Unity Object Pooling System Documentation (Beta Version)

📌 Important Notes

- **AI-Generated Documentation:**
  This documentation is generated with the assistance of AI. For accurate understanding and implementation, refer to the official example scripts that demonstrate the proper usage of the pooling system.

- **Pool Cleanup Functionality (WIP):**
  Please be aware that the automatic pool cleanup system is not fully implemented at this stage. Exercise caution when relying on automatic cleanup behavior.

- **Documentation Updates:**
  Future updates to the pooling system will not be reflected in this version of the documentation. A separate changelog or supplementary document will be provided for future changes and enhancements.

- **LRU (Least Recently Used) Pooling Strategy:**
  An LRU-based cleanup strategy is available but may introduce performance overhead. It is best suited for scenarios where object usage patterns are highly unpredictable or bursty.
  For example, if your game occasionally spawns a sudden burst of particle effects due to a chain explosion, using LRU makes sense.
  However, avoid using LRU for consistently active or predictable systems such as looping waterfall splashes—more optimized, less expensive alternatives will be introduced later for such use cases (coming soon).

For Unity Asset Store Tool

This document provides a comprehensive guide to using the Object Pooling System, a powerful tool designed to optimize performance in Unity applications by efficiently managing the instantiation and destruction of objects. This beta version offers core functionalities for creating, managing, and accessing object pools, alongside convenient editor tools and example implementations to get you started.

---

1. Introduction

Object pooling is a fundamental design pattern used in game development to manage a collection of pre-instantiated objects. Instead of repeatedly creating and destroying objects (which can be performance-intensive), objects are returned to a "pool" when no longer needed and then reused when new instances are required. This significantly reduces memory allocations, garbage collection overhead, and CPU spikes, leading to smoother gameplay and improved frame rates, especially in scenarios with many short-lived objects (e.g., bullets, particles, enemies, UI elements).

This Unity Asset Store tool provides a robust, flexible, and easy-to-integrate object pooling solution designed to enhance your application's performance.

---

2. Core Concepts

The Object Pooling System is built around three fundamental components that form its core:

- **Central Static Class:** This is the global entry point and centralized registry for all object pools within your application. It provides methods to create new pools, find existing pools, and manage the overall collection of pools.
- **PoolObject Class:** This class represents an individual object pool. Each PoolObject instance is responsible for managing a collection of data structs, which in turn encapsulate the actual pooled objects. It handles the logic for retrieving available objects, forcefully reusing objects (even if in use), injecting new objects into the pool, and providing statistics.
- **data Struct:** A lightweight, internal structure that holds metadata about each individual object stored within a PoolObject. It tracks the object's occupancy status, a unique identifier within its pool, the object itself, and its last usage time.

The system internally uses a List<data> and a Dictionary<int, data> within each PoolObject for efficient storage and lookup of pooled items.

---

3. Core System Reference

This section details the fundamental components of the Object Pooling System. These scripts (Central.cs, PoolObject.cs, data.cs) form the backbone and are designed to be highly flexible, allowing you to pool *any* Unity object (e.g., GameObject, ScriptableObject, MonoBehaviour instances) or even custom C# class instances.

3.1. Central Static Class

The Central static class is your primary interface for global pool management.

3.1.1. Pool Management Methods

- **public static PoolObject CreateNewPool(Type type, int idealsize, string name = "")**
  - **Description:** Creates and registers a new object pool. This is the first step to setting up a pool for a specific type of object. Each pool is assigned a unique internal ID, and you can optionally provide a custom name. If no name is given, the pool will automatically use the Type's name. The idealsize parameter helps the PoolObject pre-allocate its internal collections for better performance.
  - **Parameters:**
    - type: The System.Type of the objects this pool will manage (e.g., typeof(GameObject), typeof(MyCustomClass)). All objects subsequently injected into this pool *must* be compatible with this type.
    - idealsize: An integer indicating the initial suggested capacity for the pool. This value is used by the PoolObject to size its internal collections, minimizing reallocations.

