PoolKit - For Unity.

www.unitygamesdevelopment.co.uk

Created By Melli Georgiou

The ultimate system for professional and modern object pooling, spawning and despawning.

Table of Contents

Version History

v3.0.2
- Bugfix: Fixed issue when despawning objects with 2D physics and the countdown option.

v3.0.1
- Bugfix: Fixed instance recycling bug.

v3.0
- Despawner: Conditional Chain spawning for physics events.
- Despawner: Foldout for each individual chain spawning entries.

v2.0.2
- Updated for Unity 2019.3

v2.0.1
- Despawner: Hotfix for Audiosources.
- Minimum Unity version is now 2017.4

v2.0
- Spawner: Randomized Offsets feature to apply random offsets to spawn points.
- Spawner: Allow a random range of instances per spawn cycle.
- Spawner: Allow spawners to play until a specific number of instances are spawned.
- Spawner: Update prefabs, positions, offsets, rotations and scale by instance or spawn cycle.
- Despawner: New "After Raycast Event" and "After Raycast2D Event" modes.
- Despawner: Chain spawning now supports a random range of instances.

v1.0.4
- Updated how parenting works to avoid warnings with RectTransforms.

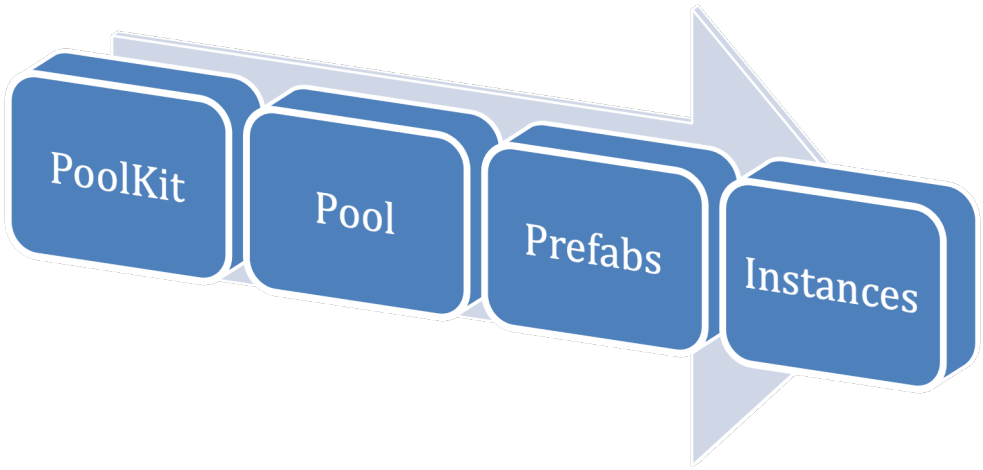
v1.0.3
- Added new SetSpawnPointPosition API method to spawners.

v1.0.2
- Bugfix for caching IPoolKitListeners on newer versions of Unity.

v1.0.1
- Prefabs can now be minimized and expanded in the Pool Editor.

v1.0
- First Commercial version of plugin.

An Introduction To PoolKit



What Is PoolKit?

PoolKit is the ultimate Unity plugin for professional and modern object pooling, spawning and despawning. It is the type of toolkit that any game project would benefit from!

PoolKit allows you to build any number of local and global GameObject pools, spawners and despawners. A Pool is essentially a manager that handles an array of "instances" (or clones) of a group of prefabs. These instances are "recycled" instead of constantly instantiating and destroying objects, which can significantly improve performance.

Why Choose PoolKit Over Other Solutions?

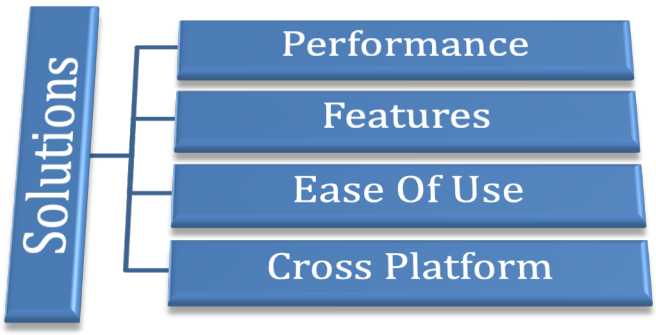
At the time of release, PoolKit has powerful and **unique** features that set it apart from traditional pooling systems such as:

- Easy to use visual editors for every part of the system!
- Multiple data structures, known in PoolKit terminology as "Pool Types". PoolKit can automatically use different data structures under the hood depending on if you are using a fixed or dynamic sized pool. This can result in significant performance increases unavailable in other pooling systems and all without any difference in the way you use the API!
- Pools can be local or global and created at any time. Global pools can also be automatically created when the game starts without any scripting!
- "Chain spawning" is a feature of the easy to use Despawner components. This allows instances to chain-spawn a list of new objects when they despawn, all with their own customizable settings!
- "Pool Protection" is an optional feature that protects the pool from you accidentally deleting an instance (it happens!). It allows instances to be recreated in real-time and avoid potential runtime errors!

- The PoolKit Spawner offers a truely incredible amount of functionality, making it great for setting up weapons, special effects and so much more!
- An all-purpose instance Despawner component as well as "Automatic Despawning" features available at the Pool's Prefab level.
- The "IPoolKitListener" interface is available for scripts to run actions when they are spawned and despawned using ultra-fast and unique caching!

PoolKit is designed to be scalable for all users. There are custom visual editors for every part of the system (as well as many built-in PlayMaker actions) that make it easier for beginners. At the same time, the sheer customizable nature of the system, the ability to make local and global pools anywhere, as well as the powerful API makes it a great choice for advanced users too.

The API allows advanced users to get their hands dirty by allowing for pool creation and modification, overriding instance creation and subscribing to a magnitude of Events to name but a few scenarios. PoolKit has you covered!



What Can PoolKit Do For Your Project?

- **BOOST PERFORMANCE**
  When you instantiate and destroy objects in your game, you create "garbage" in memory that needs to be cleaned up. This leads to frame rate hiccups that diminish the experience of your games. If you think of a typical game situation of a weapon instantiating a wave of bullets, this will invariably performance "hiccups". PoolKit uses an industry proven way of handling this problem using a unique take on "Pooling". PoolKit is so optimized that it is possible to get zero memory allocations during runtime (except for the first frame where the pool is created of course!).

- **FEATURES**
  PoolKit offers powerful functionality for setting up your pools and configuring how objects will be spawned and despawned (in 2D and 3D!). Pools can be local to a scene, persistant throughout the game or use a custom global configuration that allows you to load and unload pools at will. Unlike other systems, PoolKit's scalability means you won't waste

vast amounts of performance on features you don't need. Most features can be toggled allowing you to boost performance even more by turning off some of the bells and whistles. Spawning is easy with the PoolKit Spawner and at the time of release, we think it is unrivalled on the Asset Store with vastly superior functionality. There are also a variety of ways to despawn objects using either the "Automatic Despawn" feature on the Pool itself or using the all-purpose Despawner component on a specific prefab. Lastly, PoolKit also includes real-time Pool statistics, giving you visual insights to pool usage in the Editor. PoolKit is a truly modern powerhouse of object pooling!

- **EASE OF USE**
  As part of PoolKit's scalable design philosophy, powerful visual editors give tremendous power to advanced users but also help new developers at the same time. As well as the built-in PlayMaker actions, PoolKit has an optional "Helpful Notes" system that can be toggled in it's own Unity Preferences. These notes will explain exactly what is happening as you make different selections to pools, spawners and despawners. It even offers tips and dynamic code snippets if you're using the API! You are told if a certain configuration could cause issues or if a certain feature could affect performance if used excessively. Pooling is extremely complicated under the hood and PoolKit goes the extra mile not only to make it easier to understand but to directly help you do what you need to do!

- **CROSS PLATFORM & OPTIMIZED**
  You may have noticed that PoolKit's demos are available using WebGL builds. We've also tested and optimized for desktops and mobile!

- **GREAT SUPPORT**
  As well as the thorough documentation, online forum and video tutorials, friendly email support is always at hand if you need help! ☺

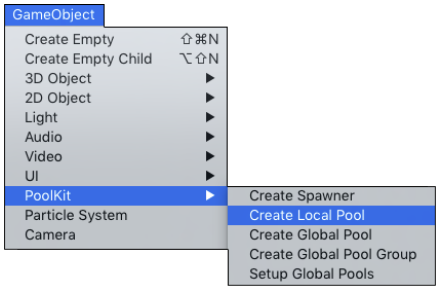
Quick Setup

After installing the PoolKit package, here's how to quickly create PoolKit elements from the menu (you can also do this by right-clicking in the Hierarchy).

How To Create A Local Pool

To create a new local pool from the menu:
GameObject > PoolKit > Create Local Pool

A Local Pool will be created and selected in the Hierarchy pane.



How To Create A Global Pool

To create a new global pool from the menu:
GameObject > PoolKit > Create Global Pool

A Global Pool will be created and selected in the Hierarchy pane. You should save this GameObject as a prefab so you can easily use it with the "Setup Global Pools" option.

How To Create A Global Pool Group (Advanced)

To create a new global pool group from the menu:
GameObject > PoolKit > Create Global Pool Group

A Global Pool within its own PoolKitSetup group will be created and selected in the Hierarchy pane. This allows you to create custom global pools that you can manage manually.

How To Setup Your Global Pools

To setup your Global Pools from the menu:
GameObject > PoolKit > Setup Global Pools

You will be taken to the PoolKit Global Pools asset to setup which global pools will be created automatically when the game starts.

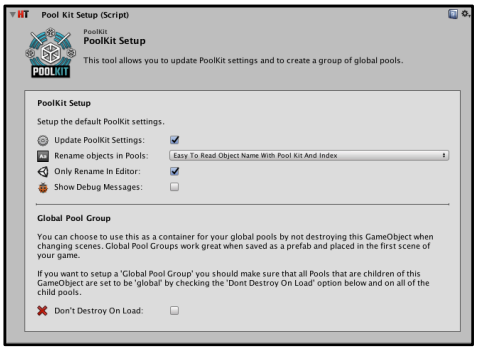
How To Create A Spawner

To create a new Pool from the menu:
GameObject > PoolKit > Create Spawner

A Spawner will be created and selected in the Hierarchy pane.

An Overview Of PoolKit

About The PoolKit Setup Script



The “PoolKitSetup” script is an easy way to setup global PoolKit options without using the API.

You can setup how instances are named, whether debug messages should be shown and you can also choose to use it as the parent object of a ‘Global Pool Group’.

About The Pool Script

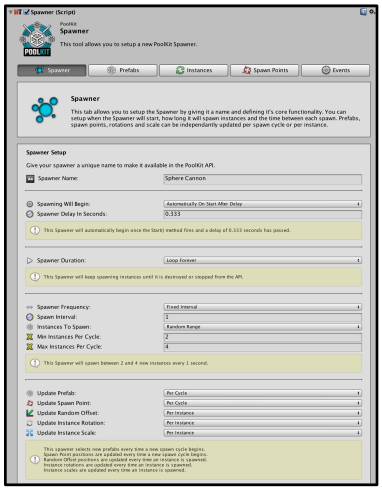
The “Pool” component allows you to setup a new PoolKit Pool.

You can setup how the pool will work by setting features like the Pool Type and Pool Protection.

The second tab allows you to setup the prefabs that will be used in the Pool, along with their prefab-specific settings and features.

Finally, you can debug the pool at runtime by using the Statistics tab.

About The Spawner Script



The Spawner component allows you to spawn any number of prefabs using a feature packed, easy to use system that is somewhat similar in concept to Unity’s Particle System.

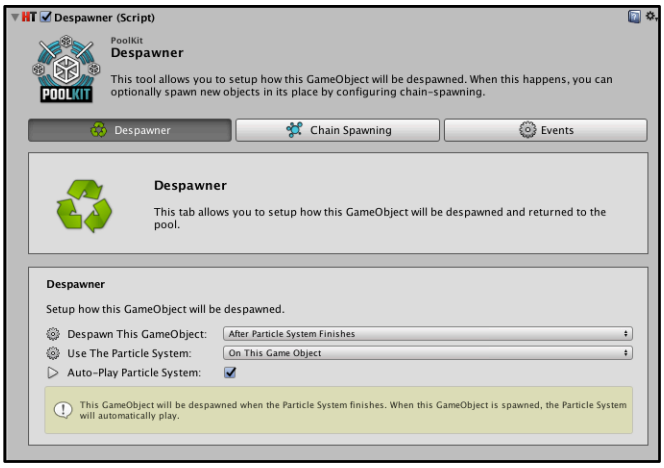
Firstly, you setup core settings such as the start event, frequency, amount, duration, etc. You then configure which prefabs to use and how they will be selected. When instances are spawned, you can setup how they are positioned, scaled, rotated and parented. Finally, events are available to connect the Spawner to other scripts.

About The Despawner Script

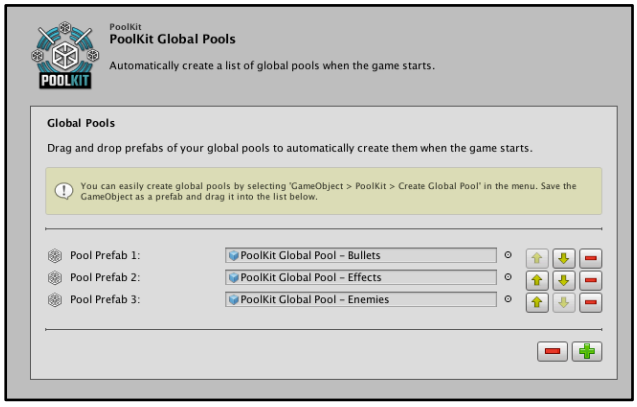
The “Despawner” script is a tool to easily despawn objects based on a variety of situations such as after a countdown, particle system, audio, or physics event.

Chain spawning can also be setup to spawn a list of new objects when an instance is despawned.

Finally, Events are available to connect to other scripts.



