# Object Pool Manager

## What is the ObjectPoolManager

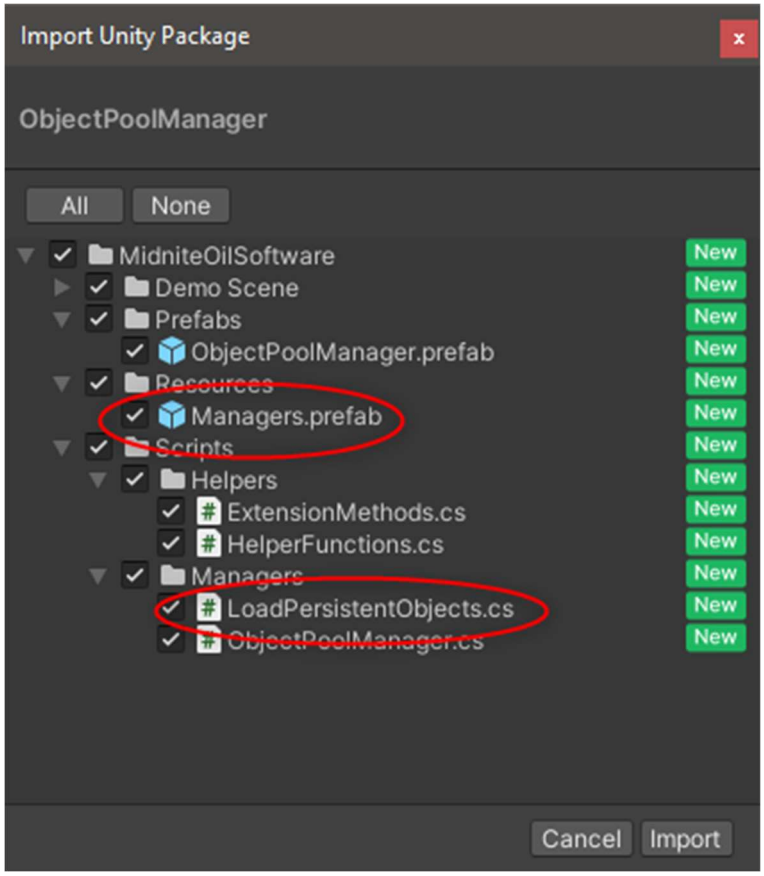
The ObjectPoolManager is a lightweight, easy-to-use system for adding object pooling to your game. It is designed specifically to work with prefabs (unlike the built-in object pooling system provided by Unity which requires you to use exact types).

Objects are only instantiated as needed and only take up storage in the pool when they are despawned. This makes retrieving pooled objects extremely performant because you don't have to search for inactive objects in the pool.

## Installation

To install the Object Pool Manager simply download and import the package. The full package includes a sample scene that demonstrates how to use the manager.

If you want the manager to be automatically loaded when your game starts (and persist across scenes) make sure you import the Managers prefab and the LoadPersistentObjects.cs script.




If you exclude those files you will need to explicitly add the ObjectPoolManager prefab to a scene to use it.

## Using the pool manager

The ObjectPoolManager is a static class which follows the singleton pattern. It is designed to work with prefabs. These can be as simple as a SerializedField of type GameObject:

```
[SerializedField]
GameObject _projectilePrefab;
```

Instead of directly instantiating the GameObject you can use the ObjectPoolManager.SpawnGameObject() method. There are multiple overloads of this method which takes optional parameters akin to the GameObject.InstantiateMethod().

```
GameObject projectile =
ObjectPoolManager.SpawnGameObject(_projectilePrefab,
transform.position, Quaternion.Identity);
```

To despawn an object simply call the ObjectPoolManager.DespawnGameObject() method passing in the object to be despawned.

```
public class DespawnAfterDelay : MonoBehaviour
{
   [SerializeField] [Range(5f, 30f)] float _despawnDelay = 10f;

   private void OnEnable()
   {
      Invoke(nameof(Despawn), _despawnDelay);
   }

   private void Despawn()
   {
      ObjectPoolManager.DespawnGameObject(gameObject);
   }
}
```