        - name (optional): A string to assign a custom, human-readable name to the pool. Defaults to type.Name if empty.
    - **Returns:** A PoolObject instance. You will use this returned PoolObject to interact directly with the newly created pool (e.g., to get objects, inject objects).
    - **Example:**
    - C#

```
// Create a pool for GameObjects with an initial suggested capacity of 10
PoolObject myGameObjectPool = Central.CreateNewPool(typeof(GameObject), 10,
"PlayerBullets");

// Create a pool for a custom C# class (e.g., a data structure)
PoolObject myDataPool = Central.CreateNewPool(typeof(MyCustomDataClass), 50); //
Name will be "MyCustomDataClass"
```

    -
- **public static void NewPoolObjectsList(bool keepCurrentObjects = true)**
    - **Description:** Resets or clears the list of active object pools managed by Central. This is useful for managing memory during scene transitions or when you need a clean slate of pools.
        - If keepCurrentObjects is false, all currently registered pools are discarded immediately from Central's management.
        - If keepCurrentObjects is true (default), the current pools are moved to an internal backup list, and a new empty list of pools is initialized. This allows for a controlled reset while retaining the ability to restore previous pools later.
    - **Parameters:**
        - keepCurrentObjects: A boolean flag. Set to true to move current pools to a backup list, or false to discard them.
    - **Example:**
    - C#

```
// Clear all existing pools without backup (they will no longer be accessible via Central)
Central.NewPoolObjectsList(false);

// Create a new empty pool list, moving current pools to backup for potential restoration
Central.NewPoolObjectsList(true);
```

    -
- **public static void RestoreFromBackup()**
    - **Description:** Restores the main list of object pools from the internal backup list. This method is typically used after calling NewPoolObjectsList(true) to bring back a previously saved set of pools. The backup list is cleared after restoration.
    - **Example:**
    - C#

```
// ... (pools were moved to backup using NewPoolObjectsList(true)) ...
Central.RestoreFromBackup(); // Bring back the backed-up pools to the main list
```

    -
- **public static void RemoveById(PoolObject pool)**
    - **Description:** Removes a specific PoolObject instance from the active list of pools managed by Central. This effectively "unregisters" the pool from the system. Note that this method requires the PoolObject instance itself, not just its ID.
    - **Parameters:**
        - pool: The PoolObject instance to be removed from Central's management.
    - **Example:**
    - C#

```
PoolObject mySpecificPool = Central.FindPoolByName("MySpecificPool");
if (mySpecificPool != null)
{
    Central.RemoveById(mySpecificPool); // Remove the pool from Central's management
}
```

    -

3.1.2. Pool Retrieval Methods (Read-Only)

These methods allow you to retrieve PoolObject instances from Central for further interaction.

- **public static IReadOnlyList<PoolObject> GetPools()**
    - **Description:** Provides a read-only list of all currently active object pools managed by Central. This is useful for inspecting the current state of your pools without inadvertently modifying them.
    - **Returns:** An IReadOnlyList<PoolObject> containing all currently registered PoolObject instances.
    - **Example:**
    - C#

```
IReadOnlyList<PoolObject> currentPools = Central.GetPools();
foreach (var pool in currentPools)
{
    Debug.Log($"Pool Name: {pool.name}, ID: {pool.id}, Type: {pool.type?.Name ?? "N/A"}");
}
```

    -
- **public static PoolObject FindPoolById(int id)**
    - **Description:** Searches for an object pool by its unique integer ID within the *current* active pool list managed by Central.
    - **Parameters:**
        - id: The unique integer ID of the pool to find.
    - **Returns:** The PoolObject if found, otherwise null.
    - **Example:**
    - C#

```
PoolObject foundPool = Central.FindPoolById(0);
```

```
if (foundPool != null)
{
    Debug.Log($"Found pool with ID 0: {foundPool.name}");
}
```

- **public static PoolObject FindPoolByIdAny(int id)**
  - **Description:** Searches for an object pool by its unique integer ID in both the *current* active pool list and the *backup* pool list managed by Central. This is useful if you've recently reset the main pool list using NewPoolObjectsList(true) but need to access a pool that was moved to backup.
  - **Parameters:**
    - id: The unique integer ID of the pool to find.
  - **Returns:** The PoolObject if found in either the current or backup list, otherwise null.
  - **Example:**
  - C#

```
PoolObject foundAnyPool = Central.FindPoolByIdAny(5);
if (foundAnyPool != null)
{
    Debug.Log($"Found pool with ID 5 (could be current or backup): {foundAnyPool.name}");
}
```