About The PoolKit Global Pools Asset



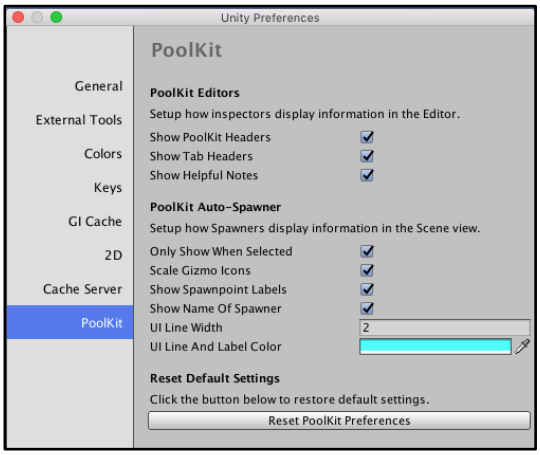
The “PoolKit Global Pools” Asset allows you to setup which global pools will be created automatically when the game starts.

You simply drag and drop your global pool prefabs into the list and you’re good to go!

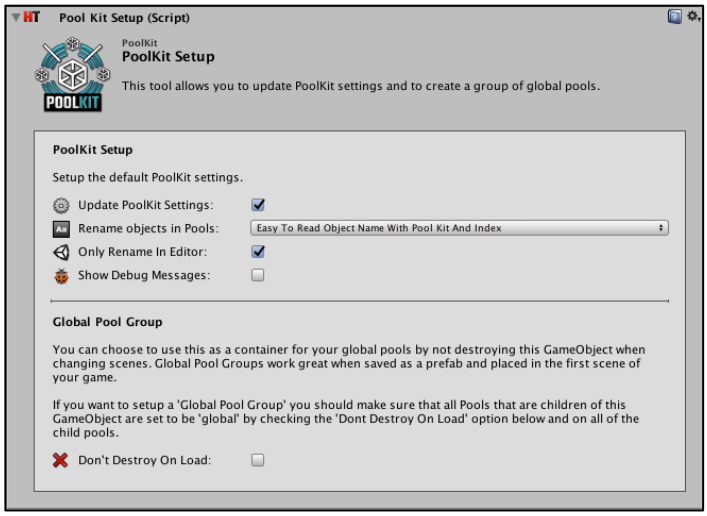
About The PoolKit Preferences

PoolKit has its own section in the Unity Preferences window.

You can setup how much info is displayed to you in the Unity Visual Editors, how the PoolKit Spawners are displayed in the scene as well as a button to revert to default settings.



Setting Up The PoolKit Setup Component



The "PoolKit Setup" script is an easy way to setup some global PoolKit options without using the API. It also allows for a great way to setup a group of global Pools (aka "Global Pool Groups").

Update PoolKit Settings

This option will determine if the actions in this section will update or not. The Global Pool Group section is treated separately.

Rename Objects In Pools

The first drop down menu allows you to setup how to rename the instances that are created in Pools. Note that renaming can be performance hungry when using rapidly growing Dynamic Lists. The following options are available:

Easy To Read Object Name With PoolKit And Index
This renames instances to use a template like this:
My_Prefab_PoolKit_0001

Easy To Read Object Name And Index
This renames instances to use a template like this:
My_Prefab_0001

Object Name With PoolKit And Index
This renames instances to use a template like this:
My Prefab(Clone)_PoolKit_0001

Object Name With Index
This renames instances to use a template like this:
My Prefab(Clone)_0001

No Renaming
This uses the Unity default name, like this:
My Prefab(Clone)

Only Rename In Editor

This is a great option that allows for more readable names in the Editor but turns it off in builds which keeps performance at its best. If you need to match the names that you see in the Editor, you can disable this option.

Show Debug Messages

This option will show PoolKit debug messages in the console. Please note that writing to the console actually causes memory allocations so this should be disabled in final builds.

Global Pool Group / Don't Destroy On Load

This option will stop this GameObject and any of its children from being destroyed when changing scenes. This makes it a great way to setup a global group of Pools, known in PoolKit terminology as a 'Global Pool Group'.

When creating a Global Pool from the menu, it will automatically be setup as a child of the "PoolKit Setup" object.

The most common setup for Global Pools is to mark all of the child pools as global (by checking the 'Don't Destroy On Load' checkbox in the Pool tab). From there, you can save it as a prefab and place it on the first scene of your game (which should be setup as a type of splash screen). This will make sure that your global pools are available as soon as your game starts!

There are many ways to configure global pools. Check out the PoolKit Workflow chapter for more information on setting up Global Pools.

Setting Up The Pool Component

The Pool script allows you to setup a new PoolKit Pool. It consists of the Pool, Prefabs and Statistics Tabs.

The Pool Tab

The Pool tab allows you to name your Pool system and define how it works. These are options that are shared across all of the prefabs that make up the pool.

Pool Name

The pool name is used to access the pool via the API. It is best to use something descriptive and short such as “SFX”, “Bullets”, “Enemies”, etc.

Pool Type

The data structure used for storing and managing instances in the Pool:

Automatic
The Pool will automatically choose the fastest Pool Type based on your settings.

Fixed Array
The fastest type of Pool. The downside is no new instances can be added at runtime and lazy preloading is not available.

Dynamic List
Very fast but not the fastest. The benefits of this type of pool are that new instances can be added over time and it can use the lazy preloading feature.

Enable Pool Protection

Pool protection can rebuild your pool in real-time if you accidentally destroy an instance instead of despawning it. There is a small performance cost for enabling this.

Don't Destroy On Load / Enable Global Pool

Stops the Pool from being destroyed when changing scenes, essentially making it a global Pool. To use Global Pools, the GameObject must either have no parent object or be a child of a properly configured 'Global Pool Group' (See PoolKit Setup). For more information on setting up local and global pools, check out the PoolKit Workflow chapter.

Enable Delegates & Events

Enabling this option will tell the pool to check if any other scripts have subscribed to its events via the API and trigger them. Unless you actually need to use this feature, leave it unchecked (you can also override this directly via the API). Here is an example of how to subscribe to Spawn and Despawn events!

```
// Make sure to add: using HellTap.PoolKit;
public string poolName = "MyPool";
Pool findPool;

void OnEnable(){

    // Find the pool (if we haven't already)
    if(findPool==null){ findPool = PoolKit.Find( poolName ); }

    // Subscribe to the events
    if(findPool!=null){
        findPool.onPoolSpawn += onPoolSpawn;
        findPool.onPoolDespawn += onPoolDespawn;
    }
}

void OnDisable(){

    // Unsubscribe from the pool if it was found
    if(findPool!=null){
        findPool.onPoolSpawn -= onPoolSpawn;
        findPool.onPoolDespawn -= onPoolDespawn;
    }
}

// Event called when the pool spawns an instance
void onPoolSpawn( Transform instance, Pool pool ) {
    Debug.Log("The Pool " + pool.poolName + " just spawned: " + instance.name );
}

// Event called when the pool despawns an instance
void onPoolDespawn( Transform instance, Pool pool ) {
    Debug.Log("The Pool " + pool.poolName + " just despawned: " + instance.name );
}
```

The Prefabs Tab (Overview)

The Prefabs tab allows you to setup the prefabs that will be managed by the Pool. Even though each of these prefabs will share the same Pool system, they work independently with their own customizable options and features.

Each Prefab stores its data internally in a class called a "Pool Item". These Pool Items are presented in 4 tabs named "Prefab", "Instances", "Features" and "Advanced".

The Prefabs Tab (Prefab)

Prefab To Pool

This is where you should drag and drop a prefab from the Project pane to be used in the pool.

Pool Size Options

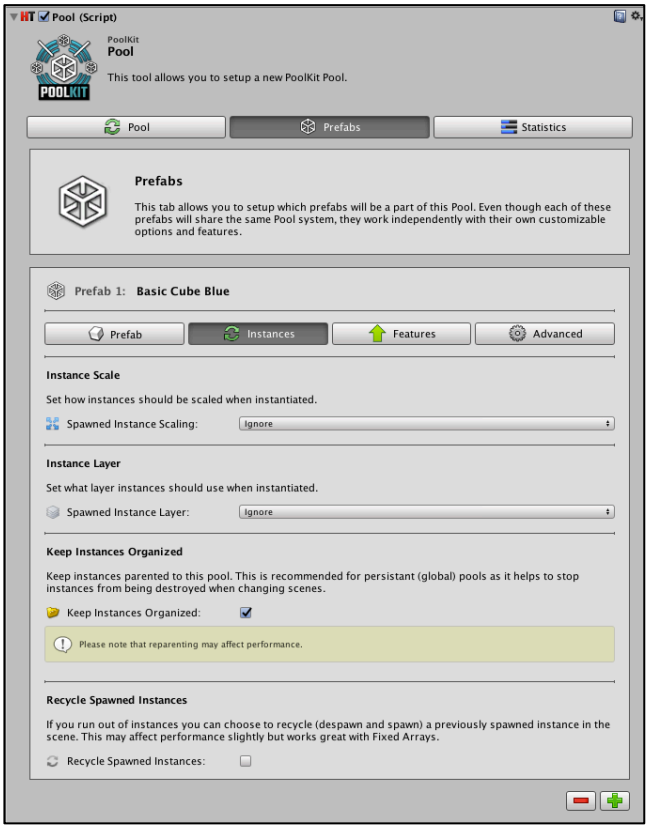
Depending on the Pool Type you've chosen, the following options are available:

Keep Pool Sized Fixed
Uses the "Default Pool Size" as the number of instances to create.

Expand Within Limit
Uses the "Default Pool Size" as the initial number of instances to create. If new instances are needed, they will be created until the pool hits the limit set in "Maximum Pool Size".

Always Expand Pool When Needed
Uses the "Default Pool Size" as the initial number of instances to create. If new instances are needed, the pool will keep creating new ones without any limits.

The Prefabs Tab (Instances)



Instance Scale

How should new instances be scaled when they are spawned?

Ignore:
No scaling is applied.

Prefab Scale:
The default scale of the prefab will be used.

Pool Scale:
The local scale of the pool will be used.

Custom Scale:
You can set a custom scale to be used.

Random Range Custom Scale:
You can set two custom scales to be used as a random range. The 'Minimum Spawn Scale' will act as the lowest possible values allowed and the 'Maximum Spawn Scale' acts as the limit.

Random Range Proportional Scale:
This is similar to the previous 'Random Range Custom Scale', except it uses two numbers (floats) as the minimum and maximum range. This allows you to have randomized results but keep the scale a uniform size.

Reset Scale On Every Spawn:
Most options allow you to re-apply the scale settings every time the same instance is re-spawned.

Instance Layer

What Layer should new instances be set to when they are spawned?

Ignore:
No Layer is set.

Prefab Layer:
The layer is set to the default layer of the prefab.

Pool Layer:
The layer is set to the layer of the Pool gameObject.

Custom Layer:
You can select the layer to use from a dropdown list.

Reset Layer On Every Spawn:
Most options allow you to re-apply the layer settings every time the same instance is re-spawned.

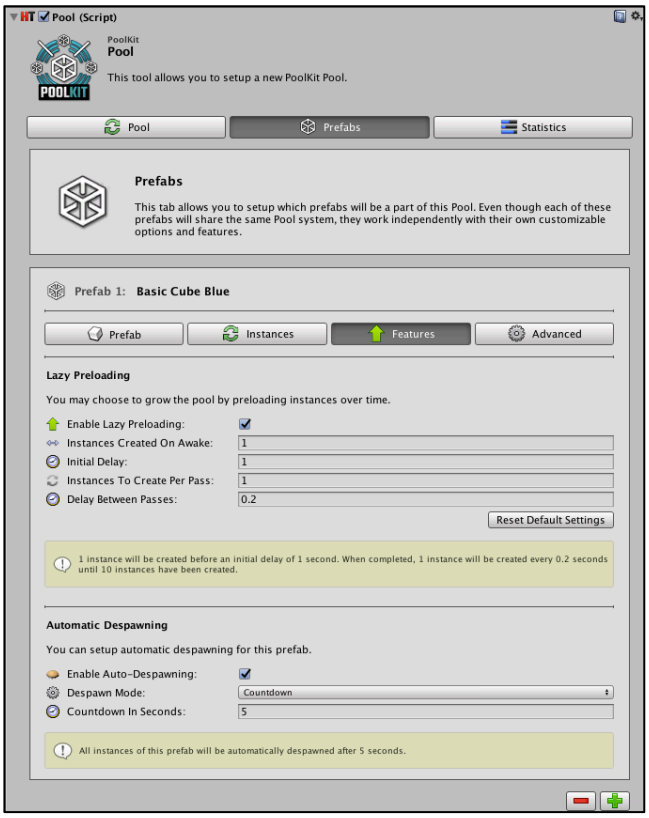
Keep Instances Organized

This keeps instances parented to the Pool's Transform. This is recommended for global pools as it helps to stop instances from being destroyed when changing scenes. Please note that when Unity moves objects from one Transform to another it may have a small affect on performance. If your Pools are local, it is recommended to turn this off in finished builds for the best performance.

Recycle Spawned Instances

Recycling is a great feature (especially for Fixed Size pools) that allow you to re-use existing instances when you run out. For example, if you try to spawn an instance but you've already used the ones available in the pool, this feature will despawn the oldest instance and re-spawn it instantly, giving you a way of continuously providing instances without going over your pool limits. If this feature is turned off and you exceed your pool size, null will be returned when trying to spawn something new.

The Prefabs Tab (Features)



Lazy Preloading

Pools using the "Dynamic List" or "Automatic" Pool Type can use lazy preloading to slowly grow the pool over time. The great thing about this is it can be used independently on the specific Prefabs that require it!

Enable Lazy Preloading:
Enables or Disables the feature.

Instances Created On Awake:
When the Pool runs its Awake method, how many instances should be created right away? 1 is the minimum.

Initial Delay:
How many seconds should the system wait until more instances are created?

Instances To Create Per Pass:
After the initial delay, how many new instances should be created on each pass? This loop will continue until the pool has reached the 'Default Pool Size' (found in the Prefab section).

Delay Between Passes
Every time a pass of new instances is created, how long should the pool wait before looping and creating another pass of instances? This value is in seconds.

Auto-Despawning

Each prefab in your pool can also be setup with Auto-Despawning functionality (no scripting required!). This configuration is great for instances that will always be used in a simple and repetitive way. Examples of this would be a blood squirt, puff of smoke, audio effect, etc. If you want to use the same instances for various things, you should setup their own Despawner component on the prefab itself.

