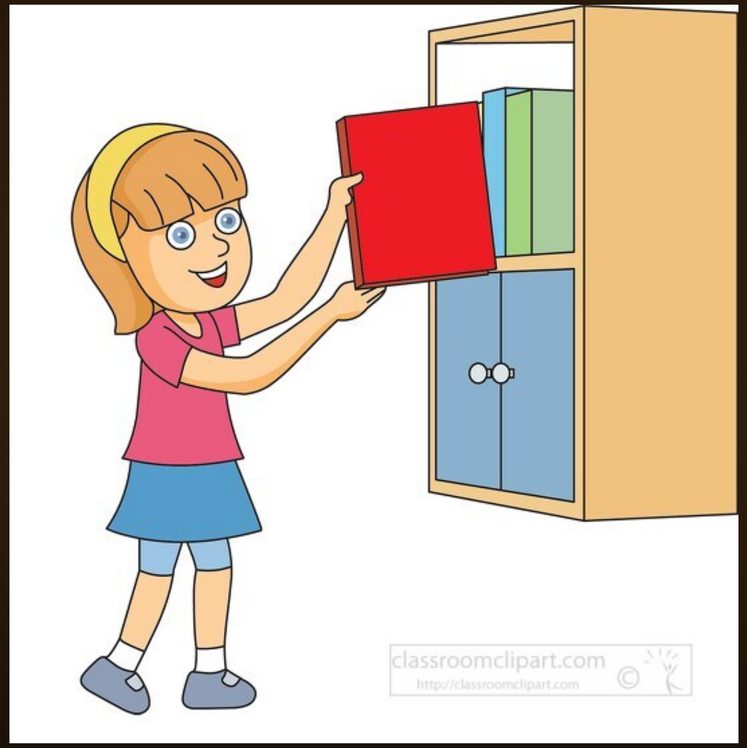
The Hidden Cost of Instantiate & Destroy

- Many games heavily rely on creating/destroying GameObjects:
  - Projectiles, Explosions, Enemies, UI Elements
- Instantiate() & Destroy() cause performance issues:
  - Frame rate drops (stuttering), especially on mobile.
  - Garbage collection spikes (GC allocation), leading to pauses.

Object Pooling:

Recycling, Not Just Performance

- A technique to reuse pre-created objects instead of constantly creating/destroying them.
- Reduces the overhead associated with memory allocation.
- Analogy:
  - Library Books. Take them out of the collection, but return them back when you are done.



Core Components of an Object Pool

- Pool:
  - The container holding inactive/available objects.
  - (List, Queue, Stack, or HashSet - consider trade-offs).
  - Can be of fixed or dynamic size
- Spawn:
  - Taking an object from the pool (making it active).
  - Potentially re-initializing or resetting state.
- Despawn:
  - Returning an object to the pool (making it inactive).
  - Clean up.

