Event Driven Pool Management System

An event based approach to pool management.

Jacob Homanics

Version 1.1.0

The Event Driven Pool Management System is a system which utilizes UnityEvents to make development an ease when dealing with GameObjects that need to be managed in a pool. What/How things are pooled is up to the developer.
The system allows for the developer to define what it means for a pooled object to be initialized, spawned, despawned, or terminated.
It is also built to work with any project, existing or new.

Features

- Provides a set of scripts that can be attached to any GameObject which allow for the utilization of the pool management optimization pattern.
- Allows for simple pool management of a GameObject prefab and paves way for extensive customization in these regards.
- Allows for complex pool management of a set of GameObject prefabs and paves way for extensive customization in these regards.
- Allows for complex pool management of a set of GameObject prefabs using weighted constraints, allowing for certain pool objects to spawn more frequently than others.

Basic API

Pool Entity

The Pool Entity is the pool object that is going to be pooled. It implements several key actions which are prevalent across the entire system. Specifically pertaining to the Pool Entity:

- Initialized: The Pool Entity was just instantiated into the scene by a Pool Entity Manager and is ready to be spawned. At this point, it is considered to be hidden from the player and not affecting any game logic.
- Spawned: The Pool Entity was just spawned by a Pool Entity Manager. At this point, it is considered to be visible to the player and is affecting game logic.
- Despawned: The Pool Entity was just despawned back to its respective Pool Entity Manager. At this point, it is considered to be hidden from the player and not affecting any game logic.
- Terminated: The Pool Entity was just terminated. Whether the pool entity has been initialized, spawned, or despawned, it will be destroyed from the scene.

The pool entity contains a set of methods and UnityEvents pertaining to these key actions. The Initialize and Spawn methods should be called exclusively by the pool entity manager to which the pool entity belongs to. The Despawn and Terminate methods may be called from any script.

The pool entity also contains a property of type PoolEntityManager. This will be the pool entity manager to which the pool entity belongs to. Usually this would be the pool entity manager that initializes the pool entity.

Properties:

- **PoolEntityManager PoolEntityManager {get; private set;}**: The pool entity manager to which the pool entity belongs to. Usually the pool entity manager that initialized the pool entity.

Fields:

- **Events events:** The container class for the UnityEvents pertaining to the pool entity.
    - **Initialized(PoolEntity)**: Invoked by the Initialize method.
    - **Spawned(PoolEntity)**: Invoked by the Spawn method.

    - **Despawned(PoolEntity)**: Invoked by the Despawn method.
    - **Terminated(PoolEntity)**: Invoked by the Terminate method.

Methods:

- **Initialize(PoolEntityManager)**: The Pool Entity sets its PoolEntityManager property to the passed in parameter. Then, invokes the Initialized UnityEvent passing itself in as a parameter.
- **Spawn(void)**: Invokes the Spawned UnityEvent passing itself in as a parameter.
- **Despawn(void)**: Invokes the Despawned UnityEvent passing itself in as a parameter.
- **Terminate(void)**: Invokes the Terminated UnityEvent passing itself in as a parameter

Pool Entity Manager

The Pool Entity Manager is the main driving force of the system. This is the script that mostly handles the Pool Entity instances (The Pool Entities Manager expands on it). As with Pool Entity, it also implements its own set of key actions. However it works a bit differently. Its methods are to be called directly by the developer, but most of its UnityEvents are actually tied to its Pool Entity instances. Here are its key actions:

- **Initialized**: The Pool Entity Manager was asked to create a number of Pool Entity instances and they are ready to be spawned or terminated.
- **OnEntityInitialized:** During the Initialization process, each pool entity initialized will fire its Initialized event, and in turn, this event fires.
- **Spawning:** The pool entity manager was asked to spawn a new pool entity.
- **OnEntitySpawned**: During the Spawning process, the pool entity chosen to be spawned will fire it's Spawned event, and in, this event fires.
- **Despawned All**: All spawned pool entities have been despawned specifically by the DespawnAll method.
- **OnEntityDespawned**: When a spawned pool entity is despawned, in turn this event fires.
- **Terminated**: The Pool Entity Manager was asked to terminate all pool entities.
- **OnTerminated**: A Pool Entity that belongs to the pool entity manager was just terminated.

Properties:

- **PoolEntitiesManager PoolEntitiesManager {get; private set;}**: The Pool Entities Manager to which the Pool Entity Manager belongs to. Usually the Pool Entities Manager that initialized the pool entity.
- **List<PoolEntity> ReadyInstances {get; private set;}**: All of the Pool Entity instances which belong to the Pool Entity Manager that are Initialized or Despawned.
- **List<PoolEntity> ActiveInstances {get; private set;}**: All of the Pool Entity instances which belong to the Pool Entity Manager that are Spawned.
- **List<PoolEntity> AllInstances {get; private set;}**: All of the Pool Entity instances which belong to the Pool Entity Manager.
- **PoolEntity LastSpawned {get; private set;}**: The last Pool Entity instance to have been spawned that belongs to the Pool Entity Manager.
- **PoolEntity LastDespawned {get; private set;}**: The last Pool Entity instance to have been despawned which belongs to the Pool Entity Manager.
- **Initialized(PoolEntityManager, List<PoolEntity>):** Reference to events.InitializationEvents.Initialized.
- **OnEntityInitialized(PoolEntityManager, PoolEntity):** Reference to events.InitializationEvents.OnEntityInitialized.
- **Spawning(PoolEntityManager, PoolEntity):** Reference to events.SpawnEvents.Spawning.
- **OnEntitySpawn(PoolEntityManager, PoolEntity):** Reference to events.SpawnEvents.OnEntitySpawn.
- **DespawnedAll(PoolEntityManager, List<PoolEntity>):** Reference to events.DespawnEvents.DespawnedAll.
- **OnEntityDespawn(PoolEntityManager, PoolEntity):** Reference to events.DespawnEvents.OnEntityDespawn.
- **Terminated(PoolEntityManager):** Reference to events.TerminationEvents.Terminated.
- **OnEntityTerminated(PoolEntityManager, PoolEntity):** Reference to events.TerminationEvents.OnEntityTerminated.

Fields:

- **PoolEntity poolEntity**: The Pool Entity prefab that will be used in the pooling system.
- **Int initialInstanceCount**: The amount of Pool Entity instances to Initialize.
- **OverflowType overflowType {Expandable, Recyclable}** : The expected behavior when the Pool Entity Manager attempts to spawn a Pool Entity instance, but there are no instances that are ready to be spawned.