Enable Automatic-Despawning:
Enables or Disables the feature.

Despawn Mode: Countdown
Every time an instance of this prefab is spawned, a countdown timer will determine when it will automatically be despawned. You can set a custom value in seconds.

Despawn Mode: Countdown Random Range
Every time an instance of this prefab is spawned, a countdown timer with a random range will determine when it will automatically be despawned. The 'Min Countdown In Seconds' determines the lowest value and the 'Max Countdown In Seconds' will set the largest value to use. These values are counted in seconds.

Despawn Mode: Wait For Audio To Finish
If the prefab has an AudioSource component, you can choose this option to wait for the audio to finish playing before it is automatically despawned.

Despawn Mode: Wait For Particle System To Finish
If the prefab has a Particle System component, you can choose this option to wait for the particle system to finish playing before it is automatically despawned.

The Prefabs Tab (Advanced)

Notifications

Notifications allow you to trigger custom functions in your instances when they are spawned and despawned. To use this feature, your prefab must have a script with public methods that are named "OnSpawn" and "OnDespawn"

Alternatively, the fastest way to trigger these kinds of actions is to use Unity's "OnEnable" and "OnDisable" methods. If you need to separate this out (for example, to only do certain actions when pooling takes place), the following methods are available:

None:
No Notifications will be sent (fastest)

PoolKit Listeners [Recommended Approach]:
The IPoolKitListener interface can be added to any MonoBehaviour script, which allows you to use PoolKit's unique caching to trigger notifications with amazing speed (it is up to 3x faster than using delegates!). You can easily add an IPoolKitListener interface to any of your own scripts like this:

```
using HellTap.PoolKit;

public class MyCoolScript : MonoBehaviour, IPoolKitListener {

    public void OnSpawn( Pool pool ){
        // Add Spawn stuff here!
    }

    public void OnDespawn(){
        // Add Despawn stuff here!
    }
}
```

When using this feature, PoolKit caches the prefab and all of its children containing an IPoolKitListener interface when they are instantiated. This allows you to trigger ultra-fast methods on any script of the prefab or its children with great performance!

Send Message:
An alternative approach is to have PoolKit use "SendMessage" to trigger "OnSpawn" and "OnDespawn" methods. This will only trigger functions on the main prefab but not it's children. Frequently sending SendMessage actions does have a performance hit.

BroadCast Message:
Similar to SendMessage but broadcasts it to all child objects also. This is a much slower approach and is generally not recommended.

Enable Delegates & Events

Unlike the Pool's "Enable Delegates & Events" which allows you to subscribe to spawning and despawning events, this setting on the prefab allows you to override how PoolKit instantiates and destroys instances. This is an advanced feature that most people will not need but it is recommended to view the example scene and script that comes with PoolKit for a better explanation on how to do this correctly.

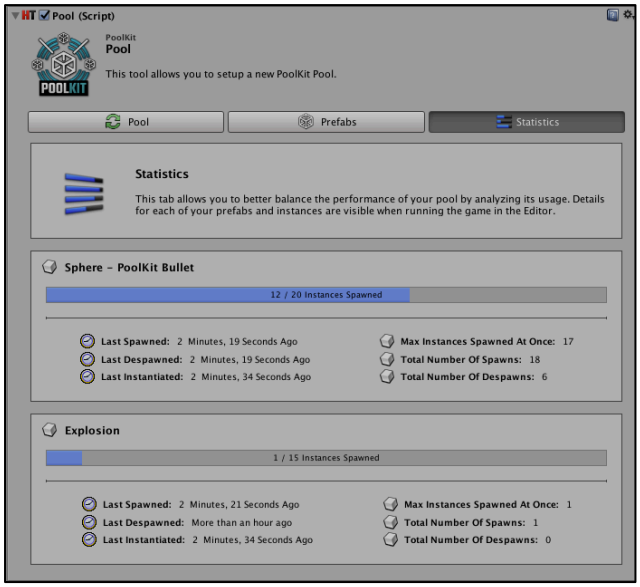
Enable Instantiation Events:
This allows you to override how instances of this prefab will be instantiated using your own scripts. You can also override this setting using the API. You can subscribe to the event like this:

```
// Make sure to add: using HellTap.PoolKit;
public GameObject prefabToOverride = null;
Pool findPool;

void OnEnable(){

    // Find the pool containing the prefab we want to track and subscribe
    if(findPool==null){
        findPool = PoolKit.GetPoolContainingPrefab( prefabToOverride );
    }

    if(findPool!=null){ findPool.OnCreateInstance += OnCreateInstance; }
}

void OnDisable(){

    // Unsubscribe from the pool if it was found
    if(findPool!=null){ findPool.OnCreateInstance -= OnCreateInstance; }

}

GameObject OnCreateInstance( GameObject prefab ) {

    // NOTE: Make sure to return the created GameObject!
    if ( prefab != null ){ return Instantiate( prefab ); }

    // Return null if something goes wrong
    return null;
}
```

Enable Destroy Events:
This allows you to override how instances of this prefab will be destroyed using your own scripts. You can also override this setting using the API. You can subscribe to the event like this:

```
// Make sure to add: using HellTap.PoolKit;
public GameObject prefabToOverride = null;
Pool findPool;

void OnEnable(){

    // Find the pool containing the prefab we want to track and subscribe
    if(findPool==null){
        findPool = PoolKit.GetPoolContainingPrefab( prefabToOverride );
    }

    if(findPool!=null){ findPool.OnDestroyInstance += OnDestroyInstance; }
}

void OnDisable(){

    // Unsubscribe from the pool if it was found
    if(findPool!=null){ findPool.OnDestroyInstance -= OnDestroyInstance; }

}

void OnDestroyInstance (GameObject instance) {

    // NOTE: Make sure to destroy the instance!
    if (instance!= null ){ Destroy( instance ); }

}
```

The Statistics Tab



The Statistics tab allows you to view real-time usage statistics of your pool while the game is running in the Editor. It can also suggest ways to improve your pools in certain conditions.

Each prefab in the pool is clearly separated in its own section, allowing you to focus in on the instances you want to optimize.

To optimize your pools, you will want to find a point where you have a pool very close to the maximum number of instances created at once. For example, if you only ever spawn 5 instances of a prefab, but your pool size is set to use 10 instances, you are wasting resources. You will want to set your pool size to 5 or perhaps 6 in order to optimize memory usage. The statistics tab offers a great visual way to do this.

The Instances Spawned Bar

The big blue bar represents how many instances of a certain object have been spawned in relation to its pool size.

Last Spawned

An easy to read countdown since the last time an instance was spawned.

Last Despawned

An easy to read countdown since the last time an instance was despawned.

Last Instantiated

An easy to read countdown since the last time an instance was instantiated.

Max Instances Spawned At Once

This value tracks the largest amount of instances to be spawned at any one time. This is a very helpful value that can often give you a good estimate of the pool size this prefab pool should be set to.

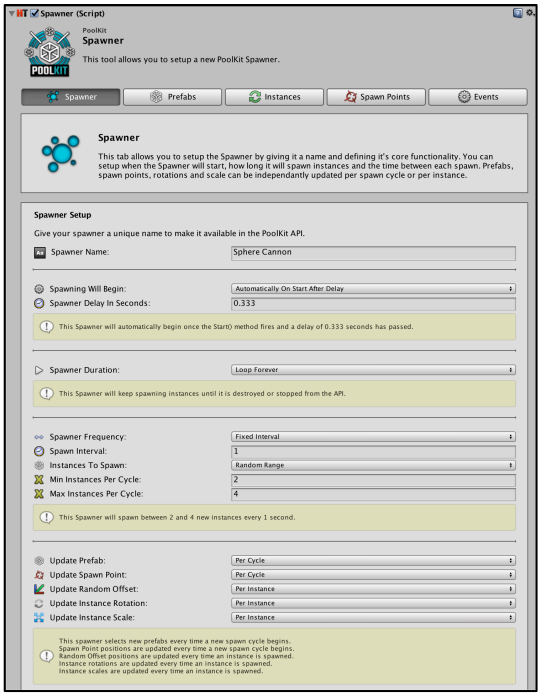
Total Number Of Spawns

This is the number of instances that have been spawned (including re-spawns) since the pool was created.

Total Number Of Despawns

This is the number of instances that have been despawned since the pool was created.

Setting Up The Spawner Component



The Spawner script allows you to setup a new PoolKit Spawner. This powerful component works by using a similar concept to Unity's own Particle System, except that it spawns instances from your Pools rather than particles. The Spawner can be used to setup special effects, enemy spawn points, weapon rigs and so much more! It consists of the Spawner, Prefabs, Instances, Spawn Points and Events Tabs.

The Spawner Tab

The Spawner tab allows you to setup the core settings of the Spawner. Firstly, you should give it a name to make it accessible from the API. You can setup options such as when the Spawner will start, how long it will spawn instances, and the time between each spawn cycle.

Spawner Name

The spawner name is used to access the spawner via the API. It is best to use something descriptive and short.

Spawning Will Begin

You can choose a variety of options to determine when this spawner should begin spawning instances. The following choices are available:

Automatically At Start:
The Spawner will automatically begin when the Start() method is fired.

Automatically At Start After Delay:
The Spawner will automatically begin when the Start() method is fired, and after a custom delay in seconds has completed. You can set 'Spawner Delay In Seconds' in the Editor to determine the delay.

Automatically On Enable:
The Spawner will automatically begin when the OnEnable() method is fired. The difference between this and the Start() method is it can be used as its own spawnable object. This means that when if the Spawner itself is spawned, OnEnable is triggered and spawning will be reset.

Automatically On Enable After Delay:
This works just like the above, but allows you to add a 'Spawner Delay In Seconds' to add a further delay before spawning begins.

Only When Called By Script:
The spawner will only begin if explicitly called by a script. You can do that like this:

```
using HellTap.PoolKit;

public class MyCoolScript : MonoBehaviour {

    void Start{

        // Find the Spawner using its Spawner Name and start it!
        Spawner mySpawner = PoolKit.GetSpawner("MySpawner");
        mySpawner.Play();
    }
}
```

Never:
This setting will never allow the Spawner to start. It can be used for debugging or to manually stop a Spawner using scripting.

Spawner Duration

The spawner duration allows you to tell the Spawner how long to keep spawning instances. The following options are available:

Play Once:
The spawner will spawn a single instance and stop.

Repeat X Cycles:
You can use the 'Spawn Cycles To Repeat' field to set how many spawn cycles you want the spawner to create before it stops.

Spawn X Instances:
You can use the 'Instances To Spawn' field to set how many instances you want to spawn before it stops. The spawner will stop early if this number is reached in the middle of a spawn cycle.

Countdown Timer:
You can use the 'Countdown In Seconds' field to set how many seconds the spawner should wait before it stops.

Loop Forever:
The Spawner will keep spawning instances until it is destroyed or stopped from a script.

Spawner Frequency

The spawner frequency section allows you to configure how often the spawner should start a new spawn cycle:

Fixed Interval:
The Spawner will start a new spawn cycle using a fixed interval of time. You can use the 'Spawn Interval' field to determine how many seconds between each spawn cycle.

Random Range:
The Spawner will start a new spawn cycle using a random range of time. You can use the 'Minimum Interval In Seconds' and 'Maximum Interval In Seconds' fields to setup a random range between each spawn cycle.

Instances To Spawn

The Spawner works by spawning a number of instances every spawn cycle. You may choose whether the number of instances spawned is a fixed number or is randomized every cycle:

Fixed Number:
The Spawner will create a fixed number of instances every spawn cycle. Changing the 'Instances Per Cycle' field tells the spawner how many instances should be spawned.

Random Range:
The Spawner will create a random number of instances every spawn cycle. Changing the 'Min Instances Per Cycle' field tells the spawner the lowest number of instances to spawn and the 'Max Instances Per Cycle' field sets the upper limit.

Spawner Updates

By default, the spawner updates the selected prefab, spawn point, randomized offset (if enabled), rotations and scale every time a new instance is spawned. However, you can create interesting effects and setups by setting some of these to update only at the start of each new spawn cycle.

As an example, you could have all instances of a spawn cycle be positioned at the same spawn point by setting the "Update Spawn Point" field to "Per Cycle" rather than "Per Instance".

Update Prefab:
Prefabs can be selected every time an instance is spawned or when a spawn cycle begins.

Update Spawn Point:
Spawn Points can be selected every time an instance is spawned or when a spawn cycle begins.

Update Random Offsets:
Randomized Offsets can be updated with every new instance or when a spawn cycle begins.

Update Instance Rotation:
Rotations applied to instances can be updated with every instance or when a spawn cycle begins.

Update Instance Scale:
Scales applied to instances can be updated with every instance or when a spawn cycle begins.

The Prefabs Tab

The prefabs tab allows you to setup which prefabs should be spawned by the spawner. Their related pools will be found automatically.

Please note that you can also use prefabs that are not a part of a pool. When the Spawner detects this, these instances will be instantiated instead of spawned which will cause memory allocations.

Prefab Selection Mode

You can setup how prefabs will be selected every time an instance should be spawned. The following options are available:

Sequence Ascending:
Prefabs will be chosen using the list in ascending order.

Sequence Descending:
Prefabs will be chosen using the list in descending order.

Ping Pong Ascending:
Prefabs will be chosen using the list in ascending order. When it gets to the end, it will loop back in descending order.

Ping Pong Descending:
Prefabs will be chosen using the list in descending order. When it gets to the beginning, it will loop back to the end in ascending order.

Random:
Prefabs will be chosen randomly.

