GameObject Pooler

User Guide

Freakshow Studio

Contents

1 Introduction

GameObject Pooler is a scripting plugin for Unity that implements a flexible and easy to use object pool.

Object pooling is essential for performance as it avoids the costly *Instantiate* and *Destroy* functions and the associated garbage that is generated and causes the garbage collector to kick in.

The Pool is simple to use, just replace your calls to *Instantiate* with *Pool.Borrow* and your calls to *Destroy* with *Pool.Return*.

Set up the items in your pool either in the editor or through code at runtime. This way you can have different items in your pool for example for each level.

Includes the possibility for logging the usage of each pool item, so you can fine-tune the number of items in your pool.

2 Quickstart

The *GameObject Pooler* lives in its own namespace, so in any script where you intend to use it, you first have to include the namespace. In C# you do this with *using FreakLib.Pooler* and in JavaScript you use *import FreakLib.Pooler*.

To get started right away, you can simply replace all your calls to *Instantiate* with *Pool.Borrow*, and your calls to *Destroy* with *Pool.Return*. The Pool will then automatically create these objects as pool objects for you without any further setup.

It is however recommended to add the Pool to the scene as this gives you more control over its options. Add it from the menu *GameObject* ➔ *Pool*. You can then set the configuration options as you please, and add pool items using the *'Add PoolItem'* button.

Once you have set up the pool and the items you wish to use, simply use *Pool.Borrow* to get an object from the pool, and *Pool.Return* to return an object back to the pool.

3 Setting up the Pool

Setting up the Pool is optional, but recommended. To use the Pool without setting it up, se the chapter Using the Pool without setup. Without setting it up beforehand you will loose the ability to configure it and pre-instantiate game objects, unless you choose to do this at runtime. See the chapter Managing the Pool at runtime for more information on how to set up pool objects at runtime.

3.1 Adding the Pool to a scene

The Pool is a singleton class, which means there should ever only be one Pool in a scene at a given time.

To add a Pool to the current scene, use the menu option *GameObject* ➔ *Pool*. This option will be disabled if there is already a Pool detected in the scene. It is also possible to add the Pool.cs script to an existing game object.

3.2 Configuring the Pool

The Pool is configured through its inspector. Select the Pool in the hierarchy to display it. The configuration options for the Pool are described below.

3.2.1 Instantiate On Awake

This option determines if the Pool should pre-instantiate all the required objects when the scene starts.

If this option is turned off, you will need to call the function Pool.Instantiate yourself from code.

You can always check if the Pool has been instantiated through the property Pool.IsInstantiated.

3.2.2 Async Instantiate

If this option is enabled, the Pool will use a co-routine to instantiate one PoolItem at a time.

Instantiating all the copies needed for the Pool can take a while if there are many objects to create, and this can cause your application to appear to hang while they are instantiated. Enabling this option will help with this, and lets the application render a few

frames while instantiating. It will however take slightly longer to instantiate the Pool.

It is important that you don't start using the Pool before all the items have been instantiated if you are using this option. Check the property Pool.IsInstantiated to check if it has finished.

3.2.3 Create If Not Exists

When you try to borrow an object from the Pool that has not been set up as a PoolItem, this will normally give you an error. However if you enable this option, the Pool will instead just create a new PoolItem for the given object.

This allows the Pool to be used with a minimum of setup, and this option can be useful especially when prototyping or in the early stages of development.

New PoolItems are however created with the default settings, and for the best performance you will usually want to set up the item yourself.

3.2.4 Warn On Grow

Growing the Pool for a given item can be an expensive operation. By enabling this option, the Pool will print a warning in the console whenever it needs to grow. This way you can identify problem objects that are used more than you have anticipated.

3.2.5 Use Interface Callback

The Pool comes with an interface, IPoolObject, which you can implement in your own classes that you use on pooled objects. See the chapter Pool item interface for more information about this.

Because finding scripts with this interface and calling its methods can be an expensive operation, it is possible to turn this functionality off here.

