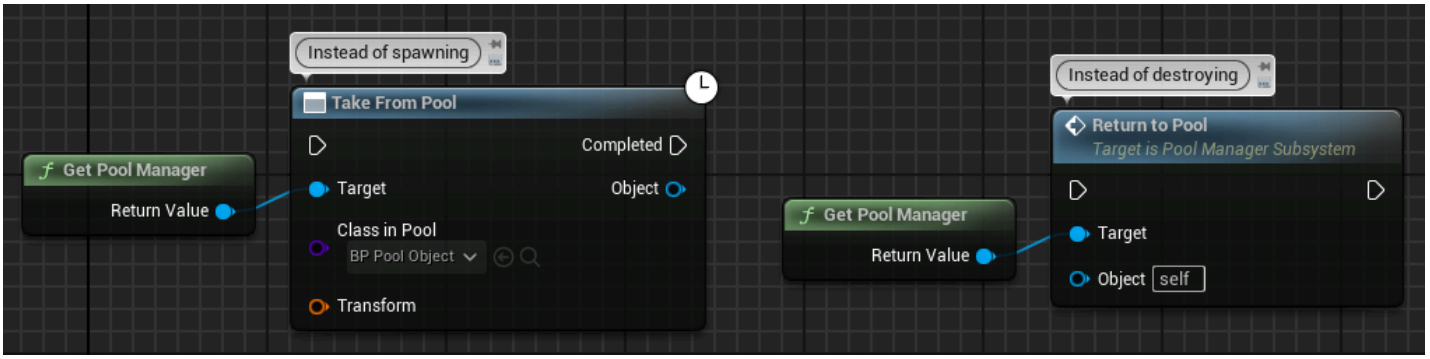
Pool Manager documentation

[GitHub page]

Introduction

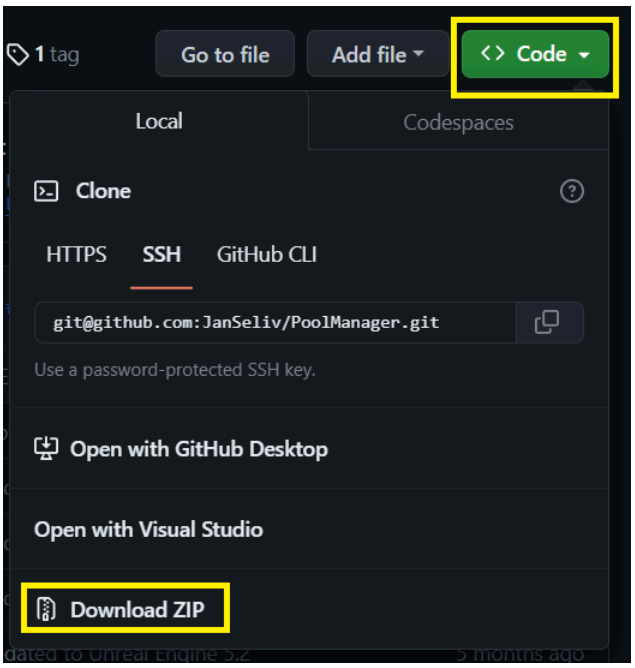
The Pool Manager helps reuse objects that show up often, instead of creating and destroying them each time. Creating and destroying objects, like projectiles or explosions, can be slow and cause issues such as making the game slow or laggy when done frequently. The Pool Manager alleviates these problems by maintaining a pool of objects. Instead of creating and destroying objects all the time, the Pool Manager keeps these objects for reuse. This strategy improves the smoothness of the game.



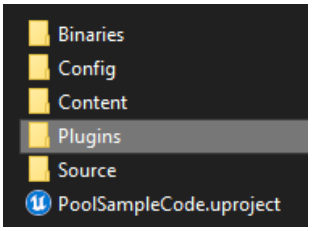
Getting Started

Install the plugin in 4 steps

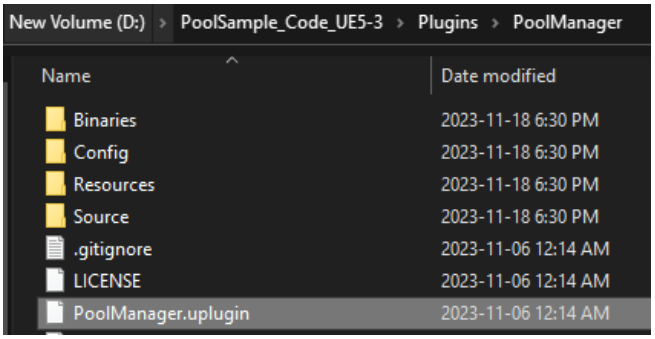
Step 1.
Download the plugin: https://github.com/JanSeliv/PoolManager