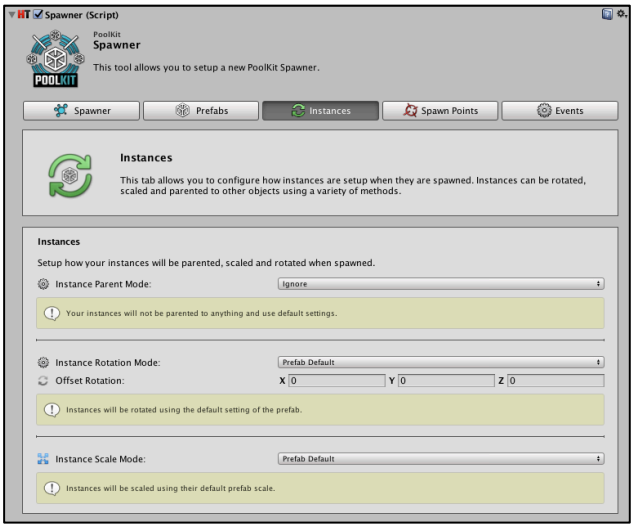
Random With Weights:
Prefabs will be chosen randomly using a Weighted Chance. Each prefab will now have a Chance slider allowing you give certain prefabs more of a chance over others. Setting a prefab to 0% means it will never be selected while setting it to 100% gives it a full random chance.

Prefab List

The prefab list tells the spawner what prefabs should be spawned. Use the green plus arrow to create new prefab slots and the red minus arrows to remove them. You can also use the vertical yellow arrows to change the order of prefabs in the list.

The Instances Tab

The instances tab allows you to configure how instances are setup when they are spawned. Instances can be rotated, scaled and parented to other objects using a variety of methods.



Instance Parent Mode

You can setup if instances should be parented to other objects using a variety of methods. Please note that you generally shouldn't do this to instances in global pools.

Ignore:
Instances will not be parented to any objects and will be ignored.

Re-parent To Spawner:
Instances will be parented to the Transform of the spawner.

Re-parent To Spawn Point:
Instances will be parented to the currently selected spawn point (Please note this only works if you are using a Transform List in the Spawn Points tab).

Re-parent To Custom Transform:
Instances will be parented to the Transform you set in the 'Custom Parent Transform' field.

Instance Rotation Mode

You can setup if instances should be rotated when they are spawned. The following options are available:

Prefab Default:
Instances will be rotated using the default rotation of its own prefab.

Spawner Rotation:
Instances will be rotated using the rotation of the Spawner.

Spawn Point Rotation:
Instances will be rotated using the rotation of the last Spawn Point (Please note this only works if you are using a Transform List in the Spawn Points tab).

Custom Euler Angles:
Instances will be rotated using the custom euler angles set in the 'Custom Euler Angles' field.

Random Rotation:
Instances will be rotated randomly.

Offset Rotation:
Many of the options allow you to add an additional rotation offset .

Instance Scale Mode

You can setup if instances should be scaled when they are spawned. The following options are available:

Pool Default:
Scaling will be ignored, allowing the default settings of the pool to be used.

Prefab Default:
Instances will be scaled using the default scale of its own prefab.

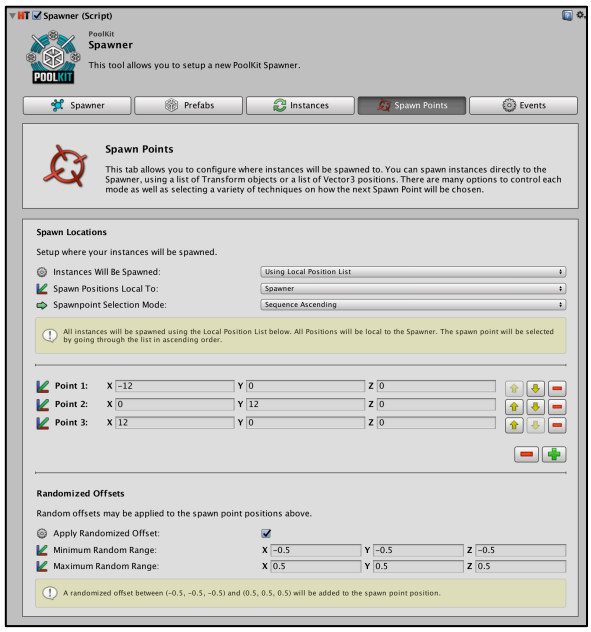
Spawner Scale:
Instances will be scaled using the scale of the spawner.

Custom Local Scale:
Instances will be scaled using the custom local scale in the 'Custom Local Scale' field.

Random Range Scale:
Instances will be scaled using a random range. The 'Minimum Local Scale' defines the smallest values that can be used and the 'Maximum Local Scale' field defines the largest.

Random Range Proportional Scale:
This is similar to the above but uses 2 numbers (floats) to define the range. This will randomize the scale of the instance but keep it uniform.

The Spawn Points Tab



This tab allows you to configure where instances will be spawned.

You can spawn instances directly to the Spawner, using a list of Transform objects or a list of Vector3 positions (local or global).

There are many options to control each mode as well as selecting a variety of techniques on how the next Spawn Point will be chosen.

Instances Will Be Spawned

This field allows you to setup what kind of spawn point system to use. The following options are available:

At This Transform:
This is the simplest setup. All instances will be spawned at the position of the Spawner.

Using Transform List:
This will use a list of Transforms to act as Spawn Points. It is generally a good idea to use child objects of the Spawner for this purpose but this is not a requirement.

Using Local Position List:
This will use a list of Vector3 positions to act as Spawn Points. These positions will be local to the option selected in 'Spawn Positions Local To' (by default this would be the Spawner).

Using Global Position List:
This will use a list of global Vector3 positions to act as Spawn Points.

Spawn Points Local To

This field is visible when selecting to use a local position list. You can choose to have positions local to the Spawner or to a custom Transform set in the 'Spawn Local To' field.

Spawn Point Selection Mode

You can setup how spawn points will be selected every time an instance should be spawned. The following options are available:

Sequence Ascending:
Spawn points will be chosen using the list in ascending order.

Sequence Descending:
Spawn points will be chosen using the list in descending order.

Ping Pong Ascending:
Spawn points will be chosen using the list in ascending order. When it gets to the end, it will loop back in descending order.

Ping Pong Descending:
Spawn points will be chosen using the list in descending order. When it gets to the beginning, it will loop back to the end in ascending order.

Random:
Spawn points will be chosen randomly.

Random With Weights:
Spawn points will be chosen randomly using a Weighted Chance. Each spawn point will now have a 'Chance' slider allowing you give certain spawn points more of a chance over others. Setting a the spawn point to 0% means it will never be selected while setting it to 100% gives it a full random chance.

Transform List

This is visible when 'Instances Will Be Spawned' is set to 'Using Transform List'. The Transform List tells the spawner what Transforms should be used as spawn points. Use the green plus arrow to create a new spawn point slot and the red minus arrows to remove them. You can also use the vertical yellow arrows to change the order of the list.

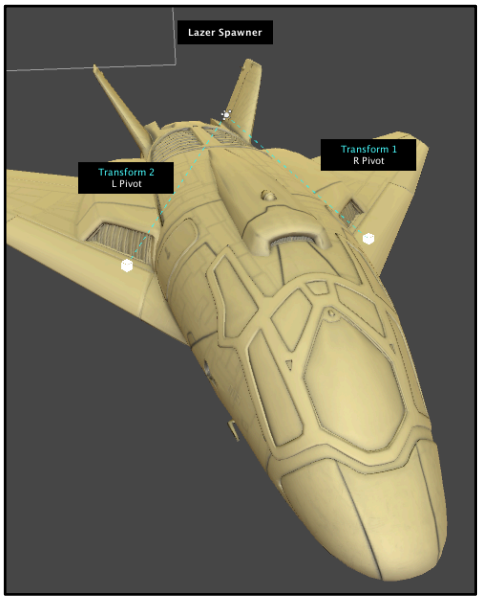
Position List

This is visible when 'Instances Will Be Spawned' is set to 'Using Local Position List' or 'Using Global Position List'. The Position List tells the spawner what positions should be used as spawn points. Use the green plus arrow to create a new spawn point slot and the red minus arrows to remove them. You can also use the vertical yellow arrows to change the order of the list.

Notes About The Scene View

You may notice that when you select the spawner in the Hierarchy, you will see gizmos displayed in the Scene view to help you understand how the Spawner is working.

The spawner name, waypoint labels and info and more can all be modified in the PoolKit section of Unity Preferences.



Randomized Offsets

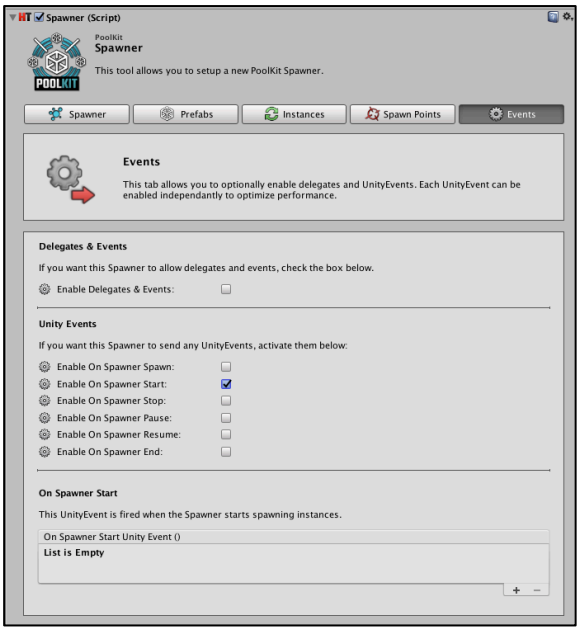
At the bottom of all Spawn Point modes is a section that allows randomized offsets to be used. This feature will use a random range of Vector3 values and apply it to the position of the current spawn point. This allows you to have much more variety without the need to setup multiple spawn points.

Apply Randomized Offset:
If this is checked, Randomized Offsets will be enabled.

Minimum Random Range:
The minimum random offset values to be applied to the current spawn point position.

Random:
The maximum random offset values to be applied to the current spawn point position.

The Events Tab



This tab allows you to optionally enable delegates and Unity Events. Each Unity Event can be enabled independently to optimize performance.

Enable Delegates & Events

Enable this if you wish to use the API to subscribe to the Spawner.

If you do not need this feature, make sure to leave this unchecked to optimize performance. You can subscribe to these Events like this:

```
// using HellTap.PoolKit;

// Find the Spawner using its Spawner Name
Spawner mySpawner = PoolKit.GetSpawner("MySpawner");
if(mySpawner != null){

    // Subscribe To Events
    mySpawner.onSpawnerSpawn += onSpawnerSpawn;
    mySpawner.onSpawnerStart += onSpawnerStart;
    mySpawner.onSpawnerStop += onSpawnerStop;
    mySpawner.onSpawnerPause += onSpawnerPause;
    mySpawner.onSpawnerResume += onSpawnerResume;
    mySpawner.onSpawnerEnd += onSpawnerEnd;
}
```

Unity Events

The following Unity Events can be enabled in the Editor:

On Spawner Spawn <Transform>:
This Unity Event will pass the last instance created by the spawner to a method of your choice. Please note that using this excessively may have an impact on performance.

On Spawner Start:
This Unity Event will fire when the spawner has just started spawning instances.

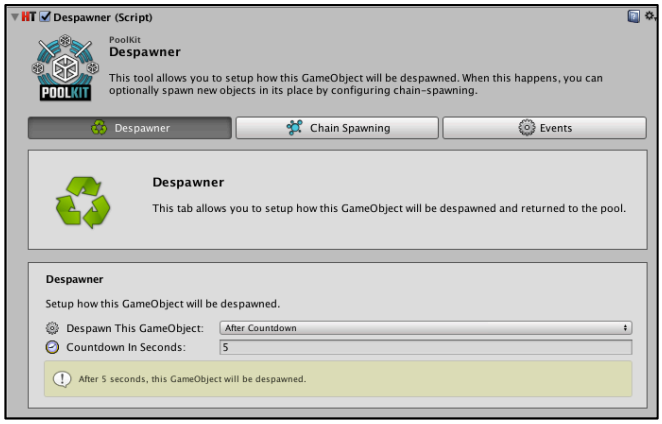
On Spawner Stop:
This Unity Event will fire when the spawner has stopped spawning instances.

On Spawner Pause:
This Unity Event will fire when the spawner has been paused.

On Spawner Resume:
This Unity Event will fire when the spawner has been resumed.

On Spawner End:
This Unity Event will fire when the spawner has finished its duration. For example, it has completed a countdown or repeated X times, etc.

Setting Up The Despawner Component



The Despawner Component allows you to setup how a specific GameObject will be despawned. When this happens, you can optionally spawn new objects in its place by configuring Chain-Spawning. It consists of the Despawner, Chain-Spawning and Events tabs.

The Despawner tab is an easy to use all-purpose way to automate the despawning of instances. This can be achieved by setting the 'Despawn This GameObject' dropdown menu. The following configurations are available:

The Despawner Tab – After Countdown

The despawner will automatically despawn after a countdown timer. This is set in seconds using the 'Countdown In Seconds' field.

The Despawner Tab – After Countdown With Random Range

The despawner will automatically despawn after a countdown timer with a random range. This is set in seconds using the ' Min Countdown In Seconds' field to define the shortest timer and the 'Max Countdown In Seconds' to set the longest timer.

The Despawner Tab – After Particle System Finishes

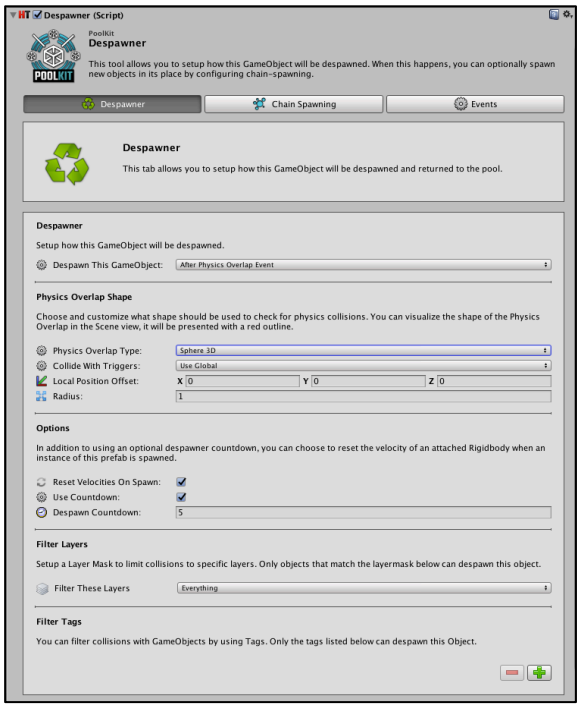
The despawner will wait for a Particle System to finish before despawning. You can choose whether the Particle System should reside on the current GameObject or a child object. Finally, you can choose to have the Particle System auto-play when it is spawned.