    - **Expandable:** Initializes then spawns a new instance of the poolEntity field.
    - **Recyclable:** Despawns the earliest known spawned Pool Entity instance, then continues to spawn it again.
- **Events events**: The container class for the UnityEvents pertaining to the pool entity manager.
    - I**nitializationEvents InitializationEvents:**
        - **Initialized(PoolEntityManager, List<PoolEntity>):** Invoked within the Initialize method.
        - **OnEntityInitialized(PoolEntityManager, PoolEntity):** Invoked after the initialization of a pool entity pertaining to this pool entity manager.
    - **SpawnEvents SpawnEvents:**
        - **Spawning(PoolEntityManager, PoolEntity):** Invoked in the spawn method before PoolEntity parameter's Spawn event has been invoked.
        - **OnEntitySpawn(PoolEntityManager, PoolEntity):** Invoked as a listener to the PoolEntity parameter's Spawned event.
    - **DespawnEvents Despawn Events:**
        - **DespawnedAll(PoolEntityManager, List<PoolEntity>):** Invoked within the DespawnedAll method.
        - **OnEntityDespawn(PoolEntityManager, PoolEntity):** Invoked as a listener to the PoolEntity parameter's Despawned event.
    - **TerminationEvents TerminationEvents:**
        - **Terminated(PoolEntityManager):** Invoked within the Terminate method.
        - **OnEntityTerminated(PoolEntityManager, PoolEntity):** Invoked as a listener to the PoolEntity parameter's Terminated event.

Methods:

- **Initialize(PoolEntitiesManager)**: The Pool Entity Manager sets its PoolEntitiesManager property to the passed in parameter. Next, it Instantiates and Initializes a set of Pool Entity instances and adds them to the ReadyInstances list. The number of instances to Instantiate and Initialize is equal to the value of initialInstancesCount. Finally, invokes the Initialized UnityEvent passing itself and a list of the Initialized instances as parameters.
- **Initialize(void)**: First, it Instantiates and Initializes a set of Pool Entity instances and adds them to the ReadyInstances list. The number of instances to Instantiate and Initialize is equal to the value of initialInstancesCount. Finally, invokes the

Initialized UnityEvent passing itself and a list of the Initialized instances as parameters.

- **Terminate(void)**: Loops through the AllInstances list and calls their Terminate method. Finally, the Pool Entity Manager invokes its Terminated UnityEvent passing itself as the parameter.
- **Spawn(void)**: Grabs the Pool Entity instance found in index 0 of ReadyInstances and calls its Spawn method. If there are no ReadyInstances, then the manager will react accordingly based on its overflowType. Then it removes the instance from ReadyInstances and adds it to ActiveInstances. Finally, it invokes its OnSpawned UnityEvent passing itself and the spawned instance as parameters.
- **DespawnAll(void)**: Loops through the ActiveInstances list and calls their Despawn method. Finally, the Pool Entity Manager invokes its DespawnedAll UnityEvent passing itself and the despawned Pool Entity instances as parameters.

The Pool Entity Manager lacks the ability to despawn a pool entity directly. There are several reasons for this. If a method existed in the pool entity manager with the parameter of a pool entity, then there would need to be checks in place to make sure that the pool entity was spawned and that it belongs to the pool entity manager. The other is kind of self explanatory. Since the pool entity knows which pool entity manager it belongs to, then there is unnecessary “cushion” added to the code if we needed to have the pool entity manager despawn the pool entity. There also does exist a method in the pool entity manager to despawn all of the active pool entity instances for ease of use.

The termination key action follows the same principles.

Pool Entities Manager

The Pool Entities Manager is used for when more complex pool entity management is necessary. For instance, after a timer reaches 0, then you may want to spawn from a random Pool Entity Manager from within a set of Pool Entity Managers. It allows for events to happen at the grand scale and not only at the Pool Entity Manager and Pool Entity scale. Its key actions are:

- **Initialized**: Initializes all Pool Entity Managers that belong to the Pool Entities Manager.
- **Despawned All**: Despawns all Pool Entity Managers that belong to the Pool Entities Manager.
- **Terminated**: Terminates all Pool Entity Managers that belong to the Pool Entities Manager.
- **OnInitialized**: A Pool Entity Manager that belongs to the Pool Entities Manager was just initialized.
- **OnSpawned**: A Pool Entity Manager that belongs to the Pool Entities Manager was just spawned from.
- **OnDespawned**: A Pool Entity Manager that belongs to the Pool Entities Manager was just despawned from.
- **OnTerminated**: A Pool Entity Manager that belongs to the Pool Entities manager was just terminated.

Fields:

- **BasePoolEntitiesContainer container** : The BasePoolEntitiesContainer which contains a list of the Pool Entity Managers to be managed by the Pool Entities Manager.

UnityEvents:

- **Initialized(PoolEntitiesManager, List<PoolEntityManager>)**: Invoked by the Initialize method.
- **DespawnedAll(PoolEntityManager, List<PoolEntity>**: Invoked by the DespawnedAll method.
- **OnInitialized(PoolEntitiesManager, PoolEntityManager)**: Invoked after a Pool Entity Manager is initialized.
- **OnSpawned(PoolEntitiesManager, PoolEntityManager, PoolEntity)**: Invoked after a Pool Entity is spawned from a Pool Entity Manager found within the container.

- **OnDespawned(PoolEntitiesManager, PoolEntityManager, PoolEntity)**: Invoked after a Pool Entity is despawned from a Pool Entity Manager found within the container.
- **OnTerminated(PoolEntityManager, PoolEntityManager)**: Invoked after a Pool Entity Manager is terminated that is found within the container.

Methods:

- **Initialize(void)**: Loops through all Pool Entity Managers from the container and calls their Initialize method, passing itself as a parameter. Then, it invokes its Initialized UnityEvent passing itself and the Pool Entity Managers as parameters.
- **Terminate(void):** Loops through all Pool Entity Managers from the container and calls their Terminate method. Then, it invokes its Terminated UnityEvent passing itself and the Pool Entity Managers as parameters.
- **Spawn(void)**: Calls the container's GetRandomEntity method, Then calls the returned Pool Entity Manager's Spawn method.
- **DespawnAll(void)**: Loops through all Pool Entity Managers from the container and calls their Despawn method. Then, it invokes its DespawnedAll UnityEvent passing itself and the Pool Entity Managers as parameters.

Abstract BasePoolEntitesContainer

The BasePoolEntitesContainer is an abstract class where the derived children define the behavior for how a Pool Entities Manager spawns from its Pool Entity Managers. Currently there are only two supported containers: PoolEntitiesContainer and WeightedPoolEntitiesContainer. The idea is for a container to be created which fits your needs, whatever they may be, without altering the PoolEntity, PoolEntityManager, PoolEntityManager, and even the BasePoolEntitiesContainer classes. The containers do not implement the key actions that the other scripts do. They are simply what they are called: Containers. They hold information and provide methods to make obtaining this information easier. They do not implement any UnityEvents.

The BasePoolEntitiesContainer is abstract, thus it only contains abstract properties which return information.

Properties:

- **Abstract PoolEntityManager GetRandomEntity**: Laying out the signature for getting a random Pool Entity Manager within the constraints of the derived class.
- **Abstract List<PoolEntityManager> GetAllEntities:** Laying out the signature for getting all Pool Entity Managers that belong to the container.