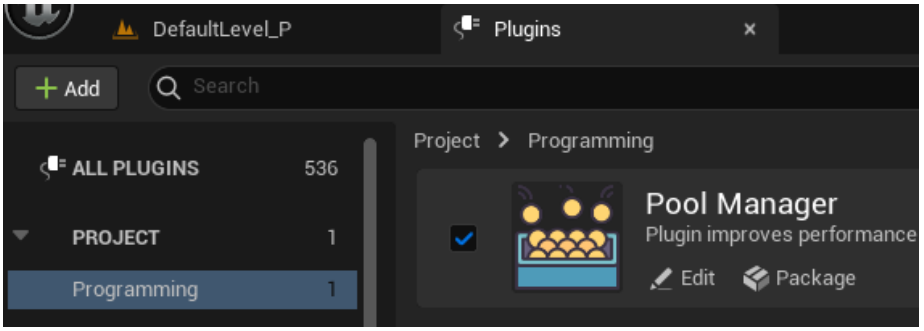
Step 2.
Create in your game project ‘Plugins’ folder and inside ‘PoolManager’ one



Step 3.
Extract downloaded files into Plugins -> PoolManager: `PoolManager.uplugin` has to be located under next path:
YourGame\Plugins\PoolManager\PoolManager.uplugin



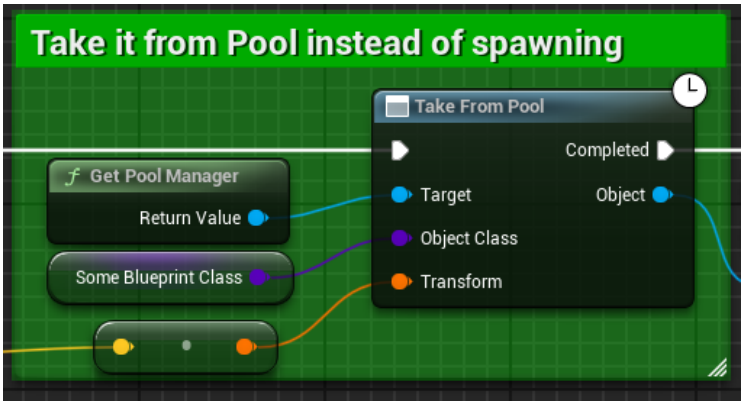
Step 4.
Open your project,'Edit' -> 'Plugins' window to make sure the 'Pool Manager' plugin is enabled for your project:



Setup the Pool Manager in 2 steps

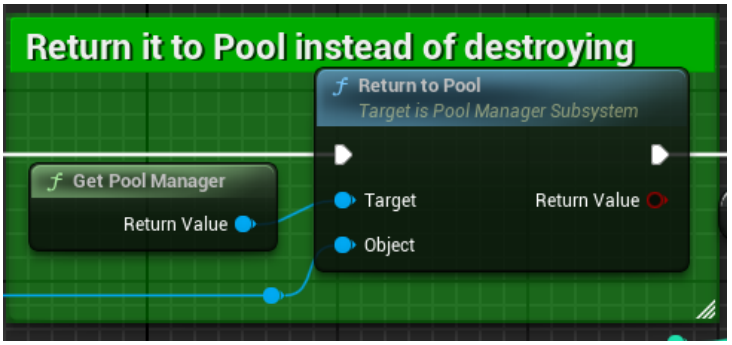
The Pool Manager is easy to use and doesn't need any in-engine setup. Just do these two simple steps to start using its powerful features.

Step 1.
Call **'Take From Pool**' whenever you need any object (*or `Take From Pool Array` for multiple objects*):



See more detailed 'Take From Pool' section or view it in action in the Sample Project.