The Despawner Tab – After Audio Source Finishes

The despawner will wait for an AudioSource to finish playing before despawning. You can choose whether the AudioSource should reside on the current GameObject or a child object. Finally, you can choose to have the AudioSource automatically play when it is spawned.

The Despawner Tab – After Physics Overlap Event

The despawner can wait for this specially designed PoolKit Physics event to provide zero allocation collision events.



Physics Overlap Type

A collider isn't required for this physics event to work. This setup works especially well for projectiles but can also be used with colliders too. A red shape will be drawn in the Scene view to show you a visualization of the overlapping collision shape.

Sphere 3D
A 3D sphere will be used to check for collisions.

Circle 2D
A 2D Circle will be used to check for collisions in 2D based games.

Box 3D
A 3D cube will be used to check for collisions.

Box 2D
A 2D box will be used to check for collisions in 2D based games.

Collide With Triggers

3D Colliders allow you to also collide with triggers. The options are:

Use Global
Uses the global settings setup in the Physics screen.

Ignore
All colliders that set as triggers will be ignored.

Collide
Triggers will also be checked for collisions.

Local Position Offset

Allows you to change the initial position of the physics shape.

Radius / Scale

Allows you to change the size of the physics shape.

Options

The following options are also available

Reset Velocities On Spawn
If this GameObject has an attached Rigidbody, you can reset its velocities every time it is spawned.

Use Countdown
You can choose to also activate a countdown to despawn the instance. This is set in seconds using the 'Despawn Countdown' field.

Filter Layers

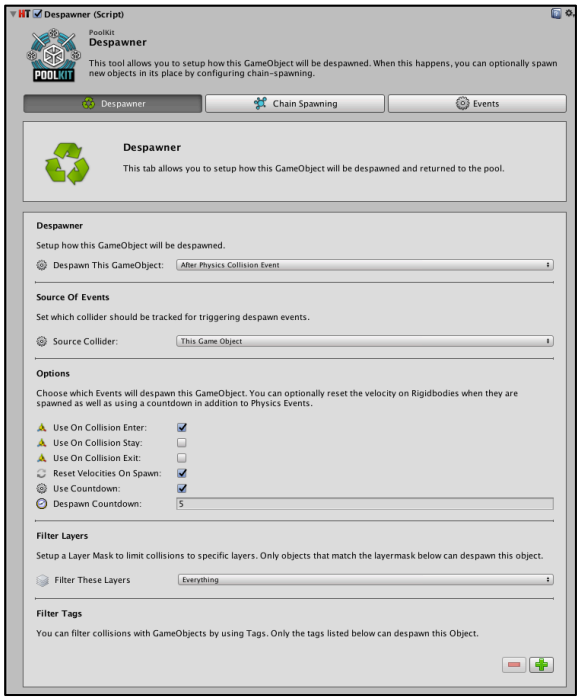
You can filter the collisions using this LayerMask. Only the layers that are checked in the list will be allowed to despawn the instance.

Filter Tags

You can also filter the collisions using Tags as well as the LayerMask. Only the Tags that are in the list will be allowed to despawn the instance. New Tags can be added by pressing the green '+' button. If no Tags are set, this feature will be ignored.

The Despawner Tab – After Collision Event

The despawner can wait for Unity's built in collision-based physics events to despawn.



Source Collider

Firstly, you need to define what Collider the despawner should check. The following options are available:

This GameObject
The despawner should use a Collider found on this GameObject.

Another Child GameObject:
This despawner should use a Collider found on a child GameObject. You can set this using the 'Use This GameObject' field.

Manual Setup
It is important to note that Unity's physics events (such as OnCollisionEnter, OnCollisionStay, etc.) all create small amounts of garbage in memory when triggered. Unfortunately, there doesn't seem to be a way around this for the moment (other than using PoolKit's Physics Overlap Event which doesn't generate any allocations at all). Because of this Unity limitation, PoolKit uses a unique optimization method where it will create specific Despawner Events at runtime based on the physics events you choose. In other words, this means that after the instance is instantiated, you should get the same kind of performance as you would with a custom script!

Advanced users can use the 'Manual Setup' setting to manually place 'Despawner Event' components on specific child objects to customize the source of colliders (for example, to use more than one source collider). These components can be found in the **Plugins > Hell Tap Entertainment > PoolKit > Despawner** folder. You should use the component with the relevant physics event. For example, if you wanted to track OnCollisionEnter, you would use the DespawnerEvent_OnCollisionEnter component.

You will also need to set the despawner field in the Editor to reference the Despawner. The screenshot above shows how you would manually setup an OnCollisionEnter event.

Options

You can choose which Physics Events will despawn the GameObject. You can optionally reset the velocity on Rigidbodies when they are spawned as well as using a countdown in addition to the Physics Events.

Use On Collision Enter
The despawner will track collisions on the OnCollisionEnter event.

Use On Collision Stay
The despawner will track collisions on the OnCollisionStay event.

Use On Collision Exit
The despawner will track collisions on the OnCollisionExit event.

Reset Velocities On Spawn
When this instance is spawned, the despawner will attempt to reset the velocity on the Rigidbody found on the source collider.

Use Countdown
In addition to tracking the above physics events, you can choose to also activate a countdown to despawn the instance. This is set in seconds using the 'Despawn Countdown' field.

Filter Layers

You can filter the collisions using this LayerMask. Only the layers that are checked in the list will be allowed to despawn the instance.

Filter Tags

You can also filter the collisions using Tags as well as the LayerMask. Only the Tags that are in the list will be allowed to despawn the instance. New Tags can be added by pressing the green '+' button. If no Tags are set, this feature will be ignored.

The Despawner Tab – After Trigger Event

'After Trigger Event' works almost identically to the After Collision Event. The only difference is a difference in some of the Physics Events that are available:

Use On Trigger Enter
The despawner will track collisions on the OnTriggerEnter event.

Use On Trigger Stay
The despawner will track collisions on the OnTriggerStay event.

Use On Trigger Exit
The despawner will track collisions on the OnTriggerExit event.

The Despawner Tab – After Collision 2D Event

'After Collision 2D Event' works almost identically to the After Collision Event. The only difference is a Collider2D must be present on the source collider and the set of Physics Events that are available:

Use On Collision Enter 2D
The despawner will track collisions on the OnCollision2DEnter event.

Use On Collision Stay 2D
The despawner will track collisions on the OnCollision2DStay event.

Use On Collision Exit 2D
The despawner will track collisions on the OnCollision2DExit event.

The Despawner Tab – After Trigger 2D Event

'After Trigger 2D Event' works almost identically to the After Collision Event. The only difference is a Collider2D must be present on the source collider and the set of Physics Events that are available:

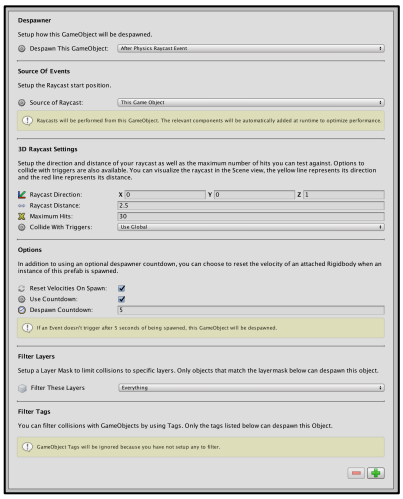
Use On Trigger Enter 2D
The despawner will track collisions on the OnTrigger2DEnter event.

Use On Trigger Stay 2D
The despawner will track collisions on the OnTrigger2DStay event.

Use On Trigger Exit 2D
The despawner will track collisions on the OnTrigger2DExit event.

The Despawner Tab – After Raycast Event

'After Raycast Event' shares many settings with the other events. However, there are specific options for setting up the raycast:



Raycast Direction
The direction of the raycast. This should be a normalized Vector3.

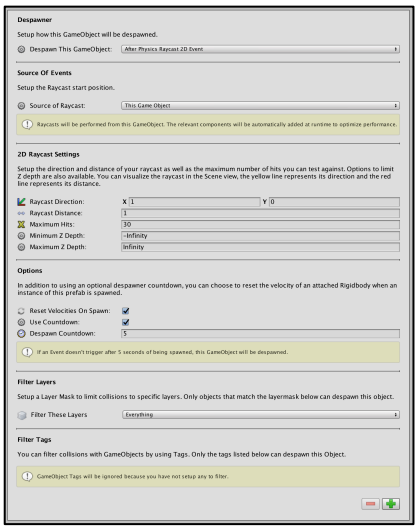
Raycast Distance
The maximum distance of the raycast.

Maximum Hits
The maximum number of hits to test against.

Collide With Triggers
Should this raycast collide with triggers?

The Despawner Tab – After Raycast 2D Event

'After Raycast 2D Event' shares many settings with the other events. However, there are specific options for setting up the raycast:



Raycast Direction
The direction of the raycast. This should be a normalized Vector2.

Raycast Distance
The maximum distance of the raycast.

Maximum Hits
The maximum number of hits to test against.

Minimum Z Depth
Only include objects with a Z coordinate (depth) greater than or equal to this value.

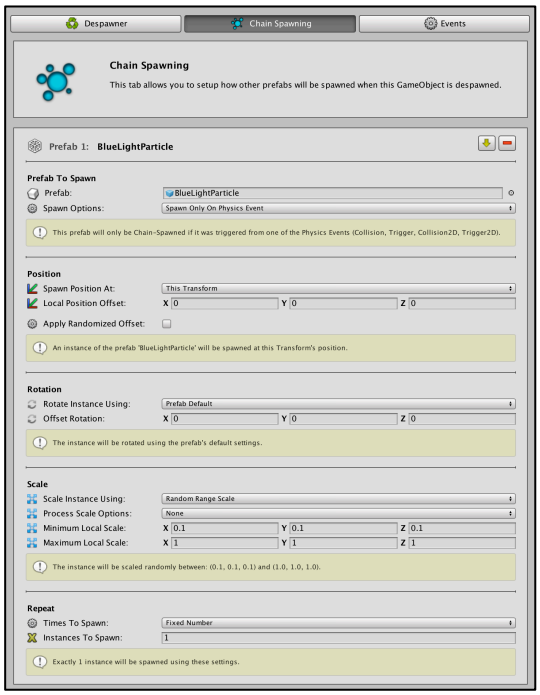
Maximum Z Depth
Only include objects with a Z coordinate (depth) less than or equal to this value.

The Despawner Tab – After Called By Script

The despawner will only despawn if a script tells it to. The purpose of this setup is so users can still take advantage of the Chain Spawning and Events tab while using a totally custom way of despawning. You can manually send actions to the Despawner like this:

```
// using HellTap.PoolKit;

// Cache the despawner on this GameObject and tell it to despawn,
Despawner myDespawner = gameObject.GetComponent<Despawner>();
if(myDespawner!= null){ myDespawner.Despawn(); }
```

The Chain-Spawning Tab



This tab allows you to setup how new prefabs will be spawned when this GameObject is despawned.

You can setup any number of prefab slots to spawn, each with their own custom settings. You can also setup multiple slots using the same prefab with different settings.

Prefab To Spawn

Drag and drop a prefab from the project pane in order to spawn it.

Spawn Options

The following Spawn Options are available:

Always Spawn
This prefab will always be spawned.

Spawn Only On Physics Event
This prefab will only be spawned if a Physics-based Event (OnCollisionEnter, OnTriggerStay, etc.) triggered the despawning process.

Spawn Except On Physics Event
This prefab will only be spawned if a Physics-based Event (OnCollisionEnter, OnTriggerStay, etc.) did NOT trigger the despawning process.

Never Spawn
This prefab will never be spawned under any condition. This is useful for debugging and custom setups. Please note that if this isn't needed in finished builds, it's much better to remove it from the editor altogether to save memory.

Use Physics Event Filters

If you're using a physics event, you can enable advanced filtering options to have chain spawning work with specific layers, tags and GameObject names. This option will enable the following three filtering options:

Filter Layers

You can filter chain spawning using this LayerMask. Only the layers that are checked in the list will be allowed to spawn new prefabs in this entry.

Filter Tags

You can also filter chain spawning using Tags. Only the Tags that are in the list will be allowed to spawn new prefabs in this entry. New Tags can be added by pressing the green '+' button. If no Tags are set, this filter will be ignored.

Filter Names

If you would like to only chain spawn instances if the object collided with a specific GameObject, you can use this feature. Only the names that are in the list will be allowed to spawn new prefabs. New names can be added by pressing the green '+' button. If no names are set, this filter will be ignored.

Spawn Position At

Choose where this instance will be spawned. The following options are available:

This Transform
The new instance will be spawned at the same position as the despawner.

Another Child Transform
The new instance will be spawned at the same position of a child Transform set in the 'Another Child Transform' field.

Last Collision
The new instance will be spawned using the position of the last collision point. If this cannot be determined, the position of the despawner will be used instead.

Local Position Offset

After the 'Last Position Offset' has been calculated, a local position offset will be applied to the final position.

Apply Randomized Offset

You can add some randomization to the final position by setting up a randomized range. This allows you to use the 'Minimum Random Range' field to set the minimum offset and the 'Maximum Random Range' field for the maximum offset.

Rotate Instance Using

Choose how this instance will be rotated. The following options are available:

Prefab Default
The new instance will be rotated using its default prefab rotation.

This Transform Rotation
The new instance will be rotated using the rotation of the despawner.

Custom Euler Angles
The new instance will be rotated using the custom euler angles set in the 'Custom Euler Angles' field.

Random Rotation
The new instance will be rotated randomly.

Offset Rotation

An additional rotation can be applied after the 'Rotate Instance Using' setting.

Scale Instance Using

Choose how this instance will be scaled. The following options are available:

Prefab Default
The new instance will be scaled using its default prefab scale.

Pool Default
The new instance will not be scaled, and use the default settings applied by the pool.

This Transform Scale
The new instance will be scaled using the scale of the despawner.

Custom Local Scale
The new instance will be scaled using the custom scale set in the 'Custom Local Scale' field.