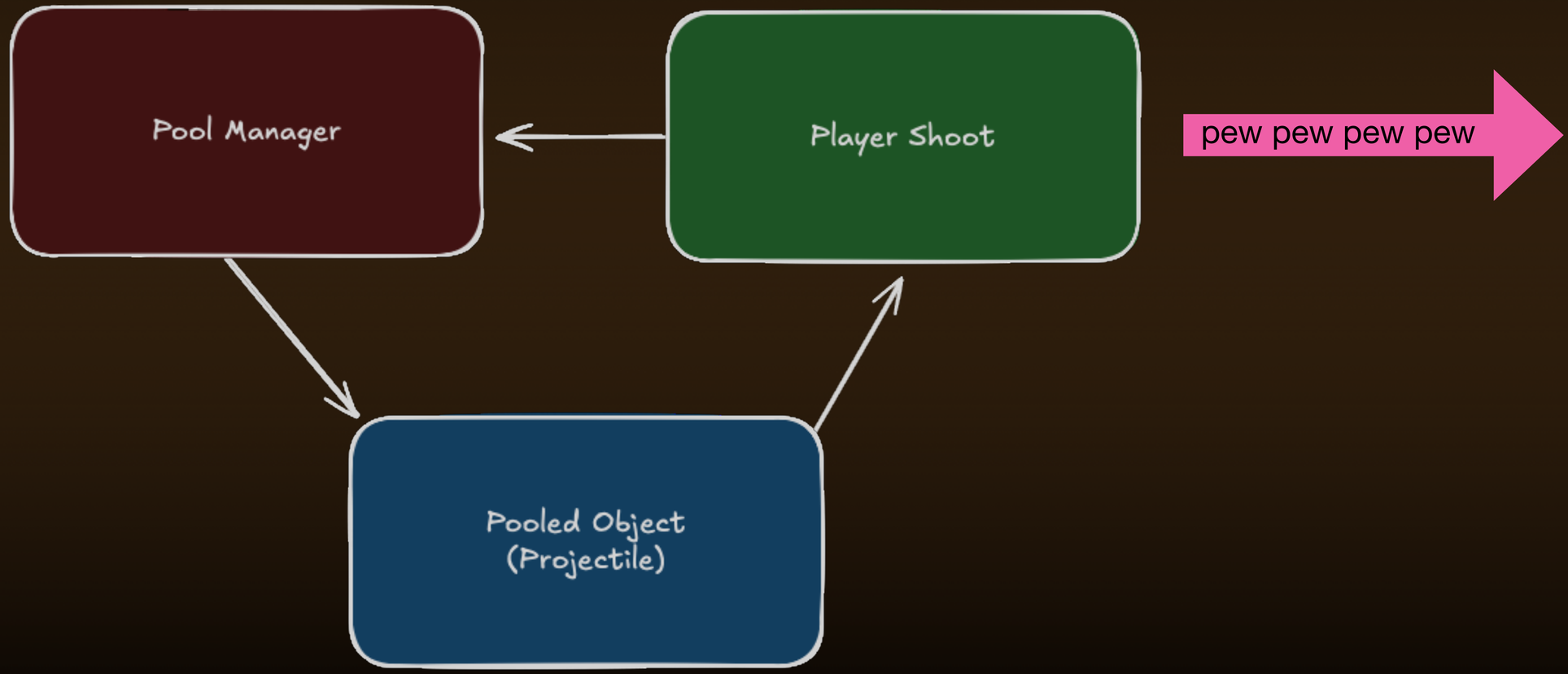
Benefits

The Power of Object Pooling

- Reduced Instantiation/Destruction Overhead:
  - Avoids the CPU cost of object creation.
- Lower Garbage Collection:
  - Reduces GC frequency, preventing frame rate spikes.
- Enforces Limits:
  - Control the number of active objects in the game.
- Lifecycle Management:
  - Reset or clean up objects when despawned.

Pattern Architecture

How the pieces fit together



Code Example: Setup

Pool Manager Class

- Pool that can handle a game object type
  - Key Fields:
    - Prefab: GameObject template
    - Size: Default Pool Size
    - Pool Collection: The GameObject pool
  - Key Methods
    - Initialize(): Pre-instantiates objects, deactivates, and adds to pool
    - Spawn(): Get the object from the pool to be used in the scene
    - Despawn(): Returns to the pool

```
public class PoolManager : MonoBehaviour
{
    [SerializeField] GameObject pooledPrefab;
    [SerializeField] private int size = 10;
    [SerializeField] private List<GameObject> pool;

    void Start()
    {
        Initialize();
    }
```

Code Example: Initialize

Pool Manager class

- Creates the objects
- Populates pool with created objects

```
private void Initialize()
{
    pool = new List<GameObject>();
    for (int i = 0; i < size; i++)
    {
        GameObject obj = Instantiate(pooledPrefab);
        obj.GetComponent<PooledObject>().Initialize(this);
        obj.SetActive(false);
        pool.Add(obj);
    }
}
```

Code Example: Spawn

Pool Manager class

- Retrieves a new object
- Ensures object is ready
- Expands pool if all in use

```
public GameObject Spawn()
{
    foreach (GameObject obj in pool)
    {
        if (!obj.activeInHierarchy)
        {
            obj.SetActive(true);
            PooledObject pooledObject = obj.GetComponent<PooledObject>();
            pooledObject.ResetPooledObject();

            return obj;
        }
    }

    // If the pool is empty, instantiate a new object
    GameObject newObj = Instantiate(pooledPrefab);
    newObj.GetComponent<PooledObject>().Initialize(this);
    pool.Add(newObj);
    return newObj;
}
```

Code Example: Despawn

Pool Manager Class

- Disables object in the scene
- Returns the object to the pool

```
public void Despawn(GameObject obj)
{
    obj.SetActive(false);
    obj.transform.position = transform.position;
}
```

Code Example: Setup

Pooled Object Class

- Pool that can handle a game object type
  - Key Fields:
    - Pool Manager: Reference to the pool
    - Resettable fields
  - Key Methods
    - Initialize(): Assigns pool reference
    - Reset(): Return any resettable fields to their original state

```
public class PooledObject : MonoBehaviour
{
    [SerializeField] float projectileSpeed = 20f;
    [SerializeField] float projectileLifeTime = 2f;

    private PoolManager _poolManager;

    int currentAnimationFrame = 0;
    float timeAlive = 0f;
```