3.2.6 Log Usage

By enabling this option, the Pool will record the maximum simultaneous active objects for all the pool items. This way, you can play through your game and afterwards look at the log to determine how many copies of each it would make sense to pre-instantiate.

The logging is only done when your application is run from the Unity editor.

3.2.7 Log File

This is the name of the log-file written when the Log Usage option is enabled.

The output file is a comma-separated-values file (.csv), and the path is relative to your project path.

Note that the file will be overwritten without warning each time play-mode is exited in Unity.

3.2.8 List of Pool Items

If any pool items have been set up, you will see a list of them in the inspector. The list displays their name, the associated object, the initial number of copies that should be used, and a delete button.

3.2.9 Add Pool Item

Use the *'Add PoolItem'* button to add more items to the Pool.

3.3 Setting up Pool Items

To create a new pool item, press the *'Add PoolItem'* button on the Pool, or drag the PoolObject.cs script to a game object. Pool items need to be a child of the Pool object in the hierarchy.

The settings for a pool item are documented below.

3.3.1 Object

This is the original game object or prefab that you want to pool.

It is possible to use scene objects, but special consideration needs to be given if your pool objects use Unity's OnEnable and OnDestroy methods.

3.3.2 Deactivate Before Instantiate

This option determines if the original game object or prefab should be de-activated when copies of it are made for the pool.

If this is turned off, the OnEnable method will be called on the copies when they are instantiated for the pool, if the original object is enabled.

When the original object is a scene object and this option is enabled, the OnDisable and OnEnable methods on the original will be called when copies are made, but OnEnable will not be called on the copies.

The normal scenario is using prefabs as the originals, and leaving this option enabled. This way OnEnable will not be called when copies are instantiated.

3.3.3 Initial Capacity

The initial capacity is the number of copies of this object that should be created when the Pool instantiates.

Increasing this number will cause instantiation to take longer, and will increase memory consumption. A too low number on the other hand, will cause the Pool to have to grow at runtime.

The number set here should be the maximum number of this object that you expect to have active at any one time.

3.3.4 Grow Strategy

The Grow Strategy determines what the Pool will do when there are no more copies, but more are requested.

Setting this to *DontResize* will cause the Pool to not grow when exhausted, but instead throw a *PoolException*.

With the *Exponential* Grow Strategy, the number of items will double. So for example if it originally had 2, it will now increase to 4. It also means it will increase from for example 500 to 1000, which means a lot of new objects need to be created, so care should be taken with this option.

The *Linear* Grow Strategy will resize the Pool by a constant number; see the Linear Grow Amount option.

3.3.5 Shrink

This option determines if the Pool should shrink when there are enough free objects.

It follows the same strategy as set in the Grow Strategy, if this is set to *DontResize* it will never shrink; if it is set to *Exponential* it will shrink to half the size when half of the copies are unused; if it is set to *Linear*, it will shrink by the number set in Linear Grow Amount when there are as many free objects.

Shrinking will conserve memory, but like growing is an expensive operation so care should be taken. You might also end up in a position where the Pool is constantly shrinking and growing because you are operating near the capacity.

3.3.6 Linear Grow Amount

This is the number of copies that will be added (or removed if using Shrink) when using the *Linear* Grow Strategy.

3.4 Using the Pool without setup

It is also possible to use the Pool without setting up any pool items as long as the Create If Not Exists option is enabled, or even without adding a Pool to the scene at all.

If no Pool is added to the scene, one will be created when you first attempt to use it.

All the objects added in this way will be set up with the standard parameters, and a capacity of only 1. It will grow as needed, using an *Exponential* Grow Strategy.

Using the Pool this way will be acceptable for some games, and is useful during prototyping and early development, but pre-instantiating will usually result in better performance.

4 Using the Pool

The Pool lives in its own namespace, so before you start using it, it is important that you include this in your scripts. In C# you do this with the statement *using FreakLib.Pooling* and if you are using JavaScript, you use the statement *import FreakLib.Pooling* at the top of your script file.