Random Range Scale
The new instance will be scaled using a random range. The 'Minimum Local Scale' field will determine its smallest size and the 'Maximum Local Scale' will determine is largest size.

Random Range Proportional Scale
This is similar to the previous option except two floats will be used to determine the range. This randomizes the size of the instance but keeps it to a uniform scale.

Process Scale Options

The custom scaling options can also be processed to make it relative to the Transform's local scale. This allows you to scale spawned instances in very interesting ways. The following options are available:

None
No processing takes place. The results are used directly.

Multiply With Local Scale
The result is multiplied with every vector of the local scale.

Multiply With Smallest Local Scale Vector
The result is multiplied with the smallest vector of the local scale.

Multiply With Largest Local Scale Vector
The result is multiplied with the largest vector of the local scale.

Multiply With Average Local Scale Vector
The result is multiplied using the average of all three vectors of the local scale.

Multiply With Local Scale X
The result is multiplied using the X vector of the local scale.

Multiply With Local Scale Y
The result is multiplied using the Y vector of the local scale.

Multiply With Local Scale Z
The result is multiplied using the Z vector of the local scale.

Times To Spawn

This value sets how many times a new instance should be created using these settings. This works extremely well with the 'Randomized Offset' option as it allows you to create several instances with slight variations in position.

You can choose between a fixed number of instances to spawn or a random range.

The Events Tab

This tab allows you to optionally enable delegates and Unity Events. Each Unity Event can be enabled independently to optimize performance.

Enable Delegates & Events

Enable this if you wish to use the API to subscribe to the Despawner.

If you do not need this feature, make sure to leave this unchecked to optimize performance. You can subscribe to these Events like this:

```
// using HellTap.PoolKit;

// Cache the Despawner
Despawner myDespawner = gameObject.GetComponent<Despawner>();
if(myDespawner!= null){

    // Subscribe To Events
    mySpawner.onDespawnerDespawn += onDespawnerDespawn;
    mySpawner.onDespawnerCollided += onDespawnerCollided;
    mySpawner.onDespawnerChainSpawn += onDespawnerChainSpawn;
}
```

Unity Events

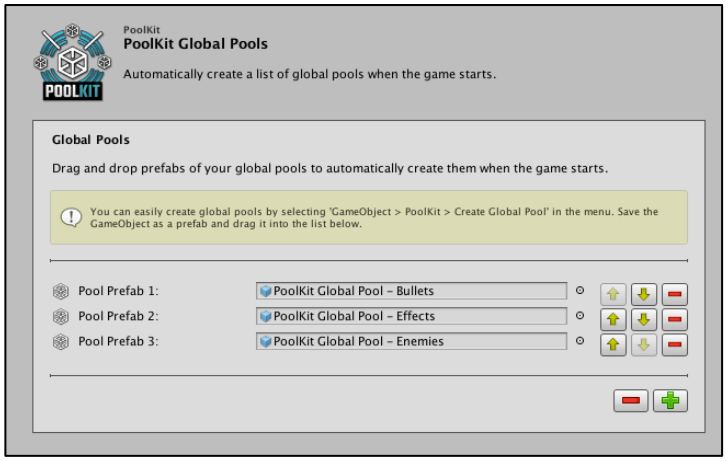
The following Unity Events can be enabled in the Editor:

On Despawner Despawn:
This Unity Event will trigger when the Despawner is about to despawn.

On Despawner Collided <GameObject>:
This Unity Event will pass the last GameObject that collided with the despawner. Please note that using this excessively may have an impact on performance.

On Despawner Chain Spawn <Transform>:
This Unity Event will pass the last instance created by the Chain-Spawner to a method of your choice. Please note that using this excessively may have an impact on performance.

Setting Up The PoolKit Global Pools Asset



The "PoolKit Global Pools" asset allows you to setup which global pools will be automatically created when the game starts. This makes global pools insanely easy to setup. You can use as many global pools as you want or none at all.

The asset can be easily accessed through the menu by selecting **GameObject > PoolKit > Setup Global Pools**.

To add a global pool to the list, press the green "+" button to add a new slot. You can then drag and drop a prefab into the list. The yellow arrows allow you to re-arrange the order while the red "-" button deletes items from the list.

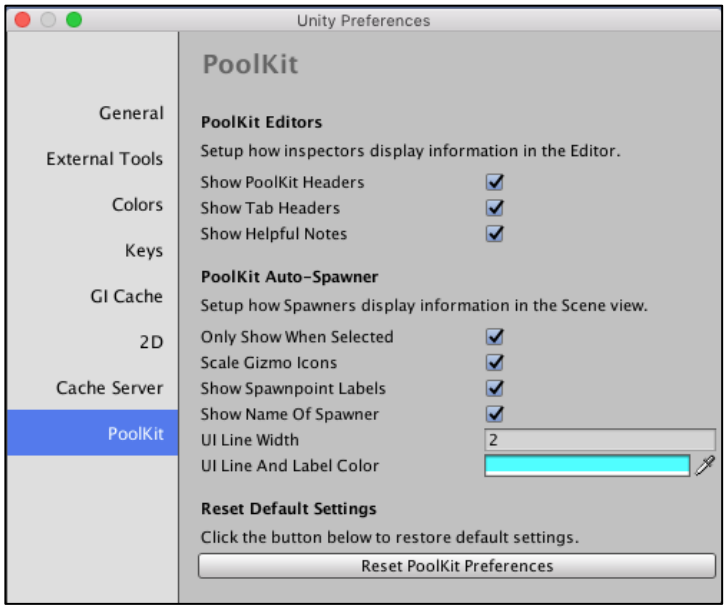
To correctly setup your global pools, you should stick to these guidelines:

- The prefabs you drag in must have a Pool component.
- The Pool components must be configured to be global ('Don't Destroy On Load' is checked)
- Every pool must have a unique Pool Name.
- You should not drag in custom global pool groups (that approach is designed to be setup manually). Only use individual Pool prefabs.

The visual editor will alert you if it detects any problems and offer suggestions.

NOTE: The 'PoolKit Global Pools' asset must always be located at *Plugins > Hell Tap Entertainment > PoolKit > Global > Resources > PoolKit Global Pools*. When accessing it using the menu command, PoolKit will re-create it if the file is accidentally deleted or moved.

Setting Up The PoolKit Preferences



The PoolKit Preferences pane allows you to customize how the PoolKit editors are presented to you, as well as how Spawners are visualized in the scene view. The following settings are available:

PoolKit Editor Settings

Show PoolKit Headers
This toggles the icon and info at the top of the PoolKit Editors.

Show Tab Headers
This toggles the icon and info underneath tabs in the PoolKit Editors.

Show Helpful Notes
This toggles the helpful notes (in yellow boxes) across the PoolKit Editors. This is helpful when you become accustomed to PoolKit and want to streamline the Editors to take up less space.

PoolKit Auto-Spawner Settings

Only Show When Selected
This allows you to always show Spawner gizmos in the Scene view, even if it is not selected.

Scale Gizmos
Icons will be scaled with distance as you move around in the Scene view.

Show Spawn Point Labels
Information about Spawn points will be shown in the Scene view.

Show Name Of Spawner
The name of the spawner is shown in the Scene view.

UI Line Width
This sets how the close the dotted lines appear in the Scene view.

UI Line And Label Color
This sets the color tint used to display dotted lines and some of the descriptive text.

Reset Default Settings

Reset PoolKit Preferences Button
Pushing this button allows you to revert the PoolKit Preferences to the default settings.

PoolKit Workflow And Setup Tips

PoolKit has a seamless and flexible approach to its workflow, allowing you to build pools pretty much anywhere and anyhow. However, here are some recommendations and tips on setting up your Pools:

Pool Types

There are 3 options for Pool Types in PoolKit: “Automatic”, “Fixed Array” and “Dynamic List”. Automatic will pick the best data structure for you based on the features you’ve chosen in your pool but you can also explicitly set which type you’d like to use too.

The Fixed Array is the fastest type of Pool. It uses a fixed built-in array, which is crazy fast but the drawback is that no new instances can be added at runtime (with the exception of Pool Protection if it is turned on). This also means that lazy preloading is incompatible with this Pool Type. These pools are usually best for non-essential items such as particle effects, background objects, and generally things that aren’t game-critical. However, with some planning and when combined with features such as instance recycling, you can actually create critical game objects too! A great example can be found in the PoolKit “Spaceship Demo Scene” where a Spawner fires lazers from the player’s ship.

The Dynamic List type is still very fast but not as fast as the Fixed Array. Its advantage is that it can grow instances in real-time and can use lazy preloading. This is typically best for things like bullets and other objects that you want to make sure are created even if a new instance must be created on the fly.

The trick to fast pools is to foresee and create the number of needed instances before you spawn them. You can use the Pool Statistics tab to help you figure out your usage needs in the Editor. If you can get away with Fixed Arrays, always choose that for the best performance!

Local Pools

Firstly, if you’re new to pooling (or programming altogether), it is recommended to stick to local pools. Local pools are often more flexible and reduces complexity. A local pool only exists in a single scene and not globally throughout the game. In other words, it does not have the “Don’t Destroy On Load” option checked and will be destroyed when changing levels.

If you need to access the same pool across different scenes, you can still do it using a “local” approach. Simply create a prefab of the relevant pool and manually place it in every scene that needs to access it. This will keep things simple and make sure that you don’t accidentally destroy any instances when changing scenes. It’s also easier to debug your games and gives you the most flexibility in the design of your pools on a scene-by-scene basis.

Global Pools

The easiest way to setup a global pool is to create it through the menu using **GameObject > PoolKit > Create Global Pool**. This will create a Pool that is already setup to be global by default (it will not be destroyed when changing scenes).

You can then save the pool's GameObject as a prefab and add it to the 'PoolKit Global Pools' list (found at **GameObject > PoolKit > Setup Global Pools**).

Doing this will automatically create the pool when the game is started (and when pressing play in the Editor, making it really easy to test and debug your games!).

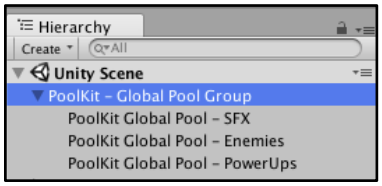
Notes On Setting Up Global Pools

On each prefab of your global pools under the Instances tab, make sure that "Keep Instances Organized" is enabled. This helps instances stay protected when changing scenes.

One thing to note is you should take care NOT to parent any of these instances outside of the pool's own Transform. If you try to change scenes this could cause missing instances and break your pools.

The easiest fix for this is to turn on Pool Protection, which can rebuild your pools in real-time but it comes at a small performance cost. To keep things as fast as possible (without Pool Protection), you will need to take care not to change the parent of these instances and to call the PoolKit.DespawnAllGlobalPools() method when changing scenes.

[Advanced] Using 'Global Pool Groups'



If you need a custom approach to handling global pools, you can do so using 'Global Pool Groups'.

This approach is best suited for those who want to manually handle loading and unloading a group of global pools at specific times (rather than automatically having them load when you start the game).

You can create a global pool within its own Global Pool Group using the GameObject menu: **GameObject > PoolKit > Create Global Pool Group**.

You will notice that this option will create a new pool inside of a 'Global Pool Group' in the Hierarchy pane. This automatically configures the PoolKitSetup component on the parent object and the Pool component on the child object.

You can add more pools to the group by duplicating the pool or by using the menu item again (which will ask if you want to add a new pool to the existing group or create a totally new group alongside it).

Once you've setup the pool group, it's recommended to save it as a prefab so you can easily create it via script or place it in a specific level of your choice.

Pool Planning

Here are some considerations when designing your pools:

- Always stick to Fixed Array Pools if possible. The 'Instance Recycling' feature is your friend here!
- Rather than just having one Dynamic List Pool that can do everything, try having a Fixed Array Pool AND a Dynamic List pool. That way you can still benefit from the instances that don't need dynamic resizing.
- Even though you can create as many pools as you want, there is some overhead in running them. Try to minimize the number of pools you're using.
- Don't waste memory by making your pools too large. Use the Pool's statistics screen to help figure out the right size of each prefab.
- Only enable features on Pools and Prefabs if you're actually using it.
- It's generally not a good idea to use Automatic-Despawning as well as a Despawner component on a prefab. Use one or the other.

Example Of The Fastest Pool

Below is an overview of the settings to create the fastest type of PoolKit pool:

Pool

Pool Type:	*Fixed Array*
Enable Pool Protection:	*false*
Don't Destroy On Load:	*false*
Enable Delegates & Events:	*false*

Prefab

Pool Size Options:	*Keep Pool Size Fixed*
Instance Scale:	*Ignore*
Instance Layer:	*Ignore*
Keep Instances Organized:	*Ignore*
Recycle Spawned Instances:	*false*
Lazy Preloading:	*false*
Automatic Despawning:	*false*
Notification Mode:	*None*
Enable Instantiation Events:	*false*
Enable Destroy Events:	*false*

PoolKit API

PoolKit has methods and events that are available via the API. Remember to add "**using HellTap.PoolKit;**" in C# scripts or "**import HellTap.PoolKit;**" in Unityscripts.

PoolKit API

The PoolKit Static class has methods and delegates that are always available. Some methods have alternate names so you can choose the style you prefer!