- **public static PoolObject FindPoolByName(string name)**
  - **Description:** Searches for an object pool by its assigned string name within the *current* active pool list managed by Central.
  - **Parameters:**
    - name: The string name of the pool to find.
  - **Returns:** The PoolObject if found, otherwise null.
  - **Example:**
  - C#

```
PoolObject namedPool = Central.FindPoolByName("GameObjectsPool");
if (namedPool != null)
{
    Debug.Log($"Found pool by name 'GameObjectsPool': {namedPool.id}");
}
```

- **public static PoolObject FindPoolByNameAny(string name)**
  - **Description:** Searches for an object pool by its assigned string name in both the *current* active pool list and the *backup* pool list managed by Central.
  - **Parameters:**
    - name: The string name of the pool to find.
  - **Returns:** The PoolObject if found in either list, otherwise null.
  - **Example:**
  - C#

```
PoolObject anyNamedPool = Central.FindPoolByNameAny("Bullet");
if (anyNamedPool != null)
{
   Debug.Log($"Found pool by name 'Bullet' (could be current or backup):
{anyNamedPool.id}");
}
```

-

3.2. PoolObject Class

The PoolObject class is the core of individual pool management. Once you obtain a PoolObject instance from Central.CreateNewPool or Central.FindPool..., you use its methods to interact with the objects within that specific pool.

3.2.1. Properties

- public string name;: The name of this pool.
- public int id;: The unique ID of this pool.
- public List<data> Objects;: The internal list holding all data structs for this pool.
- public Type type;: The System.Type of objects managed by this pool.
- public int initialCapacity;: The initial suggested capacity for the pool's internal collections.

3.2.2. Constructor

- **public PoolObject()**
  - **Description:** Initializes a new PoolObject instance. It sets up the internal Objects list and objectDictionary with the initialCapacity provided during creation via Central.CreateNewPool.

3.2.3. Object Retrieval Methods

These methods allow you to retrieve objects from the pool based on their availability and usage history.

- **public data GetObject()**
  - **Description:** Retrieves an *unused* object from the pool. If an unused object is found, its IsOcupied status is set to true, and its time is updated to Time.time. If no unused object is available, it returns a new, empty data struct (where obj will be null). This is the preferred method for getting objects without forcing reuse.
  - **Returns:** A data struct representing an unused pooled object. You should check data.obj for null to determine if an object was successfully retrieved.
  - **Example:**
  - C#

```
PoolObject bulletPool = Central.FindPoolByName("BulletPool");
if (bulletPool != null)
{
   data bulletData = bulletPool.GetObject();
   if (bulletData.obj != null)
   {
```

```
        GameObject bullet = (GameObject)bulletData.obj; // Cast to your actual type
        bullet.SetActive(true); // Assuming it's a GameObject, activate it
        // ... use the bullet ...
    }
    else
    {
        Debug.LogWarning("No unused bullets available in the pool. Consider increasing initialCapacity or instantiating a new object if dynamic growth is desired.");
        // If you need more objects than available, you might instantiate a new one here
        // and then inject it into the pool using InjectNewData.
    }
}
```

- **public data GetForcedObject()**
  - **Description:** Retrieves an object from the pool, prioritizing unused objects. If no unused objects are available, it forcefully reuses the *Least Recently Used (LRU)* object that is currently IsOcupied. The IsOcupied status of the returned object is set to true, and its time is updated. This method ensures you always get an object, even if it means interrupting an active one.
  - **Returns:** A data struct representing either an unused object or the least recently used object.
  - **Example:**
  - C#

```
PoolObject fxPool = Central.FindPoolByName("EffectPool");
if (fxPool != null)
{
    data effectData = fxPool.GetForcedObject();
    if (effectData.obj != null)
    {
        GameObject fx = (GameObject)effectData.obj;
        fx.SetActive(true);
        // ... play effect ...
    }
}
```