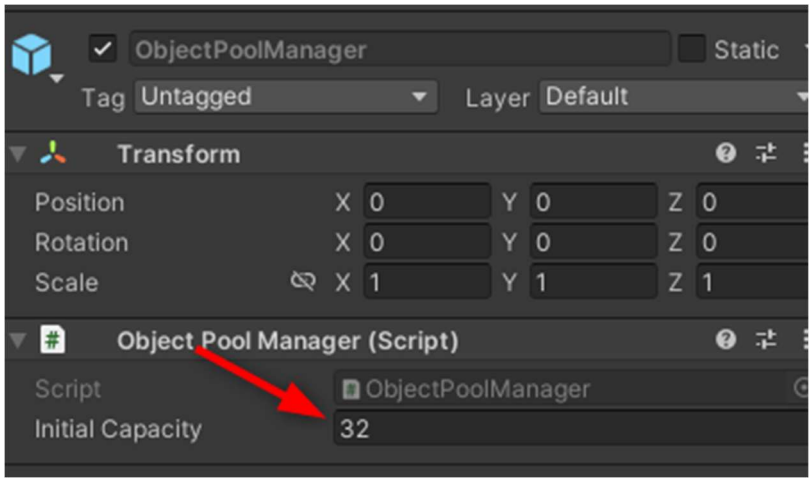
There are also methods to permanently destroy a GameObject and to clear the Object Pool.

```
public static void PermanentlyDestroyGameObjectsOfType(GameObject
prefab);
```

```
public static void EmptyPool();
```

## Initial Pool Capacity

By default, the initial capacity for each pool type is 32 and it will automatically double in size if that capacity is exceeded. You can change the default initial capacity on the ObjectPoolManager prefab in the inspector:




You can also change it in code with the SetInitialCapacity()method:

```
ObjectPoolManager.SetInitialCapacity(1000);
```

You should be sure to do this before adding any elements to the pool.

## IDespawnedPoolObject and IRetreivedPoolObject Interfaces

The `IDespawnedPoolObject` interface has a single method: `void ReturnedToPool()`

The ObjectPoolManager will look for this interface on any object it's returning to the pool and call this method if it exists. This is the idea place to do things like reset the velocity & angular velocity of a RigidBody when despawning an object so it won't go flying off in a random direction the next time it's spawned from the pool.

```
public void ReturnedToPool()
{
    if (_rigidBody)
    {
        _rigidBody.velocity = _rigidBody.angularVelocity = Vector3.zero;
    }

    if (_rigidbody2D)
    {
        _rigidbody2D.velocity = Vector2.zero;
        _rigidbody2D.angularVelocity = 0f;
    }
}
```

The `IRetrievedPoolObject` interface also has a single method:

```
void RetrievedFromPool(GameObject prefab);
```

This method takes the prefab used to spawn/retrieve the object as a parameter and can be used to copy default values to the object being retrieved from the pool. Consider this example where a "fractured" asteroid is spawned when an asteroid is destroyed. Explosion force is applied to the fragments sending them flying off in different directions. If we want to reuse the fractured asteroid object we need to reassemble the fragments.

```
public class ReassembleObject : MonoBehaviour, IRetrievedPoolObject
{
  Transform _transform;

  void Awake()
  {
     _transform = transform;
  }

  public void RetrievedFromPool(GameObject prefab)
  {
     _transform.position = prefab.transform.position;
     _transform.rotation = prefab.transform.rotation;
     ReconstructFragments(prefab);
  }

  void ReconstructFragments(GameObject prefab)
  {
     for (int i = 0; i < _transform.childCount; ++i)
     {
        var childTransform = _transform.GetChild(i);
        var prefabTransform = prefab.transform.GetChild(i);
        childTransform.position = prefabTransform.position;
        childTransform.rotation = prefabTransform.rotation;

        var childRigidBody = childTransform.GetComponent<Rigidbody>();
        childRigidBody.velocity = Vector3.zero;
        childRigidBody.angularVelocity = Vector3.zero;
      }
   }
}
```

In this example, the ReassembleObject component can simply be dropped onto the prefab object (e.g. the FracturedAsteroid).