Caching Pools

PoolKit.GetPool(string poolName) : Returns Pool
The simplest way to find and cache a pool is by finding it by name:
NOTE: You can also use alternate method names such as: PoolKit.Get(), PoolKit.Find() and PoolKit.FindPool()

```
// Cache a Pool by Name
Pool myPool = PoolKit.GetPool("MyPool");
```

PoolKit.GetPoolContainingPrefab(GameObject prefabGameObject) : Returns Pool
PoolKit.GetPoolContainingPrefab(Transform prefabTransform) : Returns Pool
PoolKit.GetPoolContainingPrefab(string prefabName) : Returns Pool

You can also find a Pool by searching for a prefab it controls, like this:
NOTE: You can also use: FindPoolContainingPrefab()

```
// Cache a pool by finding a specific prefab it controls (by GameObject)
Pool myPool = PoolKit.GetPoolContainingPrefab(prefabGameObject);

// Cache a pool by finding a specific prefab it controls (by Transform)
Pool myPool = PoolKit.GetPoolContainingPrefab(prefabTransform);

// Cache a pool by finding a specific prefab it controls (by Name)
Pool myPool = PoolKit.GetPoolContainingPrefab("My Prefab Name");
```

PoolKit.GetPoolContainingInstance(GameObject instanceGameObject) : Returns Pool
PoolKit.GetPoolContainingInstance(Transform instanceTransform) : Returns Pool

Another way of finding a pool is by searching for an instance it has already created. Please note that this is quite a complex method and should generally only be used when caching, like this:
NOTE: You can also use: FindPoolContainingInstance()

```
// Cache a pool by finding a specific instance it controls (by GameObject)
Pool myPool = PoolKit.GetPoolContainingInstance(instanceGameObject);

// Cache a pool by finding a specific instance it controls (by Transform)
Pool myPool = PoolKit.GetPoolContainingInstance(instanceTransform);
```

Caching Spawners

PoolKit.GetSpawner(string spawnerName) : Returns Spawner
You can easily find and cache a spawner by finding it by name:
NOTE: You can also use: FindSpawner()

```
// Cache a Spawner by name
Spawner mySpawner = PoolKit.GetSpawner("MySpawner");
```

Does A Pool Exist?

PoolKit.PoolExists() : Returns bool
You can easily find out if a pool exists by using this method:

```
// Does a Pool Exist? (Enter the name of the pool to find)
if( PoolKit.PoolExists("MyPool") ){
    // Do something if this pool exists!
}
```

Despawn All Pools

PoolKit.DespawnAll()
PoolKit.DespawnAllLocalPools()
PoolKit.DespawnAllGlobalPools()
You can despawn all pools, all local pools or all global pools with a single command like this:
NOTE: You can also use: DespawnAllLocal() and DespawnAllGlobal()

```
// Despawn all pools and instances
PoolKit.DespawnAll();
```

Destroying Pools

PoolKit.DestroyPool()
Other than destroying their GameObjects directly, you can easily destroy a specific pool by name:
NOTE: You can also use: Remove() and RemovePool()

```
// Destroy a pool by name
PoolKit.DestroyPool("MyPool");
```

PoolKit.DestroyAllPools()
PoolKit.DestroyAllLocalPools()
PoolKit.DestroyAllGlobalPools()
You can destroy all pools, all local pools or all global pools with a single command like this:
NOTE: You can also use: RemoveAll(), RemoveAllPools(), DestroyAll(), RemoveAllLocal(), RemoveAllLocalPools(), DestroyAllLocal(), RemoveAllGlobal(), RemoveAllGlobalPools(), and DestroyAllGlobal()

```
// Destroy all pools
PoolKit.DestroyAllPools();
```

Creating A New Pool

PoolKit.CreatePool (string poolName, Pool.PoolType poolType, bool enablePoolProtection, bool allowDelegatesAndEvents, bool dontDestroyOnLoad, PoolItem[] poolItems, GameObject usingGameObject = null) : Returns Pool.
To create a new pool at runtime you must create and setup at least 1 **PoolItem** (the equivalent of one of the prefabs in the Pool's Prefab tab). You can then add it to the CreatePool method:
NOTE: You can also use: Add(), AddPool() and Create()

```
// Firstly, Create at least 1 Pool Item to add to the pool
PoolItem myPI = new PoolItem();
myPI.prefabToPool = myPrefab;        // The prefab we're going to pool (GameObject)
myPI.poolSize = 10;                  // The size of the pool at start

// NOTE: Look at the PoolItem.cs script to understand the class in more detail!
// Everything you can do in the editor is possible with scripting.

// Add Pool Item to an entirely new pool
Pool myNewPool = PoolKit.CreatePool (
    "My New Pool",                   // Name of the new pool
    Pool.PoolType.Automatic,         // The type of the new pool
    true,                            // Should we enable Pool Protection?
    true,                            // Should we allow delegates and Events on this pool?
    false,                           // Don't destroy on Load
    new PoolItem[]{ myPI },          // A list of Pool Items ( like the prefab tab )
    null                             // GameObject to add Pool component. Null = new
);
```

Overriding How PoolKit Instantiates And Destroys Instances

// Global Override Events	// Local Override Events
PoolKit.OnCreateInstance	**Pool.OnCreateInstance**
PoolKit.OnDestroyInstance	**Pool.OnDestroyInstance**

If you wish to globally override the way PoolKit instantiates and Destroys instances, this can be achieved by subscribing to the PoolKit **InstantiatePrefab** and **DestroyInstance** events, which will act as an override. You can also do this locally at the Pool level using the same approach.

It is important to note that only prefabs that have had their '**Enable Instantiation Events**' and '**Enable Destroy Events**' checkboxes turned on in the pool can use this feature (otherwise delegates would continuously be checked needlessly which would diminish performance).

You can set this up easily in the Editor. Click the relevant pool and select the 'Prefabs' tab. Find the prefab you want to override and click into the 'Advanced' tab to find the override checkboxes.

Alternatively, you can forcibly turn on the checkboxes at runtime with scripting. In the example below, we attempt to override the way a specific prefab will be instantiated and destroyed. In the first part of OnEnable, we find the pool containing the prefab we're interested in and then we request the **PoolItem** within the Pool that manages the prefab (the **PoolItem** is essentially what you see in the Prefabs tab). From there, we have access to the checkboxes and can force them on. If the prefab already has the checkboxes set in the Editor, this section of the code isn't needed.

We can then subscribe to the Events using the template below.

```
public GameObject prefabToOverride;  // This is the prefab we want to take over
Pool _findPool;                      // This is a helper variable to cache the pool

void OnEnable(){

    // OPTIONAL: Find the PoolItem containing the prefab we want to override, and make sure
    // it is turned on - Useful if we've forgotten to enable it in the Pool's inspector!

    // Find the pool containing the prefab we want to override ...
    if(_findPool==null){ _findPool = PoolKit.GetPoolContainingPrefab( prefabToOverride ); }

    // Make sure the instantiate and destroy delegates are enabled for the prefab (PoolItem)
    if( findPool != null ){
        PoolItem pi = _findPool.GetPoolItem( prefabToOverride );
        if(pi!=null){
            pi.enableInstantiateDelegates = true;
            pi.enableDestroyDelegates = true;
        }
    }

    // Subsribe to the override events
    PoolKit.OnCreateInstance += OnCreateInstance;
    PoolKit.OnDestroyInstance += OnDestroyInstance;
}

void OnDisable(){

    // Unsubsribe from the override events
    PoolKit.OnCreateInstance -= OnCreateInstance;
    PoolKit.OnDestroyInstance -= OnDestroyInstance;
}

// Event called when we must instantiate a new instance
GameObject OnCreateInstance( GameObject prefab ){

    // NOTE: Make sure to return the created GameObject!
    if ( prefab != null ){ return Instantiate( prefab ); }

    // Return null if something goes wrong
    return null;
}

// Event called when we must destroy an instance
void OnDestroyInstance( GameObject instance ){

    // Make sure to destroy the instance
    if ( instance != null ){ Destroy(instance);}
}
```

Pool API

Once you have access to a Pool, you can use a variety of methods and events.

Spawn An Instance By Name

The following methods return the instance's Transform.

Pool.Spawn(string prefabName) : Returns Transform
the fastest way to spawn an instance by name is to request its prefab name. The instance's position, rotation, scale and parent will all be handled using the Pool's default settings.

```
// Spawn an instance of the prefab named "MyPrefab"
Transform myInstance = pool.Spawn("MyPrefab");
```

Alternate versions are available, allowing you specifically set positions, rotations and parents.

Pool.Spawn(string prefabName, Vector3 position, Vector3 eulerRotation, Transform parent = null) : Returns Transform

```
// Spawn instance of "MyPrefab" at the origin, using Vector3.zero rotation with default parent
Transform myInstance = pool.Spawn("MyPrefab", Vector3.zero, Vector3.zero, null);
```

Pool.Spawn(string prefabName, Vector3 position, Quaternion rotation, Vector3 localScale, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, with no rotation, default scale with default parent
Transform myInstance = pool.Spawn("MyPrefab", Vector3.zero, Quaternion.identity, Vector3.one,
null);
```

Pool.Spawn(string prefabName, Vector3 position, Vector3 eulerRotation, Vector3 localScale, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, Vector3.zero rotation, default scale with default parent
Transform myInstance = pool.Spawn("MyPrefab", Vector3.zero, Vector3.zero, Vector3.one, null);
```

Pool.Spawn(string prefabName, Vector3 position, Quaternion rotation, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, without rotation with default parent
Transform myInstance = pool.Spawn("MyPrefab", Vector3.zero, Quaternion.identity, null);
```

NOTE: To return the instance as a GameObject instead of a Transform, replace the **Spawn** method with **SpawnGO**. For example:

```
// Spawn an instance of the prefab named "MyPrefab" and return its GameObject
GameObject myInstance = pool.SpawnGO("MyPrefab");
```

Spawn An Instance By GameObject / Transform

The following methods can be used to spawn an instance of a prefab. You can use either the prefab's **GameObject** or **Transform** reference as the first parameter.

Pool.Spawn(GameObject / Transform prefab) : Returns Transform
the fastest way to spawn an instance is by passing a reference to its prefab. The instance's position, rotation, scale and parent will all be handled using the Pool's default settings.

```
// Spawn an instance of the prefab variable "MyPrefab"
Transform myInstance = pool.Spawn(MyPrefab);
```

Alternate versions are available, allowing you specifically set positions, rotations and parents.

Pool.Spawn(GameObject / Transform prefab, Vector3 position, Vector3 eulerRotation, Transform parent = null)
: Returns Transform

```
// Spawn instance at the origin, using Vector3.zero rotation with default parent
Transform myInstance = pool.Spawn(MyPrefab, Vector3.zero, Vector3.zero, null);
```

Pool.Spawn(GameObject / Transform prefab, Vector3 position, Quaternion rotation, Vector3 localScale, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, with no rotation, default scale with default parent
Transform myInstance = pool.Spawn(MyPrefab, Vector3.zero, Quaternion.identity, Vector3.one,
null);
```

Pool.Spawn(GameObject / Transform prefab, Vector3 position, Vector3 eulerRotation, Vector3 localScale, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, Vector3.zero rotation, default scale with default parent
Transform myInstance = pool.Spawn(MyPrefab, Vector3.zero, Vector3.zero, Vector3.one, null);
```

Pool.Spawn(GameObject / Transform prefab, Vector3 position, Quaternion rotation, Transform parent = null) : Returns Transform

```
// Spawn instance at the origin, without rotation with default parent
Transform myInstance = pool.Spawn(MyPrefab, Vector3.zero, Quaternion.identity, null);
```

NOTE: To return the instance as a GameObject instead of a Transform, replace the **Spawn** method with **SpawnGO**. For example:

```
// Spawn an instance of the prefab variable "MyPrefab" and return it's GameObject
GameObject myInstance = pool.SpawnGO(MyPrefab);
```

Despawn A Specific Instance By GameObject / Transform

Pool.Despawn(GameObject / Transform instance) : Returns bool (true = success, false = failed)
the fastest way to despawn an instance is to directly call the Despawn method on the pool that is controlling it.

```
// Tell myPool to despawn this GameObject
myPool.Despawn(gameObject);
```

Despawn All Instances In A Pool

Pool.DespawnAll() : void
Pool.DespawnAll(GameObject / Transform prefab) : void
You can despawn all instances in a pool with a single command. Alternatively you can despawn all instances of a specific prefab.

```
// Tell myPool to despawn all instances of the prefab "myPrefab"
myPool.DespawnAll(myPrefab);

// Tell myPool to despawn everything
myPool.DespawnAll();
```

Get Pool Item (container for prefabs)

Pool.GetPoolItem(GameObject prefab) : Returns PoolItem
If you need to get the PoolItem of a prefab within a pool, you do so like this:

```
// Get the PoolItem from the pool "myPool" that manages "myPrefab".
myPool.GetPoolItem(myPrefab);
```

Get Instance Counts

Pool.GetInstanceCount() : Returns int
Get the total of all spawned and despawned instances in a pool

Pool.GetInstanceCount(GameObject / Transform prefab) : Returns int
Get the total of all spawned and despawned instances of a specific prefab in a pool.

Pool.GetInstanceCount(string prefabName) : Returns int
Get the total of all spawned and despawned instances of a specific prefab in a pool by name.

Pool.GetActiveInstanceCount() : Returns int
Get the total of all spawned instances in a pool

Pool.GetActiveInstanceCount(GameObject / Transform prefab) : Returns int
Get the total of all spawned instances of a specific prefab in a pool.

Pool.GetActiveInstanceCount(string prefabName) : Returns int
Get the total of all spawned instances of a specific prefab in a pool by name.

Pool.GetInactiveInstanceCount() : Returns int
Get the total of all despawned instances in a pool

Pool.GetInactiveInstanceCount(GameObject / Transform prefab) : Returns int
Get the total of all despawned instances of a specific prefab in a pool.

Pool.GetInactiveInstanceCount(string prefabName) : Returns int
Get the total of all despawned instances of a specific prefab in a pool by name.

Get An Array Of All Instances

Pool.GetPoolKitInstances() : Returns PoolKitInstance[]
This will create and return a safe copy of all managed instances in the pool. This is a very complicated operation and should be used sparingly. PoolKitInstances is a class used to manage the state of all instances managed by a Pool. You will likely prefer GetInstances() instead.

```
// Get a copy of all the PoolKitInstance classes from myPool
PoolKitInstance[] instances = myPool.GetPoolKitInstances();
```

Pool.GetInstances() : Returns Transform[]
This will create and return an array of all instances in the pool by caching their Transforms. This is a very complicated operation and should be used sparingly.

```
// Get a copy of all instance Transforms in myPool
Transform[] instances = myPool.GetInstances();
```

Pool.GetInstances(GameObject / Transform prefab) : Returns Transform[]
This will create and return an array of all instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by passing a reference of either it's GameObject or Transform.

```
// Get a copy of all instance Transforms of myPrefab in myPool
Transform[] myPrefabInstances = myPool.GetInstances(myPrefab);
```

Pool.GetInstances(string prefabName) : Returns Transform[]
This will create and return an array of all instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by name.