- **public data[] GetUnUsedObjs()**
  - **Description:** Retrieves an array of all currently *unused* objects from the pool.
  - **Returns:** An array of data structs for all available (unused) objects.
  - **Example:**
  - C#

```
PoolObject enemyPool = Central.FindPoolByName("EnemyPool");
if (enemyPool != null)
{
    data[] availableEnemies = enemyPool.GetUnUsedObjs();
    Debug.Log($"Number of available enemies: {availableEnemies.Length}");
}
```

    -
- **public data[] GetUsedObjs()**
    - **Description:** Retrieves an array of all currently *used* (occupied) objects from the pool.
    - **Returns:** An array of data structs for all objects currently marked as IsOcupied.
    - **Example:**
    - C#

```
PoolObject particlePool = Central.FindPoolByName("ParticlePool");
if (particlePool != null)
{
   data[] activeParticles = particlePool.GetUsedObjs();
   Debug.Log($"Number of active particles: {activeParticles.Length}");
}
```

    -
- **private data GetUnUsedObj(bool usingThis)**
    - **Description:** (Internal Helper) Retrieves a single unused object. If usingThis is true, it marks the found object as occupied and updates its time.
    - **Returns:** A data struct for an unused object, or a new empty data struct if none found.
- **private data GetForceObj()**
    - **Description:** (Internal Helper) Retrieves the least recently used object from the pool and marks it as occupied. This is called by GetForcedObject when no unused objects are available.
    - **Returns:** A data struct for the least recently used object.
- **public data[] GetForceObjs(int count)**
    - **Description:** Retrieves an array of count objects, prioritizing unused ones first, then filling the remaining count with least recently used (LRU) objects that are currently in use. This is useful for scenarios where you need a specific number of objects and are willing to force reuse.
    - **Parameters:**
        - count: The desired number of objects to retrieve.
    - **Returns:** An array of data structs containing the requested number of objects.
    - **Example:**
    - C#

```
// Get 5 objects, prioritizing unused, then forcing reuse of LRU
data[] fiveObjects = myPool.GetForceObjs(5);
```

    -
- **public data[] GetOverFill()**
    - **Description:** Identifies and returns objects in the pool that exceed the initialCapacity. These are typically objects that were added when the pool grew beyond its initial size. The objects returned are the least recently used among the "overfill."
    - **Returns:** An array of data structs representing the overfilled objects, or null if no overfill exists.
    - **Example:**

    - C#

```
data[] excessObjects = myPool.GetOverFill();
if (excessObjects != null)
{
   Debug.Log($"Found {excessObjects.Length} overfilled objects.");
}
```

    -

3.2.4. Object Injection & Update Methods

These methods are used to add new objects to the pool or to update the status of existing objects.

- **public void InjectNewData(object importObj, bool IsOcupied)**
    - **Description:** Adds a new object to the pool. This method should be called after you instantiate a new object that you want to be managed by this pool. It assigns a unique objid and sets its initial IsOcupied status and time. It also performs a type check to ensure the injected object matches the pool's type.
    - **Parameters:**
        - importObj: The actual object (e.g., GameObject instance, new MyClass()) to be added to the pool.
        - IsOcupied: A boolean indicating whether this new object should initially be marked as in use (true) or available (false).
    - **Example:**
    - C#

```
GameObject newBullet = Instantiate(bulletPrefab);
newBullet.SetActive(false); // Initially inactive
bulletPool.InjectNewData(newBullet, false); // Add to pool as unused
```

    -
- **public void InjectData(object objToUpdate, bool isOccupied)**
    - **Description:** Updates the IsOcupied status and time of an existing object in the pool using its object reference. This is typically used to "return" an object to the pool (by setting isOccupied to false) or mark it as in use (by setting isOccupied to true).
    - **Parameters:**
        - objToUpdate: The object instance whose data in the pool needs to be updated.
        - isOccupied: The new boolean status for the object's occupancy.
    - **Example:**
    - C#

```
// When a bullet hits something and should be returned to the pool
bulletPool.InjectData(myBulletInstance, false); // Mark as unused
myBulletInstance.SetActive(false); // Deactivate the GameObject
```

    -
- **public void InjectData(bool isOccupied, int id)**

- **Description:** Updates the IsOcupied status and time of an existing object in the pool using its unique objid. This is particularly useful when you have stored the objid alongside the object (e.g., in a component on a GameObject) and need to return it to the pool efficiently.
- **Parameters:**
  - isOccupied: The new boolean status for the object's occupancy.
  - id: The unique objid of the object to be updated within this pool.
- **Example:**
- C#

```
// Assuming you have the objid stored on your GameObject
int bulletId = bulletGameObject.GetComponent<PooledItemInfo>().poolInstanceId;
bulletPool.InjectData(false, bulletId); // Mark as unused by ID
bulletGameObject.SetActive(false);
```

-

3.2.5. Pool Management & Cleanup Methods

These methods help maintain the health and size of your pool.