PoolEntitiesContainer

This is the most basic container. It holds a list of Pool Entity Managers and has the ability to return one of them completely at random.

Fields

- **List<PoolEntityManager> poolEntityManagers**: A list of Pool Entity Managers.

Properties:

- **Override PoolEntityManager GetRandomEntity**: Generates a random number from 0 to the count of the poolEntityManagers list and returns a PoolEntityManager at the index of the random number.
- **Override List<PoolEntityManager> GetAllEntities:** returns the poolEntityManagers list.

WeightedPoolEntitiesContainer

This container is a bit more complex. It allows for weight to be added to a set of PoolEntityManager lists. This is useful if you have a PoolEntitiesManager that can spawn a kind of mineral. Sapphires and Rubies should have a 75% chance to spawn. Meanwhile Diamonds and Emeralds should have a 25% chance to spawn.

Fields

- **List<WeightedPoolEntities> weightedPoolEntities**: A list of Weighted Pool Entities.

Properties:

- **Override PoolEntityManager GetRandomEntity**: Generates a random number from 0 to 1 and adds all Pool Entity Managers found within the WeightedPoolEntites list to a local PoolEntityManager list. Then, it generates a random number from 0 to the count of the local poolEntityManagers list and returns a PoolEntityManager at the index of the random number.
- **Override List<PoolEntityManager> GetAllEntities:** Returns all PoolEntityManagers found in the list of weightedPoolEntities.

WeightedPoolEntities

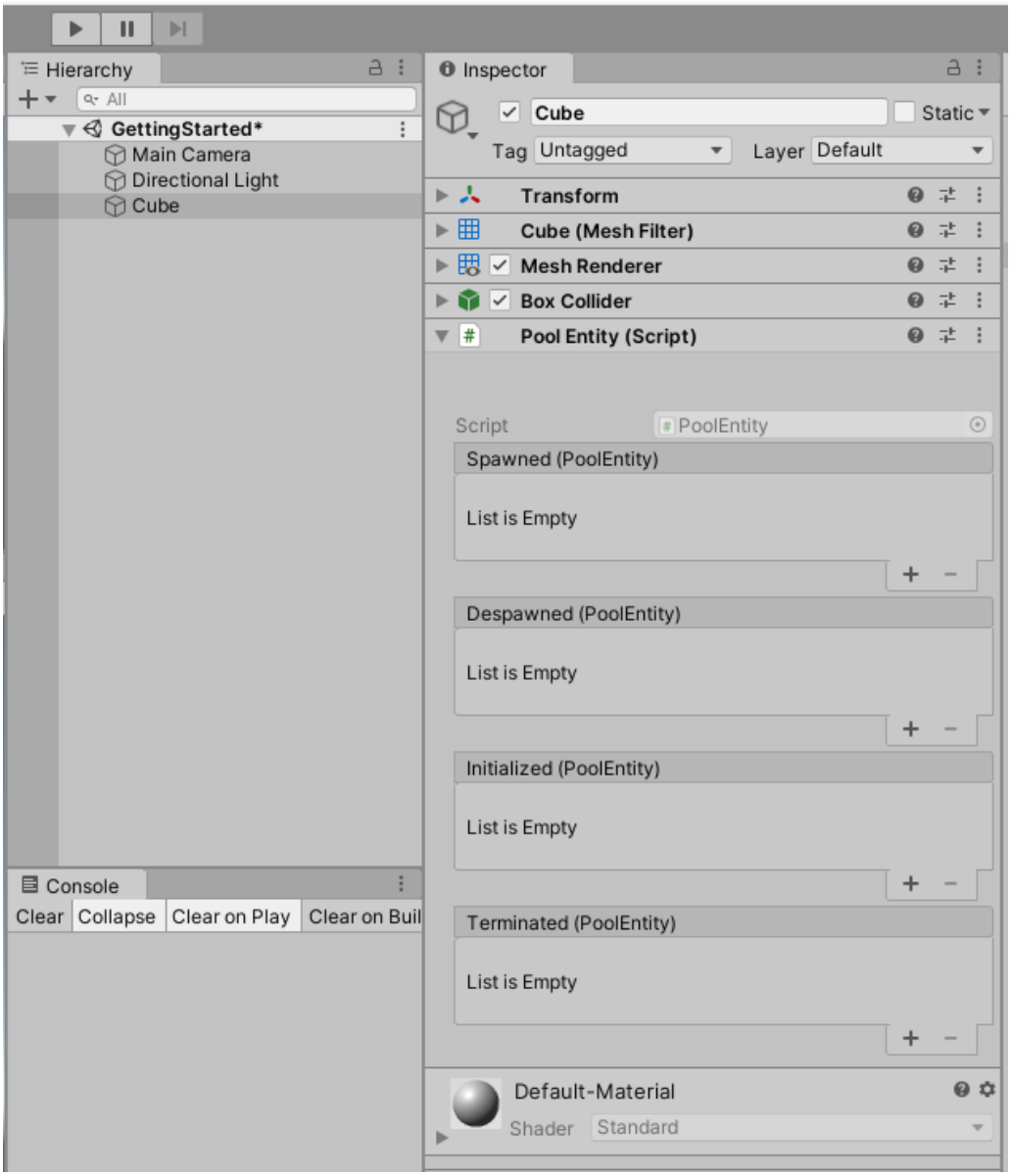
This can be thought of as a sub-container for a set of Pool Entity Managers. It contains no functionality as the information is retrieved from it and handled by the WeightedPoolEntitiesContainer class.

Fields

- **List<PoolEntityManager> poolEntityManagers**: A list of Pool Entity Managers.
- **Float minWeight**: The minimum weight that will decide if this WeightedPoolEntities gets chosen in the GetRandomEntity method in WeightedPoolEntitiesContianer. The value must strictly be between 0 and 1.
- **List<PoolEntityManager> poolEntityManagers**: The maximum weight that will decide if this WeightedPoolEntities gets chosen in the GetRandomEntity method in WeightedPoolEntitiesContianer. The value must strictly be between 0 and 1.