```
// Get a copy of all instance Transforms of "myPrefab" in myPool
Transform[] myPrefabInstances = myPool.GetInstances("MyPrefab");
```

Get An Array Of All Spawned Instances

Pool.GetActiveInstances() : Returns Transform[]
This will create and return an array of all active instances in the pool by caching their Transforms. This is a very complicated operation and should be used sparingly.

```
// Get a copy of all spawned instance Transforms in myPool
Transform[] instances = myPool.GetActiveInstances();
```

Pool.GetActiveInstances(GameObject / Transform prefab) : Returns Transform[]
This will create and return an array of all active instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by passing a reference of either it's GameObject or Transform.

```
// Get a copy of all spawned instance Transforms of myPrefab in myPool
Transform[] myPrefabInstances = myPool.GetActiveInstances(myPrefab);
```

Pool.GetActiveInstances(string prefabName) : Returns Transform[]
This will create and return an array of all active instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by name.

```
// Get a copy of all spawned instance Transforms of "myPrefab" in myPool
Transform[] myPrefabInstances = myPool.GetActiveInstances("MyPrefab");
```

Get An Array Of All Despawned Instances

Pool.GetInactiveInstances() : Returns Transform[]
This will create and return an array of all inactive instances in the pool by caching their Transforms. This is a very complicated operation and should be used sparingly.

```
// Get a copy of all despawned instance Transforms in myPool
Transform[] instances = myPool.GetInactiveInstances();
```

Pool.GetInactiveInstances(GameObject / Transform prefab) : Returns Transform[]
This will create and return an array of all inactive instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by passing a reference of either it's GameObject or Transform.

```
// Get a copy of all despawned instance Transforms of myPrefab in myPool
Transform[] myPrefabInstances = myPool.GetInactiveInstances(myPrefab);
```

Pool.GetInactiveInstances(string prefabName) : Returns Transform[]
This will create and return an array of all inactive instances of a prefab by caching their Transforms. This is a very complicated operation and should be used sparingly. The prefab is found by name.

```
// Get a copy of all despawned instance Transforms of "myPrefab" in myPool
Transform[] myPrefabInstances = myPool.GetInactiveInstances("MyPrefab");
```

Helper Methods

Pool.HasActiveInstances() : Returns bool
If there are any spawned instances in a pool, this will return true.

Pool.HasActiveInstances(GameObject / Transform prefab) : Returns bool
If there are any spawned instances of a specific prefab in a pool, this will return true.

Pool.HasActiveInstances(string prefabName) : Returns bool
If there are any spawned instances of a specific prefab in a pool (by name), this will return true.

Pool.HasInactiveInstances() : Returns bool
If there are any despawned instances in a pool, this will return true.

Pool.HasInactiveInstances(GameObject / Transform prefab) : Returns bool
If there are any despawned instances of a specific prefab in a pool, this will return true.

Pool.HasInactiveInstances(string prefabName) : Returns bool
If there are any despawned instances of a specific prefab in a pool (by name), this will return true.

Add A New Prefab To An Existing Pool

Pool.Add(PoolItem poolItem) = Returns bool (true = successful, false = failed)
To add a new prefab to an existing pool, you must first setup a new **PoolItem** class to configure the settings of the prefab within the pool. Then you can use the Pool.Add() method like this:

```
// Firstly, Create a Pool Item to add to the pool
PoolItem myPI = new PoolItem();       // Create a new PoolItem
myPI.prefabToPool = myPrefab;         // Add the prefab we're going to pool (GameObject)
myPI.poolSize = 10;                   // The size of the pool at start

// Add the PoolItem to an existing pool
myExistingPool.Add(myPI);
```

More Information About Pool Items

You may have noticed that a PoolItem is required before creating new pools or adding new prefabs to existing pools.

The code snippets above do not go into full detail about the options available in a Pool Item. However, every single variable is fully documented in PoolItem.cs, which can be found in the PoolKit plugins folder. The core class variables are the basis of the very same settings found in the **Pool > Prefabs** tab. It is highly recommended to look through that script file and the *Pool Delegate Examples* scene found in the *PoolKit Demos & Extras* folder.

Subscribe To Events When Instances Are Spawned And Despawned

// Events
Pool.onPoolSpawn
Pool.onPoolDespawn

If you wish to subscribe to Events that trigger when an instance is spawned and despawned, you can do it using an approach such as the example below. Please note that you must have '**Enable Pool Events**' enabled on the pool for this to work.

The code also allows you to turn on Pool Events dynamically, but if you have already set '**Enable Pool Events**' in the Pool's inspector, that part of the code can be safely omitted.

```
public string poolName = "MyPool";    // Enter the name of the pool to cache it!
Pool findPool;                        // Helper variable to store the pool

void OnEnable(){

    // Find the pool (if we haven't already)
    if(findPool==null){ findPool = PoolKit.Get( poolName ); }

    // If we have found the pool, subscribe!
    if(findPool!=null ){

        // < Optional >
        // Turn on Pool Events if you've forgotten to set it in the Pool's inspector!
        findPool.enablePoolEvents = true;

        // Subscribe to the Pool Events
        findPool.onPoolSpawn += onPoolSpawn;
        findPool.onPoolDespawn += onPoolDespawn;

    }
}

void OnDisable(){

    // If we already found the pool, unsubscribe!
    if(findPool!=null){

        // Unsubscribe from the Pool Events
        findPool.onPoolSpawn -= onPoolSpawn;
        findPool.onPoolDespawn -= onPoolDespawn;
    }
}

// Event called when the pool spawns an object
void onPoolSpawn( Transform instance, Pool pool ) {
    Debug.Log("EXAMPLE: The Pool " + pool.poolName + " just spawned: " + instance.name );
}

// Event called when the pool despawns an object
void onPoolDespawn( Transform instance, Pool pool ) {
    Debug.Log("EXAMPLE: The Pool " + pool.poolName + " just despawned: " + instance.name );
}
```

Spawner API

Once you have access to a Spawner, you can use a variety of methods and events.

Core Actions

Spawner.Stop() : void
Stops the spawner.

Spawner.Play() : void
Starts the spawner.

Spawner.RestartAndPlay() : void
Restarts the spawner and starts it.

Spawner.Pause() : void
Pauses the spawner.

Spawner.Resume() : void
Resumes the spawner.

Helper Methods

Spawner.CanSpawn() : Returns bool
Returns true if the spawner is ready to create new instances.

Spawner.AddPrefabToSpawner(GameObject prefab, float randomWeight = 100f) : void
Allows you to add a new prefab to the spawner at runtime and define it's random chance weight.

Spawner.RemovePrefabFromSpawner(GameObject prefab) : void
Allows you to remove an existing prefab from the spawner at runtime.

Spawner.ReplacePrefabInSpawner(GameObject oldPrefab, GameObject newPrefab) : void
Allows you to replace an existing prefab with a new one.

Spawner.SetRandomWeightOfPrefab(GameObject prefab, float newWeight) : void
Allows you to update a prefab's random chance weight.

Spawner.SetRandomWeightOfPrefab(int arrayIndex, float newWeight) : void
Allows you to update a prefab's random chance weight by finding the prefab using its array index

Spawner.SetRandomWeightOfVector3Position(int arrayIndex, float newWeight) : void
Allows you to update a Vector3 Spawn Point's random chance weight by finding it using its array index

Spawner.SetRandomWeightOfTransformPosition(int arrayIndex, float newWeight) : void
Allows you to update a Transform Spawn Point's random chance weight by finding it using its array index

Spawner.SetSpawnPointPosition (int arrayIndex, Vector3 newPosition) : void
Allows you to update the position of a Vector3 Spawn Point using its array index.

Spawn Methods

Spawner.Spawn(GameObject useThisPrefab = null) : void
Allows you to force the Spawner to spawn another instance. You can also specifically tell it which prefab to use (provided it is already setup in the Spawner).

Subscribe To Spawner Events

// Events
Spawner.onSpawnerSpawn< Transform instance >
Spawner.onSpawnerStart
Spawner.onSpawnerStop
Spawner.onSpawnerPause
Spawner.onSpawnerResume
Spawner.onSpawnerEnd

You can easily subscribe to a Spawner's Events to get access to when a Spawner's state changes and even to receive spawned instances (use this sparingly, especially with fast spawners).

Note that a Spawner must have "**Enable Delegates & Events**" enabled in its Events tab or you have to set it in within the script. Here's an example showing how to subscribe to all events:

```
public string spawnerName = "SPAWNER";      // Enter the name of the Spawner to cache it
Spawner spawner = null;                     // Helper variable to store the spawner

void OnEnable(){

    // Cache the Spawner
    if(spawner==null){ spawner = PoolKit.GetSpawner( spawnerName ); }

    // Subscribe to the Spawner's events
    if( spawner != null ){

        // <Optional> Force the Spawner to allow spawner events
        spawner.enableSpawnerEvents = true;

        // Subscribe
        spawner.onSpawnerSpawn += onSpawnerSpawn;
        spawner.onSpawnerStart += onSpawnerStart;
        spawner.onSpawnerStop += onSpawnerStop;
        spawner.onSpawnerPause += onSpawnerPause;
        spawner.onSpawnerResume += onSpawnerResume;
        spawner.onSpawnerEnd += onSpawnerEnd;
    }
}

void OnDisable(){

    // Unsubscribe from the Spawner's events
    if( spawner != null ){

        spawner.onSpawnerSpawn -= onSpawnerSpawn;
        spawner.onSpawnerStart -= onSpawnerStart;
        spawner.onSpawnerStop -= onSpawnerStop;
        spawner.onSpawnerPause -= onSpawnerPause;
        spawner.onSpawnerResume -= onSpawnerResume;
        spawner.onSpawnerEnd -= onSpawnerEnd;
    }
}


// SPAWNER JUST SPAWNED AN INSTANCE
void onSpawnerSpawn( Transform theInstance ){
    // You should always check to make sure the instance is not null
    if(theInstance!=null){ Debug.Log("EXAMPLE: " + spawnerName + " Just Spawned " +
theInstance.name ); }
}

// SPAWNER JUST STARTED
void onSpawnerStart(){ Debug.Log("EXAMPLE: " + spawnerName + " has Started!" );}

// SPAWNER JUST STOPPED
void onSpawnerStop(){ Debug.Log("EXAMPLE: " + spawnerName + " has Stopped!" ); }

// SPAWNER JUST PAUSED
void onSpawnerPause(){ Debug.Log("EXAMPLE: " + spawnerName + " has paused!" ); }

// SPAWNER JUST RESUMED
void onSpawnerResume(){ Debug.Log("EXAMPLE: " + spawnerName + " has resumed!" ); }

// SPAWNER JUST ENDED
void onSpawnerEnd(){ Debug.Log("EXAMPLE: " + spawnerName + " has ended!" ); }
```

Despawner API

Once you have access to a Despawner, you can use a variety of methods and events.

Despawn Actions

Despawner.Despawn(bool triggeredByPhysicsEvent = false) : void
Despawn the instance immediately. You can optionally choose to simulate a physics event, which has an impact on which GameObjects are chain-spawned.

Despawner.Despawn(float despawnCountdown) : void
Despawn the instance after a countdown.

Despawner.Despawn(float minDespawnCountdown, float maxDespawnCountdown) : void
Despawn the instance after a random range countdown.

Subscribe To Despawner Events

// Events
Despawner.onDespawnerDespawn
Despawner.onDespawnerCollided< GameObject go >
Despawner.onDespawnerChainSpawn< Transform instance >

You can easily subscribe to the events of a Despawner and get notifications. You'll be notified when it is about to despawn, what it has collided with and even what instances are being chain spawned.

Note that a Despawner must have "**Enable Delegates & Events**" enabled in its Events tab or you have to set it in within the script. Here's an example showing how to subscribe to all events:

```
public Despawner despawner = null;    // <- Set despawner here

void OnEnable(){

    // Subscribe to the Despawner's events
    if( despawner != null ){
        despawner.enableDespawnerEvents = true; // <Optional> Force despawner to use events

        // Subscribe
        despawner.onDespawnerDespawn += onDespawnerDespawn;
        despawner.onDespawnerCollided += onDespawnerCollided;
        despawner.onDespawnerChainSpawn += onDespawnerChainSpawn;
    }
}

void OnDisable(){

    // Unsubscribe from the Despawner's events
    if( despawner != null ){
        despawner.onDespawnerDespawn -= onDespawnerDespawn;
        despawner.onDespawnerCollided -= onDespawnerCollided;
        despawner.onDespawnerChainSpawn -= onDespawnerChainSpawn;
    }
}

// DESPAWNER JUST DESPAWNED
void onDespawnerDespawn(){ Debug.Log( despawner.gameObject.name + " has despawned!" ); }

// DESPAWNER JUST COLLIDED WITH A GAMEOBJECT
void onDespawnerChainSpawn( GameObject go ){
    if(go!=null){ Debug.Log( despawner.gameObject.name + " Collided With " + go.name ); }
}

// DESPAWNER JUST CHAIN-SPAWNED AN INSTANCE
void onDespawnerChainSpawn( Transform inst ){
    if(inst!=null){ Debug.Log( despawner.gameObject.name + " Chain-Spawned " + inst.name ); }
}
```

PoolKit Setup API

You can also use scripting to change the same options available in the PoolKitSetup component.

PoolKit.renameObjectsInPool = PoolKit.RenameFormat.NoRenaming;
You can setup how PoolKit renames instances when they are instantiated. Not renaming them is the fastest way of dealing with this but just like in the inspector, the following options are available:

```
enum PoolKit.RenameFormat {

    EasyToReadObjectNameWithPoolKitAndIndex,
    EasyToReadObjectNameWithIndex,
    ObjectNameWithPoolKitAndIndex,
    ObjectNameWithIndex,
    NoRenaming

}
```

PoolKit.onlyRenameObjectsInEditor = true;
Tell PoolKit whether renaming should happen in builds or just in the Editor.

PoolKit.debugPoolKit = false;
Tell PoolKit whether to show debug messages in the console. Please note that debug messages causes memory allocations!

Support

If you need any assistance or have suggestions for this plugin, feel free to visit our website at:

www.unitygamesdevelopment.co.uk

I hope you find this system useful, as I have in my own personal projects! =)

All the best!

- Mel