For using the Pool, the two main functions are *Borrow* and *Return*. These look and act in the same way as Unity's built in *Instantiate* and *Destroy* methods, so to start using the Pool you can simply replace any call in your code to *Instantiate* with *Pool.Borrow*, and *Destroy* with *Pool.Return*. Just make sure you don't call *Destroy* on an object that you have borrowed with *Pool.Borrow*.

4.1 Getting objects from the Pool

To get an object from the Pool, use the *Pool.Borrow* function, and pass in the original prefab or scene item, and it will return a copy of this object from the Pool.

This function comes in three flavors, one with only the original object as a parameter, one with an optional position, and a third with yet another optional rotation.

These functions mirror Unity's built in *Instantiate* method, and can for the most part replace this directly. The differences are that *Borrow* returns a *GameObject* instead of an *Object*, and the first parameter also needs to be a *GameObject*. This means that *Borrow* will not work for components.

It is important that you don't call *Destroy* on game objects borrowed from the Pool, but instead use the function *Pool.Return*.

4.2 Returning objects to the Pool

Returning objects to the Pool is done with the function *Pool.Return*. This mirrors Unity's *Destroy* method, with the exception that *Pool.Return* only accepts a *GameObject*.

There are two versions of this function, one that accepts only the *GameObject*, and one that has an optional time before it should be deactivated.

The *GameObject* passed to *Pool.Return* should be the copy that is returned from a call to *Pool.Borrow*, and not the original.

5 Managing the Pool at runtime

For some applications it is enough to set the Pool up once and just use it, but more complicated applications might need to modify the Pool at runtime. For example, different levels could have different requirements, and to conserve memory you'd want to set up the Pool specifically for each level.

Refer to the included API Reference for more information on the provided functions.

5.1 Getting PoolItem references from the Pool

To get an array of all the pool items currently in the Pool, use the function *Pool.GetPoolItems*.

If you want a specific PoolItem, use the *Pool.GetPoolItem* function, and pass in the original object. The PoolItem object will be returned in the out parameter, and will be null if the original object wasn't found in the Pool.

You can also check if an object exists in the Pool with the function *Pool.ExistsInPool*.

5.2 Adding objects to the Pool

Adding new objects to the Pool is done with the function *Pool.AddPoolItem*. You can either pass in the original object and an initial capacity, or a PoolItem object. Passing in a PoolItem object allows you to configure it the way you'd like beforehand.

When *Pool.AddPoolItem* is called, the copies for this pool item is also instantiated. As this can take some time, you should do this at non-critical times in your application.

5.3 Removing objects from the Pool

To clear the Pool completely, use the *Pool.ClearPool* function. This will remove everything from the Pool.

If you wish to remove just one item from the Pool, use the *Pool.RemovePoolItem* function. This will accept either the original object you wish to remove, or the corresponding PoolItem object.

5.4 Using the PoolItem class directly

While not recommended, if you know what you are doing you might in some cases want to bypass the Pool class, and use the PoolItem class directly.

The PoolItem class has all the needed functionality to instantiate copies, as well as borrowing and returning these. However if you choose to use this directly, you will need to manage the borrowed objects yourself, and return them to the correct PoolItem.

Refer to the implementation in Pool.cs to see how this can be done.

6 Pool item interface

To use the pool item interface, this option needs to be enabled on the Pool, see Use Interface Callback.

Once enabled, implement the IPoolItem interface in your own scripts. These scripts need to be placed on the original pool item game object or prefab, not on a child.

The methods *OnBorrow* and *OnReturn* that you have implemented will now be called on any copies of this object whenever they are borrowed or returned from the pool.

7 One-Shot scripts

Also included are two one-shot scripts, one for audio clips and one for particle systems. These can be used to have an audio clip or particle system play only once and then return itself to the pool.

To use them, add them to a game object with an audio source or particle system component, and then create a prefab out of these, and add the prefab to the pool.

Then, to play the clip or particle, just borrow them from the pool. Once they have stopped playing they will return themselves.

The audio source and particle system should not be set to loop; if they are then the object will continue playing and not return itself.