Getting Started

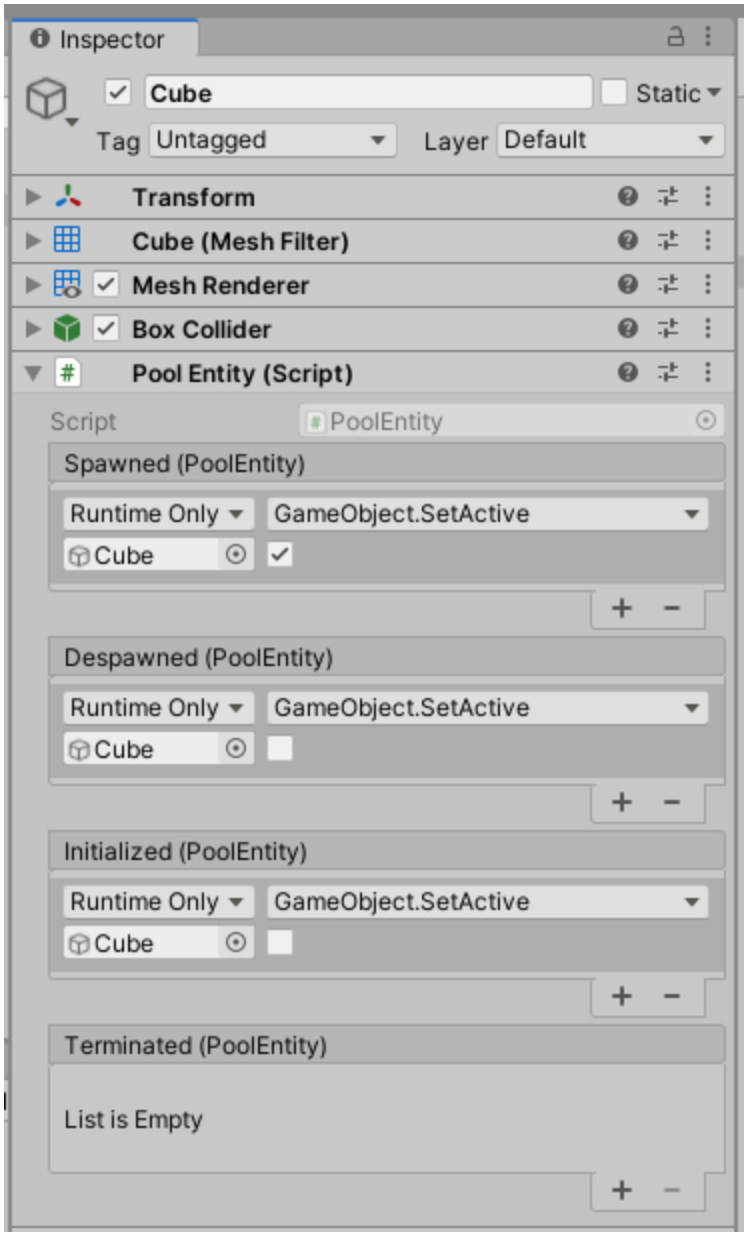
Starting from an empty scene, create a Cube GameObject. Add the Pool Entity script to it.



You can see that the inspector only shows the UnityEvents representing the key actions. The PoolEntityManager is hidden by default however you are able to view but not modify it by entering the inspector's debug window.

The most used UnityEvents are Initialized, Spawned, and Despawned. Initialized and Despawned end up having very similar, if not identical, listeners. However, there will be situations where they end up with different listeners.

Add the Cube itself as a listener to the Spawned, Despawned, and Initialized events. Set the listener function to GameObject.SetActive(). Set the parameter value for the Spawned listener to true.



Initialized and Despawned usually mean that the GameObject is not visible to the player and is not an active "contributor" to the scene. Therefore, we want to set the GameObject to inactive whenever those UnityEvents are invoked.

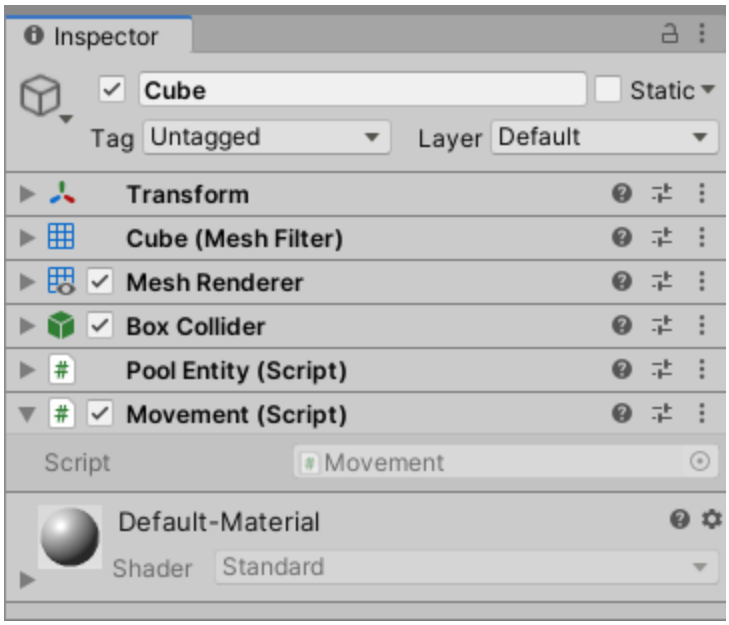
When a Pool Entity is Spawned, this usually means that the GameObject is active and is "contributing" to the scene.

As you can see, the listeners do not need to accept the same parameters as the UnityEvents. However for more dynamic uses, you may want to make a custom method which does so.

Create a new script called “Movement”. Its a simple script that will move the Cube GameObject forward on the z Axis.

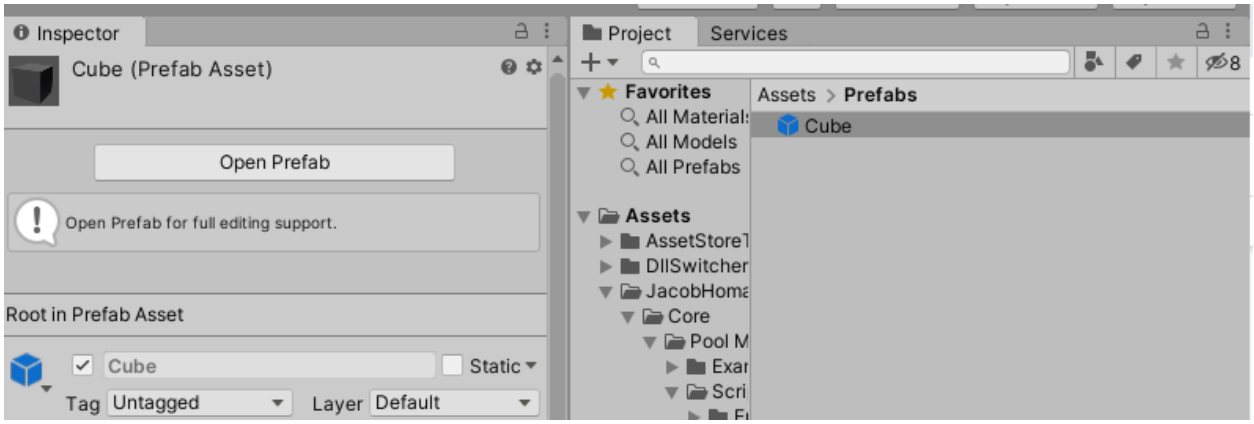
```
public class Movement : MonoBehaviour
{
    void Update()
    {
            transform.Translate(Vector3.forward * Time.deltaTime);
    }
}
```

After creating it, return to the editor and add it to the Cube GameObject.