Step 2.
Call `**Return To Pool** whenever object is not needed anymore:



See more detailed in 'Return To Pool' section or view it in action in the Sample Project.

Public Functions

For a full list of all available functions, see `PoolManagerSubsystem.h`. All functions are well-commented and should be clear even to those who do not understand the code.

See functions in action in the Sample Project.

Take From Pool

The **Take From Pool** function efficiently retrieves objects from a pool based on the specified class. This function is designed to operate asynchronously, meaning it returns the object when it is ready, without blocking other processes.

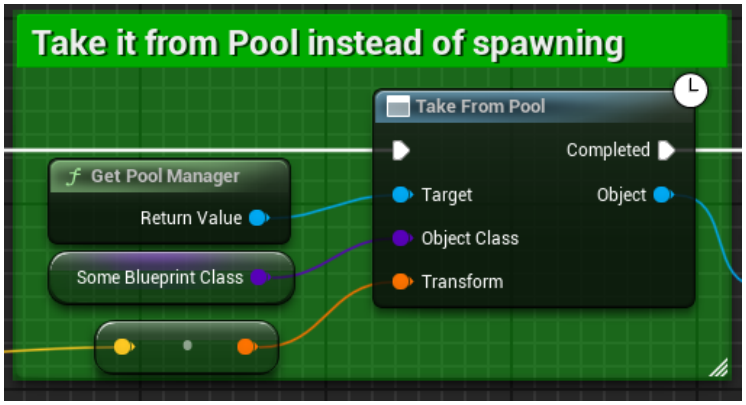
Key Features
- If a free object is found in the pool, it is activated and returned immediately.
- If no free object is available, a new one is spawned asynchronously in the following frames and then registered in the pool.
- **In code**, all functions additionally return a Handle: hash (ID) associated with the object (but not the object itself). `*ReturnToPool*` by Handle is more reliable as it handles even the case when the object is not spawned yet:

```
FPoolObjectHandle Handle = UPoolManagerSubsystem::Get().TakeFromPool(...);
```

Warnings and Notes:
- The speed at which new objects are created is influenced by the *'SpawnObjectsPerFrame*' setting, which can be adjusted in *'Project Settings' -> "Plugins" -> "Pool Manager".*
- use `**TakeFromPoolArray**` instead of requesting one by one in for/while: 'Completed' output does not work in loops.

Examples:

1. In blueprints:



2. Another blueprint example: look at Blueprint Sample: widgets, where `**WBP_Canvas**` pools randomly up to 5 of `*WBP_Text*` subwidgets numerating them.

3. In code without callback:

```
UPoolManagerSubsystem::Get().ReturnToPool(MyObjectPtr);
```

4. In code with callback:

```
const FOnSpawnCallback OnCompleted = [](const FPoolObjectData& CreatedObject)
{
   CreatedObject.GetChecked<AMyActor>().OnMyActorTakenFromPool();
};
UPoolManagerSubsystem::Get().TakeFromPool(SomeBPClass, FTransform::Identity, OnCompleted);
```

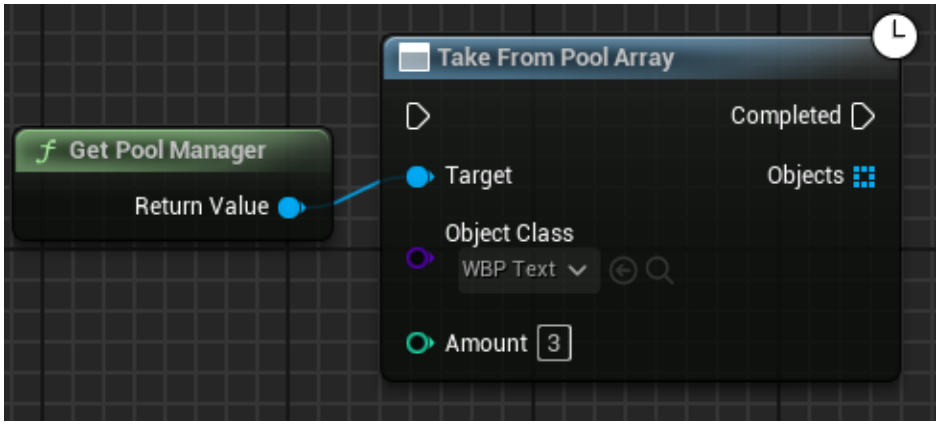
5. Another code example: GeneratedMap.cpp in `SpawnActorByType` function.