- **public data RemoveObjectFromPool(object objectToRemove)**
  - **Description:** Removes a specific object from the pool's internal management without destroying the actual object. This is useful if an object needs to be permanently removed from pooling (e.g., destroyed for a specific game reason).
  - **Parameters:**
    - objectToRemove: The object instance to be removed from the pool.
  - **Returns:** The data struct of the removed object, or default if not found.
  - **Example:**
  - C#

```
// Remove a specific bullet from the pool's management
data removedBulletData = bulletPool.RemoveObjectFromPool(myBulletInstance);
if (removedBulletData.obj != null)
{
    // Now you might destroy the actual GameObject if it's no longer needed at all
    Destroy((GameObject)removedBulletData.obj);
}
```

  -
- **public data RemoveObjectFromPool(int objectId)**
  - **Description:** Removes a specific object from the pool's internal management using its unique objid.
  - **Parameters:**
    - objectId: The objid of the object to be removed from the pool.
  - **Returns:** The data struct of the removed object, or default if not found.
  - **Example:**
  - C#

```
// Remove an object by its ID
```

data removedObjectData = myPool.RemoveObjectFromPool(123);

- **public void CleanupData()**
  - **Description:** Cleans up the pool by removing any data entries whose obj reference has become null. This can happen if the actual underlying Unity object was destroyed outside of the pool's control.
  - **Example:**
  - C#

myPool.CleanupData(); // Remove any stale entries

- **public void CleanupOverfill()**
  - **Description:** Reduces the size of the pool by removing objects that exceed the initialCapacity. It prioritizes removing the least recently used objects among the "overfill." This method only removes them from the pool's internal lists; it does *not* destroy the actual Unity GameObjects or other objects. You would need to handle the destruction of the actual objects separately if they are UnityEngine.Objects.
  - **Example:**
  - C#

myPool.CleanupOverfill(); // Reduce pool size by removing excess objects

- **public (int Total, int Unused, int InUse) GetPoolStats()**
  - **Description:** Retrieves current statistics for the pool, including the total number of objects, the count of unused objects, and the count of objects currently in use.
  - **Returns:** A tuple containing Total, Unused, and InUse counts. Returns (-405, -405, -405) if the pool is uninitialized or empty.
  - **Example:**
  - C#

var stats = myPool.GetPoolStats();
Debug.Log($"Pool Stats: Total={stats.Total}, Unused={stats.Unused}, InUse={stats.InUse}");

- **public data[] CleanupPool()**
  - **Description:** Clears the entire pool by removing all data entries from its internal lists. It returns an array of all data structs that were removed. This method only clears the pool's management; it does *not* destroy the actual underlying objects. You would need to iterate through the returned data array and destroy any UnityEngine.Objects if they are no longer needed.
  - **Returns:** An array of data structs that were present in the pool before cleanup.
  - **Example:**
  - C#

data[] clearedObjects = myPool.CleanupPool();
foreach (var item in clearedObjects)

```
{
   if (item.obj is GameObject go)
   {
      Destroy(go); // Destroy the actual GameObjects
   }
   // Handle other types of objects if necessary
}
```

- **public void Log(string message)**
  - **Description:** (Internal Helper) Logs a message to the Unity console, prepending it with the pool's name for easy debugging.

3.3. data Struct

The data struct is a simple, lightweight container that holds essential information about each object managed by a PoolObject.