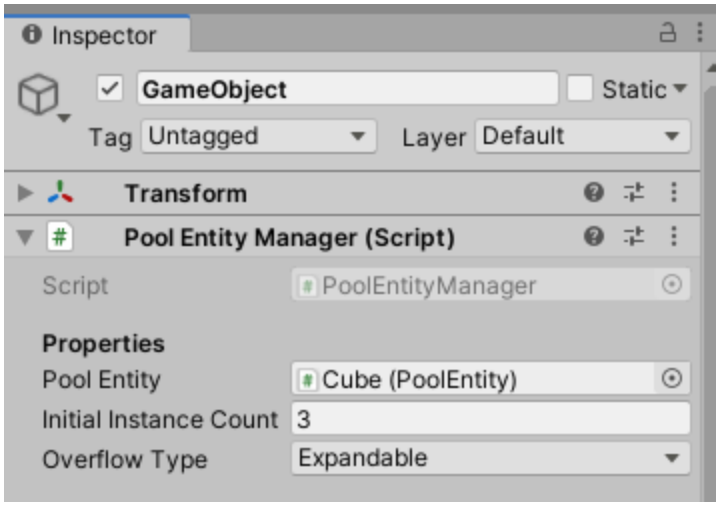


At this point the cube is now able to be pooled with some purpose. Whenever it is spawned, it will move forward on the z axis, and whenever Initialized and Despawned it will be hidden from the player and not affecting the scene in any way.

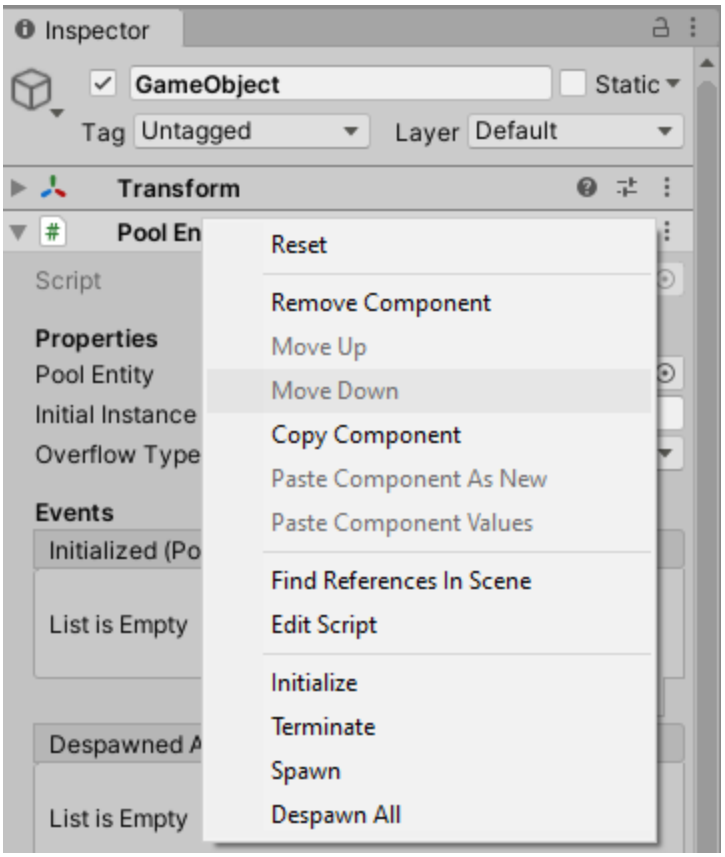
Drag the Cube into your projects folder to turn it into a Prefab.
Also remove the instance of the cube from your scene.



Create an empty GameObject and add the Pool Entity Manager script to it.
Add the Cube Prefab into the Pool Entity field.
Set Initial Instance Count to 3.

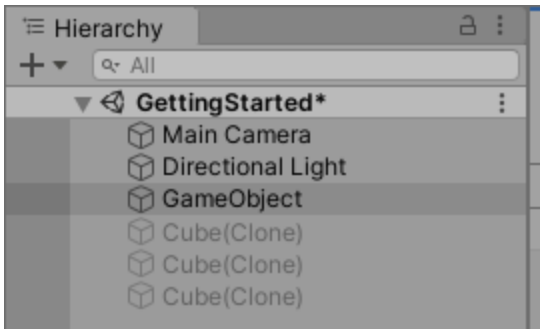


The Pool Entity Manager has commands for each of its key actions. These are helpful for quick testing/debugging. The Pool Entity Manager does not Initialize itself at the start of the scene as there are situations where that is not preferred. Another great example for the developer to decide how the system is set up.

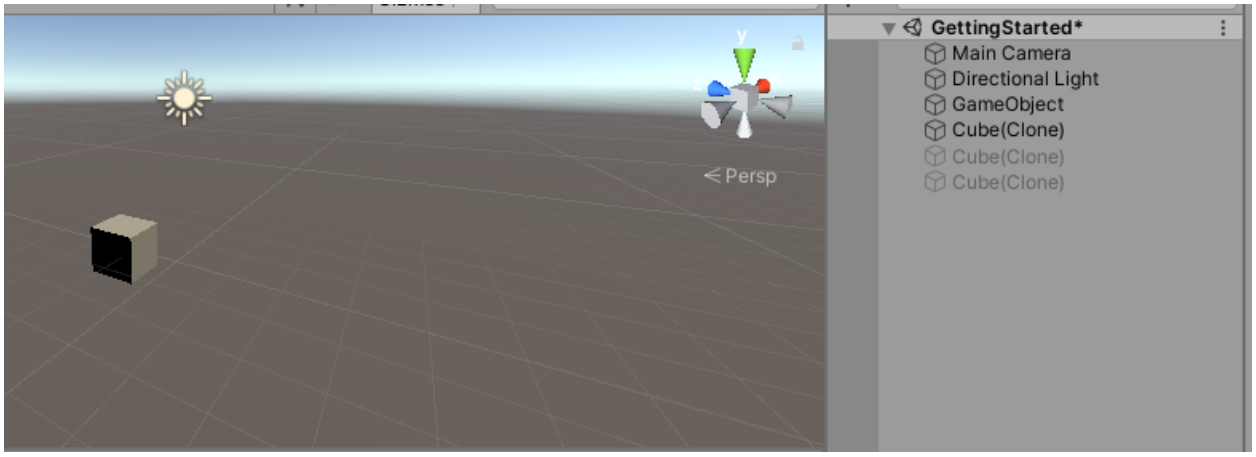


At this point it is possible to enter Play Mode. Use the context menu to Initialize the Pool Entity Manager.

Now the Pool Entity instances are created. Remember, their behavior at the point of Initialization is to be hidden from the player and not contribute to the scene at all. In this case, simply means that the GameObject is set to inactive.



Using the same context menu, select the Spawn option.