Take From Pool Array

The **Take From Pool Array** functions extend the capabilities of the individual object retrieval method by allowing multiple objects to be requested simultaneously.

Examples:

1. In blueprints:



2. Code example to take objects of the same class:

```
const FOnSpawnAllCallback OnCompleted = [](const TArray<FPoolObjectData>& CreatedObjects)
{
   for (const FPoolObjectData& It : CreatedObjects)
   {
      It.GetChecked<AMyActor>().OnMyActorTakenFromPool();
   }
};

TArray<FPoolObjectHandle> OutHandles;
UPoolManagerSubsystem::Get().TakeFromPoolArray(OutHandles, SomeBPClass,/*Amount*/5, OnCompleted);
```

3. Another code example of taking the same class: look at Code Sample: Draw Boxes, where **`*SomeGameplayClass*`** infinitely pools randomly up to 50 Draw Boxes at once each 0.2 seconds in random location on the level.

4. Code example to take objects from different classes:

```
TArray<FSpawnRequest> InRequests;
InRequests.Emplace(SomeBPClass);
InRequests.Emplace(MyWidgetClass);
InRequests.Emplace(MyGeometryClass);

FOnSpawnAllCallback OnCompleted = [](const TArray<FPoolObjectData>& CreatedObjects)
{
   for (const FPoolObjectData& It: CreatedObjects)
   {
      // ...
   }
};
TArray<FPoolObjectHandle> OutHandles;
UPoolManagerSubsystem::Get().TakeFromPoolArray(OutHandles, InRequests, OnCompleted);
```

5. Another code example of taking different classes: GeneratedMap.cpp in `SpawnActorsByTypes` function. It forms requests of different Pools (bombs, walls, boxes, items, players) calling TakeFromPoolArray only once.

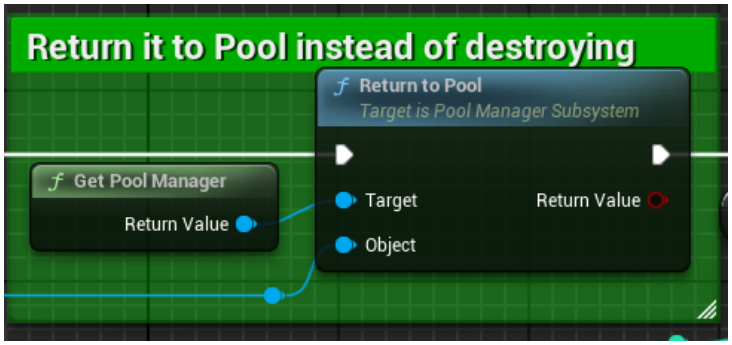
Return To Pool

The `**Return to Pool**` function is an essential part of the pool management system, designed to streamline the process of managing the lifecycle of pooled objects.

Key Features:
- **Efficient Object Management**: Enables objects taken from the pool to be returned and automatically deactivated.
- **Success Confirmation**: Confirms a successful return to the pool with a `true` value, or indicates failure with `false`.

Examples:

1. In Blueprints:



2. In code by object:

```
UPoolManagerSubsystem::Get().ReturnToPool(MyObjectPtr);
```

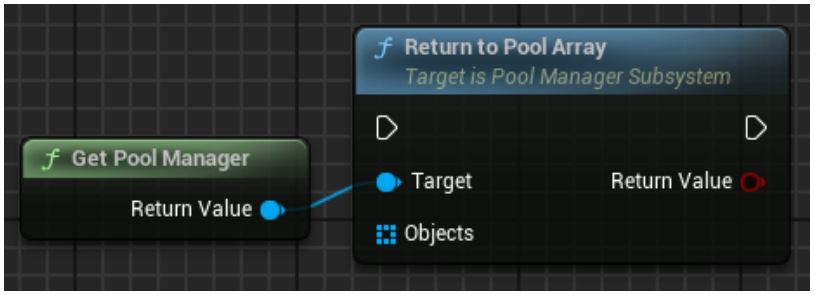
3. In code by Handle: an alternative for returning objects indirectly by their handle (hash). It is more reliable as it handles even the case when the object is not spawned yet:

```
UPoolManagerSubsystem::Get().ReturnToPool(Handle);
```

