Example of a situation needing Object Pooling

When the spaceship shoots, it creates several bullets that travel up the screen, it will then destroy the objects if they collide with another object or leave the screen. This creates a considerable drag on the CPU and therefor, Object Pooling would be a great candidate to optimize in this situation as it will alleviate the need to constantly create and destroy objects during runtime.

Creating an Object Pooling Script

1. Create a new script and attach it to your game controller (**Figure 02**)

```
public class ObjectPool : MonoBehaviour {

    public static ObjectPool SharedInstance;
    public List<GameObject> pooledObjects;
    public GameObject objectToPool;
    public int amountToPool;

    void Awake()
    {
        SharedInstance = this;
    }

    private void Start()
    {
        pooledObjects = new List<GameObject>();
        for (int i = 0; i < amountToPool; i++)
        {
            GameObject obj = (GameObject)Instantiate(objectToPool);
            obj.SetActive(false);
            pooledObjects.Add(obj);
        }

    }

}
```


Figure 02: Initial setup of the Object Pool

This simple setup allows you to specify a GameObject to pool and a number to pre-instantiate. The For Loop will instantiate the objectToPool the specified number of times in numberToPool. Then the GameObjects are set to an inactive state before adding them to the pooledObjects list.

2. Select the game controller which contains the script you just created. It'll have the pooledObjects and the amountToPool where you can set both respectively. Dragging the bullet prefab to pooledObjects will tell the script what object you wish the pool to consist of.

3. Set the amountToPool to a relatively large number such as 20. The reason for this is we want to make sure we have enough GameObjects to work with (**Figure 03**).

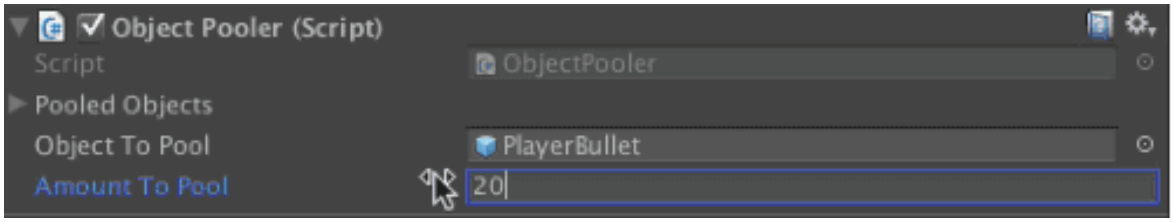

Figure 03: Setting the Object Pool script

Now the script will always create 20 PlayerBullets before the game even runs. This way there will always be a collection of pre-instantiated bullets for our use. In order to take advantage of this we need to do two more things at minimum

4. Reopen the Object Pool script you created so you can create a new function to call from other scripts to utilize the Object Pool. This will help utilize the idea of not needing to redundantly instantiate and destroy objects during runtime. This will also allow the other scripts to set the objects to active or inactive states which creates a graceful process where we adhere to the Object Pooling design

```
public GameObject GetPooledObject()
{
    for (int i = 0; i < pooledObjects.Count; i++)
    {
        if (!pooledObjects[i].activeInHierarchy)
        {
            return pooledObjects[i];
        }
    }
    return null;
}
```

Figure 04: Additional Code to support Object Pooling

5. Go into the scripts that instantiate the bullets. Here you will want to replace any code that instantiates the bullets, such as: '`Instantiate(playerBullet, turret.transform.position, turret.transform.rotation);`'

Use the following code to replace the Instantiate calls:

```
GameObject bullet = ObjectPooler.SharedInstance.GetPooledObject();

  if (bullet != null) {

    bullet.transform.position = turret.transform.position;
```

```
    bullet.transform.rotation = turret.transform.rotation;

    bullet.SetActive(true);

  }
```

The code will request a GameObject to become active, and set the properties of that given GameObject. It removes the need to instantiate a new object and efficiently requests and acquires a GameObject that is only pre-instantiated, relieving the burden from the CPU to having to create and destroy a new one.