The first Cube instance will be set to active and it will be moving forward on the z axis. You can use the context menu's Spawn option as many more times as you would like. If your Pool Entity Manager's OverflowType is set to Expandable and if you select the Spawn option when all current instances are spawned, then a new cube instance will be created. If it is set to Recycable, then you will notice some weird behavior. The earliest spawned instance should reset its position back to the start. Another thing that we do not have implemented at this point is despawning the cube instances. Let's go ahead and add both of those functions.

Create a script called "TransformSpawner".

```
using JacobHomanics.Core.PoolManagement;
using UnityEngine;

public class TransformSpawner : MonoBehaviour
{
    public void OnSpawn(PoolEntityManager manager, PoolEntity entity)
     {
           entity.transform.position = transform.position;
     }
}
```

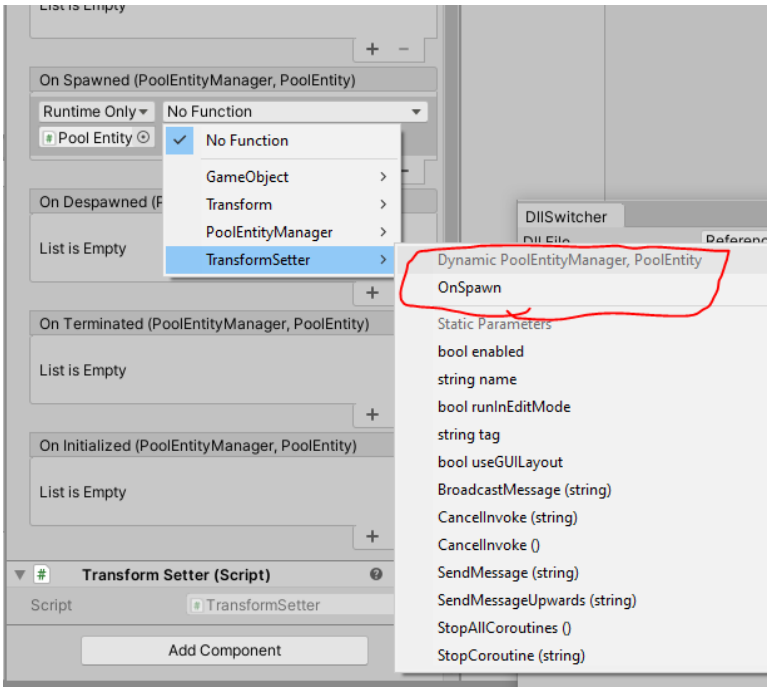
This script's OnSpawn method will be added as a listener to the Pool Entity Manager's OnSpawned UnityEvent. Anytime a Pool Entity is spawned from the Pool Entity Manager this method will fire. Its parameters match with the OnSpawned UnityEvent so we are able to dynamically access the spawned entity and handle it appropriately.

Lets go ahead and add it as a listener.

Add the TransformSpawner to the same GameObject that your Pool Entity Manager script is attached to.

Drag the TransformSetter script to the script field in the OnSpawned UnityEvent.

This part is important, as we need the information passed in from the event into the TransformSetter's OnSpawn method.



Make sure that you select the Dynamic version of the method.

Now whenever the cube instances are spawned, they will always start at the position of the transform which the TransformSetter is attached to.

We also still need to make the Pool Entity instance despawn.

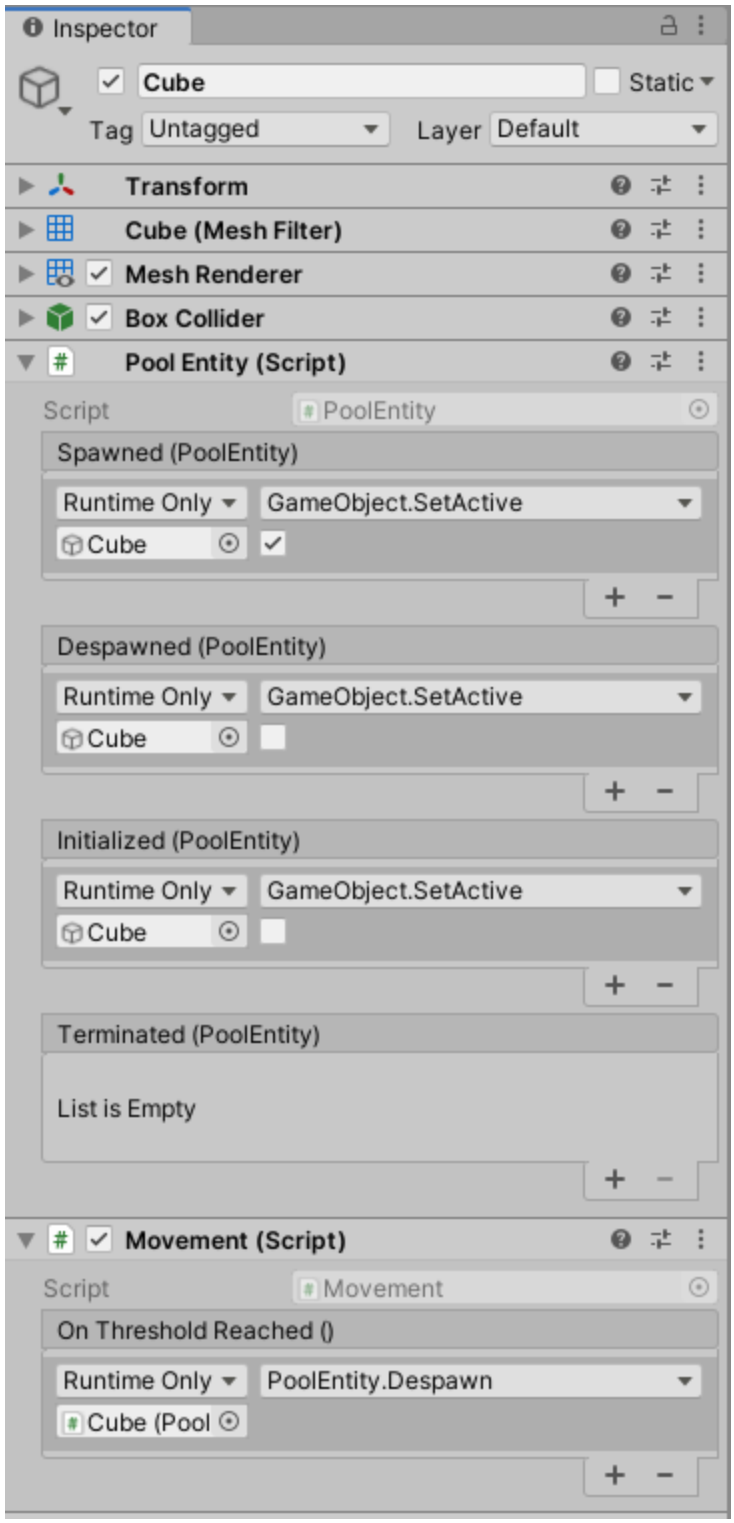
```
public class Movement : MonoBehaviour
{
      public UnityEvent OnThresholdReached;

    void Update()
    {
            transform.Translate(Vector3.forward * Time.deltaTime);
            if (transform.position.z > 3f)
                  OnThresholdReached?.Invoke();
      }
}
```

Update your movement script to include the UnityEvent, add the if statement, and invoke the UnityEvent. Now, whenever the cube instance moves past a certain distance it will invoke the UnityEvent.