Return To Pool Array

The **Return To Pool Array** functions extend the capabilities of the standard Return to Pool method by allowing multiple objects or handles to be returned to the pool simultaneously.
Examples:

1. In Blueprints:



2. In code by object:

```
UPoolManagerSubsystem::Get().ReturnToPoolArray(MyObjects);
```

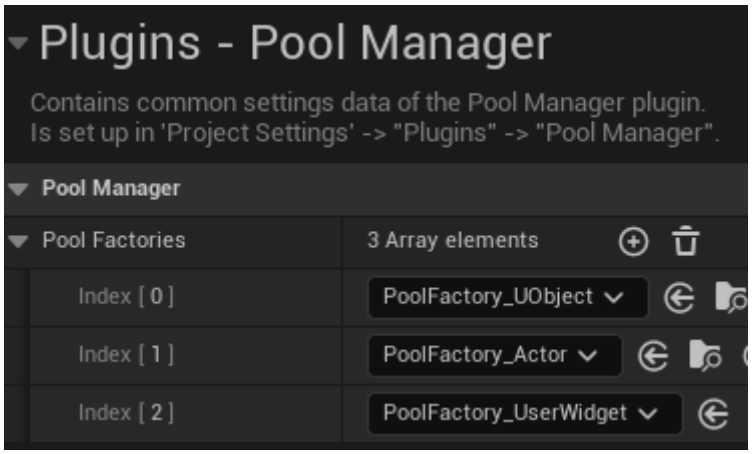
3. In code by Handle:

```
UPoolManagerSubsystem::Get().ReturnToPoolArray(Handles);
```

Factories

What is Factory

Default Classes that the Pool Manager supports out of the box:



- **PoolFactory_UObject**: is base class for all Factories. It handles all **simple** UObjects. Simple means just regular UObjects, but not other engine classes like Actors, Components, VFXs, Sounds etc.
- **PoolFactory_Actor**: handles all Actor-inherited objects.
- **PoolFactory_UserWidget**: handles all UserWidgets-inherited objects.

Implementing a new Factory is advantageous in next cases:

- **Extending supported classes:** If you wish to add support for pooling specific elements beyond Default Classes. By creating new Factory, you can implement support of anything like Components, VFXs, sounds, etc.
  *For such core Factories, **your pull request is appreciated** to be shared with others.*
- **For your own gameplay objects** like inventory object or trigger box actor to handle pool-specific logic like give them call about any event happened (taken from pool, returned etc).

The Factory architecture is as follows: our pooled object itself does not have any dependencies on the Pool Manager system and doesn't even know about it, while its Factory describes the logic of how to behave in Pool for this type of object and its children when the object is being spawned, destroyed, pooled, taken etc.

Unlike other solutions, where an Interface is added to the class to override any interface method, here you create a child factory to handle any custom pool-related logic of handled class.

Create new Factory

Any factory can be created either in code or in blueprints.

Step 1
Create new code or blueprint class inherited from one of defaul Pool Factory
E.g: create PoolFactory_ActorComponent inherited from PoolFactory_UObject for handling all Actor Components.

Step 2
Add new class to the '*Project Settings' -> "Plugins" -> "Pool Manager" -> "Pool Factories":*

Step 3
Override ***GetPoolObject*** function inside your new factor class: it should return the class you want to handle, it could be some base class, so all its children also will be handled by this Factory.
E.g: return UserWidget class for handling all widgets.

Step 4
Override any other method to handle custom logic. Calling *Super* inside for overridden function is optional to keep original behavior.
E.g: for Actor Components, I assume you would override `OnTakeFromPool` to register the component, change the Outer and enable ticking; override `OnReturnToPool` to unregister the component and stop ticking.