3.3.1. Fields

- **public bool IsOcupied;**
  - **Description:** A boolean flag indicating whether the object is currently in use (true) or available in the pool (false).
- **public int objid;**
  - **Description:** A unique integer identifier assigned to this specific data entry (and thus to the obj it holds) within its PoolObject. This ID is crucial for efficiently updating or retrieving objects from the PoolObject's internal dictionary.
- **public object obj;**
  - **Description:** The actual object being pooled. This can be any UnityEngine.Object (e.g., GameObject, MonoBehaviour instance) or a custom C# class instance. You will need to cast this object to its specific type when retrieving it from the pool.
- **public float time;**
  - **Description:** The Time.time at which the object was last marked as occupied or released. This timestamp is used by the PoolObject's Least Recently Used (LRU) logic for GetForcedObject and GetForceObjs.

3.4. Superiority and Goodness of this Asset

This Object Pooling System offers significant advantages:

- **Ultimate Flexibility:** Unlike many pooling solutions tied to GameObjects, this system allows you to pool *any* UnityEngine.Object (e.g., GameObjects, ScriptableObjects, MonoBehaviour instances) or even your own custom C# class instances. This makes it incredibly versatile for managing various types of reusable data or components, not just visual entities.
- **Centralized Management:** The Central static class provides a single, easy-to-access point for creating and finding all your pools, simplifying your project structure.
- **Robust Object Retrieval:** Beyond simple "get unused," the PoolObject offers a "forced retrieval" (GetForcedObject, GetForceObjs) mechanism based on Least

Recently Used (LRU) logic. This ensures you always get an object when needed, even if all are currently in use, preventing null references and allowing for dynamic resource management.
- **Efficient Internal Structure:** By using both a List and a Dictionary for data management, PoolObject balances iteration speed with fast lookups by ID, optimizing performance for various operations.
- **Clear State Management:** The IsOcupied flag and time stamp within the data struct provide clear tracking of an object's state and usage history, enabling intelligent reuse.
- **Editor Integration (See Section 4):** The included editor tools provide a visual way to monitor and manage your pools at design time, greatly enhancing the debugging and development experience.

---

4. Editor Tools Reference

The Object Pooling System includes custom Unity Editor windows to help you visualize and manage your object pools directly within the Unity Editor. These tools are invaluable for debugging and understanding the state of your pools at runtime.

4.1. PoolManagerEditor Window

This is the main editor window for overseeing all active object pools.

- **Access:** Navigate to Tools/Pool Manager in the Unity Editor menu.
- **Description:** The PoolManagerEditor window displays a list of all pools currently registered with the Central static class. For each pool, it shows its name, ID, type, and real-time statistics (Total, Unused, In Use).
- **Features:**
    - **Pool List:** Displays all active pools.
    - **Stats:** Shows live statistics for each pool.
    - **Select/Unselect Button:** Allows you to select a specific pool to reveal more detailed actions.
    - **"Clean Up Data" Button:** Calls pool.CleanupData() for the specific pool, removing any data entries whose obj reference is null.
    - **Detailed Actions (when selected):**
        - **"View the Objects" Button:** Opens the UnusedObjectDropdown window (see Section 4.2) to inspect the individual used and unused objects within the selected pool.
        - **"Clean Up Overfill" Button:** Calls pool.CleanupOverfill() to remove objects exceeding the initialCapacity from the pool's internal management. Note: For GameObjects, this tool also *destroys* the actual GameObject instances that are identified as overfill.
        - **"Clean Up Pool" Button:** Calls pool.CleanupPool() to clear all objects from the pool's internal management. Note: For GameObjects, this tool also *destroys* the actual GameObject instances that were in the pool.
        - **"Expand By" Field & "Expand Pool" Button:** Allows you to manually expand a pool by a specified number of *newly created*

GameObjects. This is primarily for testing and demonstration purposes within the editor.

4.2. UnusedObjectDropdown Window

This window provides a detailed view of the individual objects within a selected pool.

- **Access:** Opened automatically by clicking the "View the Objects" button in the PoolManagerEditor for a selected pool.
- **Description:** This window lists all used and unused objects within the chosen pool, displaying their last used time (time) and allowing you to inspect the actual UnityEngine.Object reference in the Inspector.
- **Features:**
  - **Foldouts:** Separate sections for "Used Objects" and "Unused Objects" that can be expanded or collapsed.
  - **Object Fields:** For UnityEngine.Objects, an EditorGUILayout.ObjectField is used, allowing you to click on the object to highlight it in the Hierarchy or Project window, and view its properties in the Inspector.
  - **Last Used Time:** Displays the time value for each object, indicating when it was last accessed.