Return to your Cube Prefab.

Add the Pool Entity's Despawn method as a listener to the Movement script's OnThresholdReached UnityEvent.

Remember, we set the cube to be inactive whenever it is despawned. This is the same behavior we saw with the Initialized event.



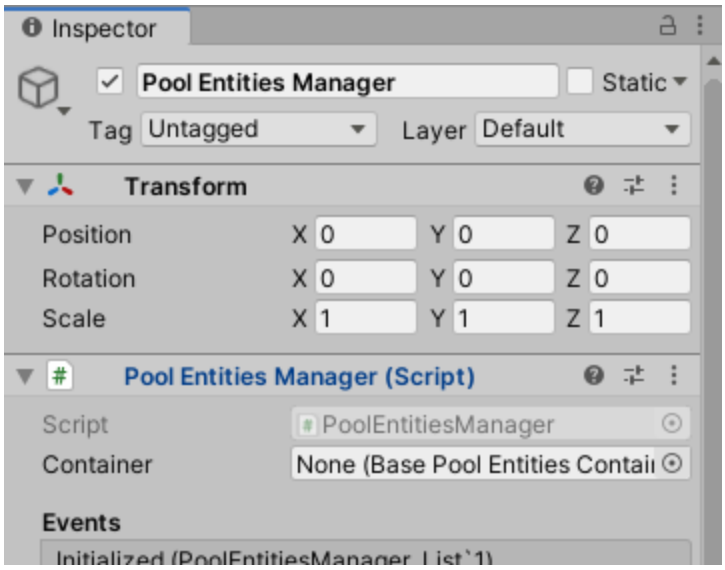
Re enter play mode and start spawning instances!

Now the cube instances should be despawning properly, Their positions are being reset on spawn, and you can now successfully re-use the pool instances!

Congratulations, you have made your first pool management system. It is so easily expandable and your code is as decoupled as can be!

……

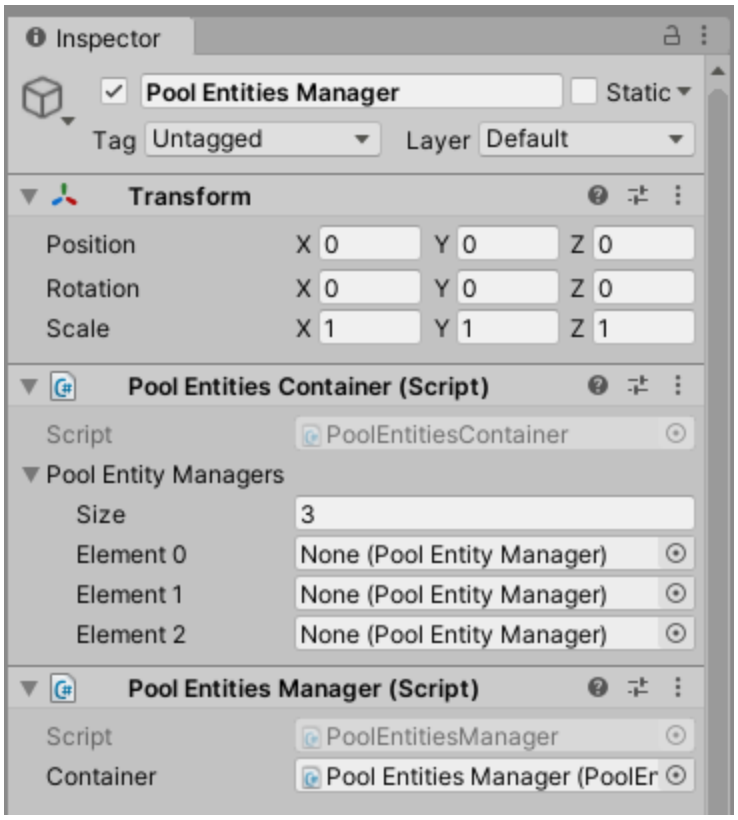
Ok, so you don't have a simple game. You have three different objects to spawn. Fortunately, the Pool Entities Manager has you covered.
Create an empty GameObject and add the Pool Entities Manager script to it.
Make sure the new GameObject's position is zeroed out.



There are currently two types of containers: PoolEntitiesContainer and WeightedPoolEntitiesContainer. Go ahead and add the PoolEntitiesContainer to the same GameObject.

Drag the Pool Entities Container into the Container field of the Pool Entities Manager.

Also set the size of the Pool Entity Managers list to 3 in Pool Entities Container.

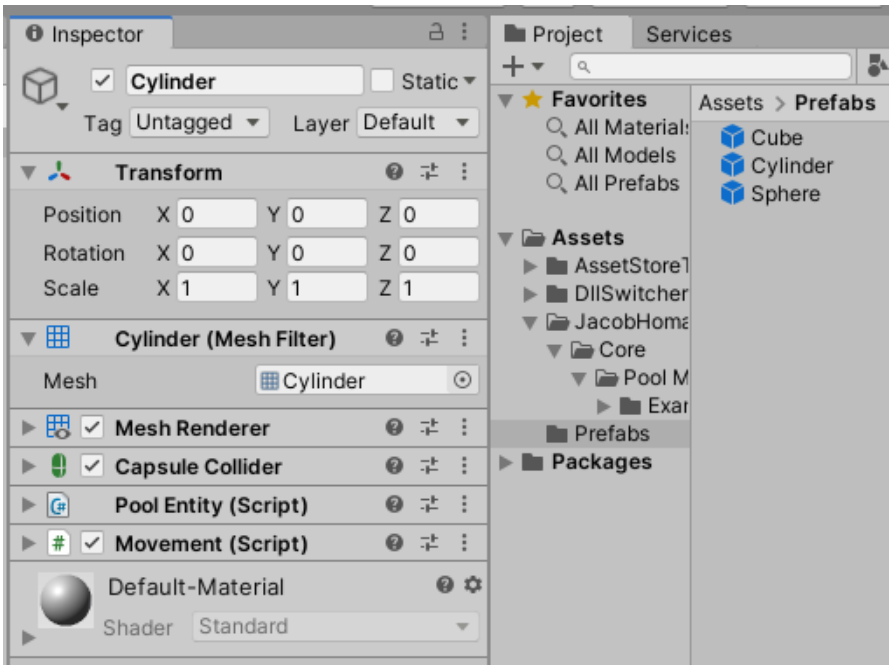


We will come back to this.