Code Example: Object Request

Player Shoot Class

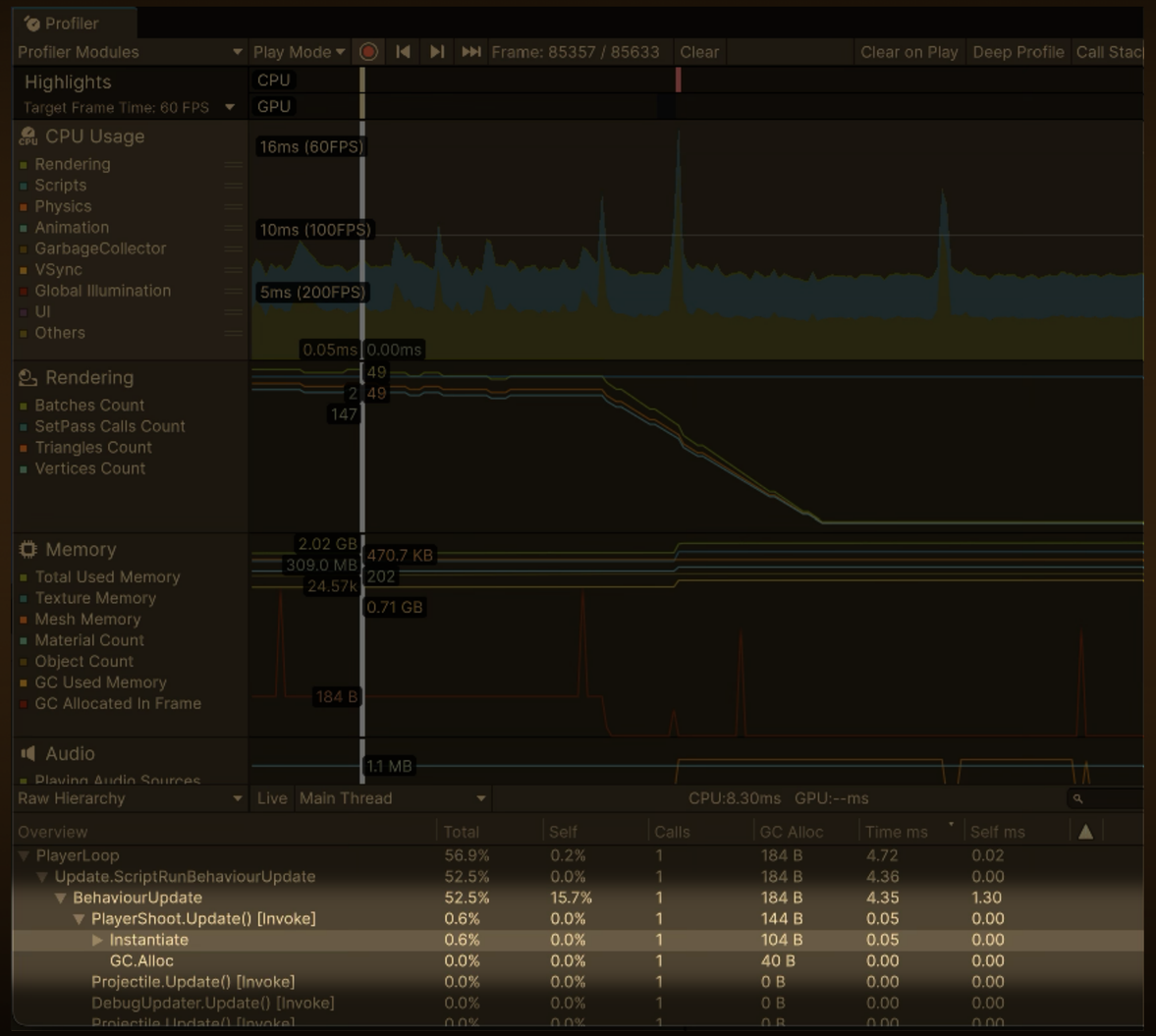
- Traditionally we would Instantiate to create an object
- Instead, now we just request it from the pool and position it where we want it to appear

```
public class PlayerShoot : MonoBehaviour
{
    [SerializeField] PoolManager poolManager;
    private void Shoot()
    {
        GameObject newProjectile = poolManager.Spawn();
        newProjectile.transform.position = transform.position;
    }
```

Impact:

Unpooled Garbage Allocations

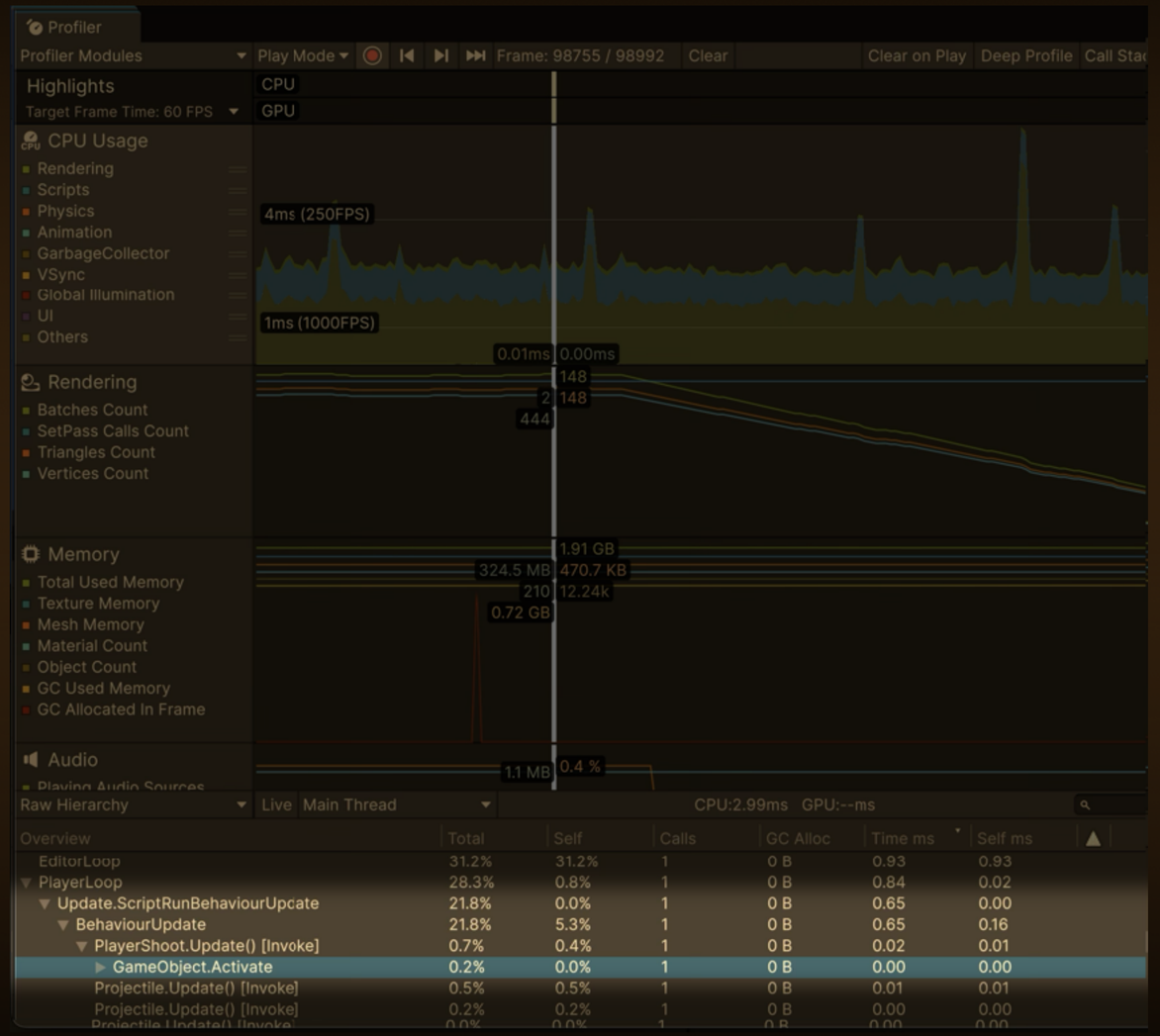
- 144B GC Allocations per frame



Impact:

Pooled Garbage Allocations

- 0B GC Allocations per frame



BUT... Object Pooling Isn't Always a Silver Bullet!

Object Pooling - Caveats

- Could hurt performance if:
  - Pooling only a few simple objects (the overhead might outweigh the benefit).
  - Pool is accessed frequently from different threads without proper synchronisation.
  - Memory fragmentation occurs, leading to increased memory use/GC pressure.
- Requires careful analysis and profiling.

Considerations

Just a few things to think about..

- How are objects stored in the pool (data structures)?
- How do we retrieve the objects from the pool?
- How best can we expand the pool when it is outgrown?

Thank You!

Happy Pooling 🤗

Code examples and further references:
https://github.com/robrab2000/ObjectPoolingExampleProject

Rob Homewood