---

5. Example Implementations

The following scripts are provided as practical examples of how to integrate and use the core Object Pooling System (Central, PoolObject, data) within your Unity projects. You can use these as-is, modify them, or create your own custom pooling logic using the core API.

5.1. GameObjectPooler.cs (Example MonoBehaviour)

This script provides a convenient, drag-and-drop MonoBehaviour for pooling GameObjects. It encapsulates the interaction with the Central and PoolObject classes for common GameObject pooling scenarios.

- **Purpose:** To simplify the process of setting up and using a GameObject pool without writing custom pooling logic for each type of object.
- **How to Use:**
  - Create an empty GameObject in your scene (e.g., "BulletPoolManager").
  - Add the GameObjectPooler component to it.
  - Assign a poolName (e.g., "PlayerBullets").
  - Drag your prefab (the GameObject you want to pool) into the Prefab slot.
  - Set initialCapacity (e.g., 20).
  - Optionally assign a GOParent (Transform) under which pooled GameObjects will be instantiated in the Hierarchy.
  - Choose initializationMethod (OnStart for automatic initialization or OnDemand for manual control).
  - Choose retrievalMethod (GetUnused for strict unused retrieval, or GetForced for LRU-based forced reuse).
- **Key Features:**
  - **Initialization:** Can initialize the pool on Start (as a Coroutine to prevent frame drops for large pools) or manually OnDemand.

    - **GetObject():** Retrieves a GameObject from the pool based on the chosen retrievalMethod. It also ensures the PooledItemInfo component is added and its poolInstanceId is set.
    - **ReturnObject(GameObject obj, int id):** Returns a GameObject to the pool using its objid (obtained from PooledItemInfo), deactivating it.
    - **GetPoolStats():** Provides current statistics for this specific pool.
    - **ExpandPool(int additionalCount):** Allows runtime expansion of the pool by instantiating new prefabs and injecting them.
    - **CleanupOverfill():** Cleans up objects exceeding initialCapacity from this pool.
    - **CleanupPool():** Clears all objects from this pool.
    - **Automatic Central Integration:** Automatically registers and unregisters its PoolObject with Central on Awake and OnDestroy.

5.2. PooledItemInfo.cs (Helper Component)

This is a simple MonoBehaviour that acts as a tag for pooled GameObjects.

- **Purpose:** To store the unique objid of a pooled GameObject (assigned by GameObjectPooler) directly on the GameObject itself. This allows the GameObject to "know" its own ID within the pool, making it easy to return it to the correct pool instance.
- **How to Use:** It is automatically added by GameObjectPooler when an object is retrieved. You typically don't need to add this manually.
- **Field:**
    - public int poolInstanceId;: Stores the objid from the data struct.

5.3. ForceApplier.cs (Example Usage)

This script demonstrates how to use the GameObjectPooler to spawn objects and apply physics forces to them.

- **Purpose:** To show a common use case where objects are frequently spawned and require physics interactions.
- **How to Use:**
    1. Ensure you have a GameObjectPooler set up for the GameObject you want to spawn.
    2. Create an empty GameObject (e.g., "Spawner").
    3. Add the ForceApplier component to it.
    4. Drag the GameObjectPooler instance from your scene into the Pooler slot.
    5. Set spawnPosition (an empty GameObject marking where to spawn).
    6. Adjust spawnInterval, upwardForce, sideForceRange, and torqueRange as desired.
- **Functionality:** In Start, it begins a coroutine to repeatedly GetObject() from the assigned GameObjectPooler, positions the object, and applies random forces and torque to its Rigidbody.

5.4. ReturnToPoolOnLowY.cs (Example Usage)

This script demonstrates how a pooled GameObject can automatically return itself to its pool based on a condition (in this case, its Y-position falling below zero).