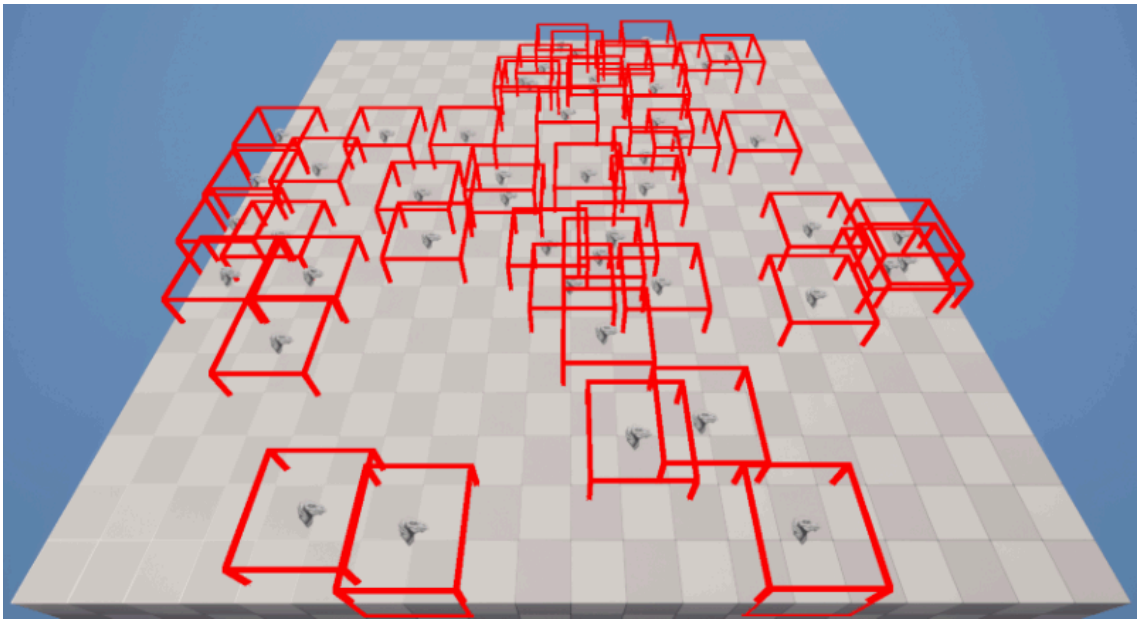
Sample Project

Please download the sample project from the next link to see how it works, featuring two examples in one project: first for blueprint-only developers pooling widgets and another version implemented in code pooling Draw Boxes: they both perform exactly the same logic, but for different object types.
https://github.com/JanSeliv/PoolManager/releases

1. Blueprint Sample: widgets

- **WBP_Canvas**: pools randomly up to 5 of WBP_Text subwidgets numerating them, '**TakeFromPoolArray'** and '**ReturnToPoolArray'** are used.

2. Code Sample: Draw Boxes



In code, pooling Draw Boxes demonstrate pooling randomly up to 50 Draw Boxes with the significant performance achieved by using a Pool Manager, with stable and high FPS:

- `***SomeGameplayClass***` *is* actor that infinitely pools randomly up to 50 Draw Boxes each 0.2 seconds in random location on the level. Instead of actual **spawning** and destroying, '**TakeFromPoolArray'** and '**ReturnToPoolArray'** are used.
- `***DrawBox***` is our object we pool. It's ticking to visualize red geometry box in current location. It is regular object to make example more interesting using next Factory. However, in real world, it would be just regular actor.
- `**PoolFactory_DrawBox**`: contains pooling-specific logic: resets location when Draw Box is returned to pool. This factory is optional and regular Actor factory can be used automatically by default.

3. Other Projects Examples

To view the Pool Manager in action, look at next projects searching by `TakeFromPool` and `ReturnToPool:

- Bomber Project repository: GeneratedMap.cpp. Pooling all actors on level (bombs, walls, boxes, items, players).
- Progression System plugin: PSMenuWidget.cpp. Pooling 'Star' widgets to display progression scores.

Cheats

At this moment, the Pool Manager does not provide its own cheats or debug visualizations. However, you can find useful built-in Engine cheats. All the following cheats help to track the amount of created objects.

Displays all stored objects in all pools:

- *DisplayAll PoolManagerSubsystem PoolsInternal*

Displays how many objects the engine created of a given code class. The name of the class should be without any prefix. For example, to display all objects of 'UDrawBox', it would look as follows:

- *DisplayAll DrawBox*

Displays all objects ever created in the Engine (even those that are not pooled):

- *stat UObjects*