- **Purpose:** To show how pooled objects can self-manage their lifecycle and return to the pool when no longer needed, rather than being destroyed.
- **How to Use:**
    - Add this component to the prefab that is being pooled by a GameObjectPooler.
    - Set the poolName property to match the poolName of the GameObjectPooler instance managing this prefab.
- **Functionality:**
    - In Start, it finds the correct GameObjectPooler instance in the scene based on the provided poolName.
    - In Update, it continuously checks if the GameObject's Y-position falls below 0f.
    - If the condition is met, it retrieves the poolInstanceId from its PooledItemInfo component and calls assignedPooler.ReturnObject() to return itself to the pool, which also deactivates the GameObject.
    - Includes fallback Destroy(gameObject) if the pooler or PooledItemInfo cannot be found, preventing unmanaged objects.

---

6. Best Practices

- **Define idealsize Carefully:** Set the initialCapacity for your pools based on the maximum number of objects of a specific type you expect to have active simultaneously. This minimizes runtime allocations and reallocations.
- **Pool Frequently Instantiated/Destroyed Objects:** Prioritize pooling objects that are created and destroyed often (e.g., projectiles, enemies, visual effects, UI elements).
- **Activate/Deactivate Pooled GameObjects:** When retrieving a GameObject from a pool, remember to activate it (gameObject.SetActive(true)). When returning it, deactivate it (gameObject.SetActive(false)).
- **Reset Object State:** When an object is retrieved from a pool, ensure you reset its state (position, rotation, velocity, health, etc.) to a default or desired value before use. The GameObjectPooler example handles activation/deactivation, but you are responsible for resetting other properties.
- **Use objid for Efficient Returns:** When returning GameObjects, storing the objid (e.g., using PooledItemInfo) and using pool.InjectData(false, id) is generally more efficient than pool.InjectData(object, false).
- **Manage Pool Cleanup on Scene Transitions:** Consider calling Central.NewPoolObjectsList(false) or Central.NewPoolObjectsList(true) when transitioning between scenes to manage memory effectively and prevent pools from persisting unnecessarily.
- **Monitor Pools with Editor Tools:** Regularly use the PoolManagerEditor to inspect the health and statistics of your pools, especially during development and testing.
- **Handle Overfill:** Use CleanupOverfill() on your PoolObject instances (or via the editor tool) to reclaim memory from objects that were added when the pool grew beyond its initialCapacity. Remember to destroy the actual UnityEngine.Objects if they are no longer needed after cleanup.
- **Error Handling:** While the system provides warnings, consider adding more robust error handling (e.g., custom exceptions or more explicit fallback behavior) in your

production code when FindPool... methods return null or GetObject fails to provide an object.

---

7. Known Issues / Beta Limitations

- **No Automatic Object Pre-instantiation:** While initialCapacity is passed to PoolObject, the PoolObject itself does not automatically pre-instantiate objects. This is typically handled by the consumer (e.g., GameObjectPooler's InitializePoolAsync or your custom initialization logic) when InjectNewData is called.
- **Manual Object Destruction on Cleanup:** PoolObject.CleanupOverfill() and PoolObject.CleanupPool() only remove objects from the pool's internal management. For UnityEngine.Objects (like GameObjects), you must manually destroy the actual objects after these cleanup operations if they are no longer needed. The PoolManagerEditor and GameObjectPooler examples demonstrate this.
- **Limited Debugging Information:** While GetPoolStats() provides basic counts, more detailed statistics (e.g., hit/miss ratio, average retrieval time) are not yet available.

---

8. Future Development (Planned for Full Release)

- **Automated Object Creation/Destruction Callbacks:** Allowing users to specify createFunc, actionOnGet, actionOnRelease, and actionOnDestroy delegates within the PoolObject for more automated and controlled object lifecycle management.
- **Advanced Pool Resizing Strategies:** Options for automatically resizing pools based on usage patterns (e.g., shrinking when idle, growing dynamically).
- **Improved Debugging and Monitoring:** More detailed runtime statistics, visualizers, and logging options.
- **Performance Optimizations:** Continuous profiling and optimization of internal data structures and algorithms.
- **Batch Operations:** Methods for getting or returning multiple objects at once for further performance gains.

  ***note these are in consideration and on creation but can not guarantee 100% certainty

---

Your feedback during this beta phase is highly valuable! Please report any issues or suggestions to help us improve this Object Pooling System.