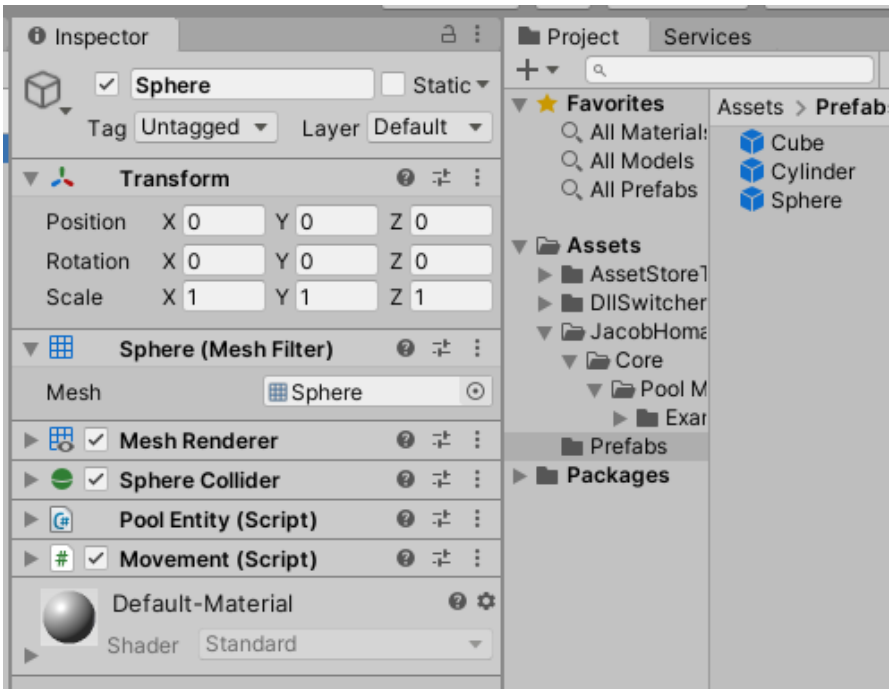
Duplicate your Cube Pool Entity prefab.

Rename it to Cylinder.

Remove its Box Collider and change its Mesh Filter's Mesh field to a Cylinder.



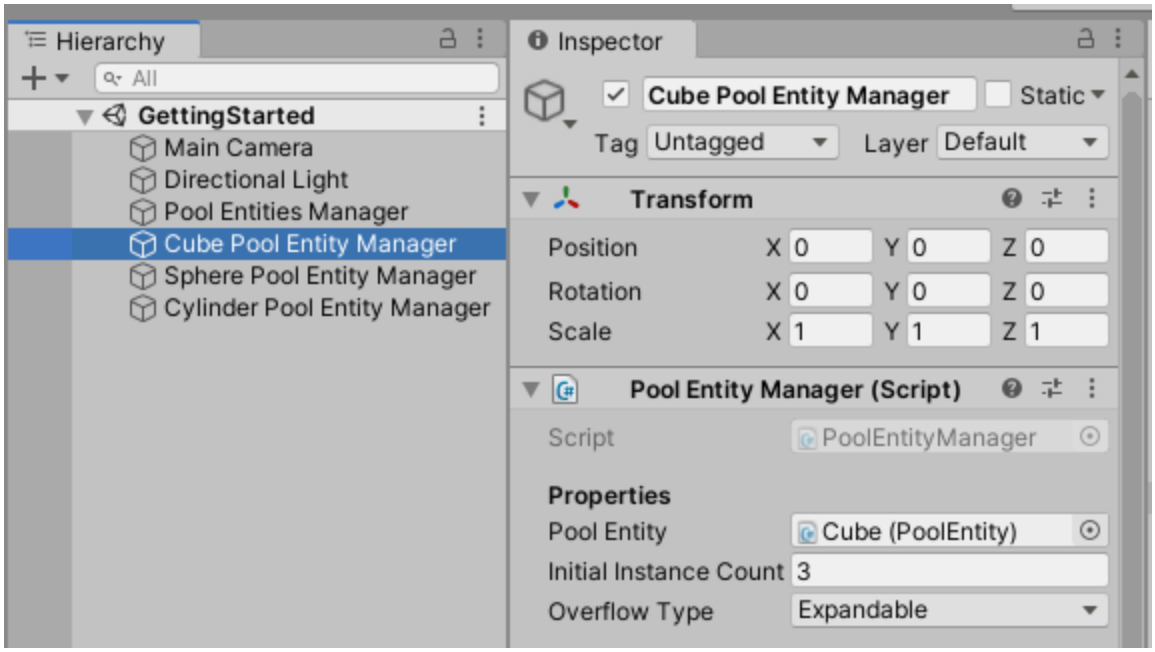
Repeat the process doing the same thing to create a sphere.

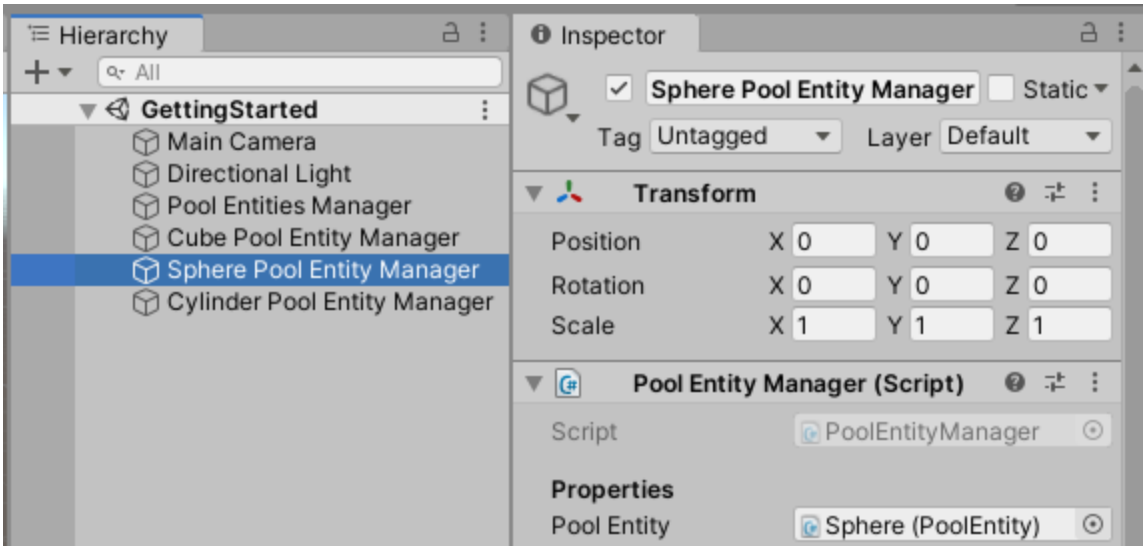


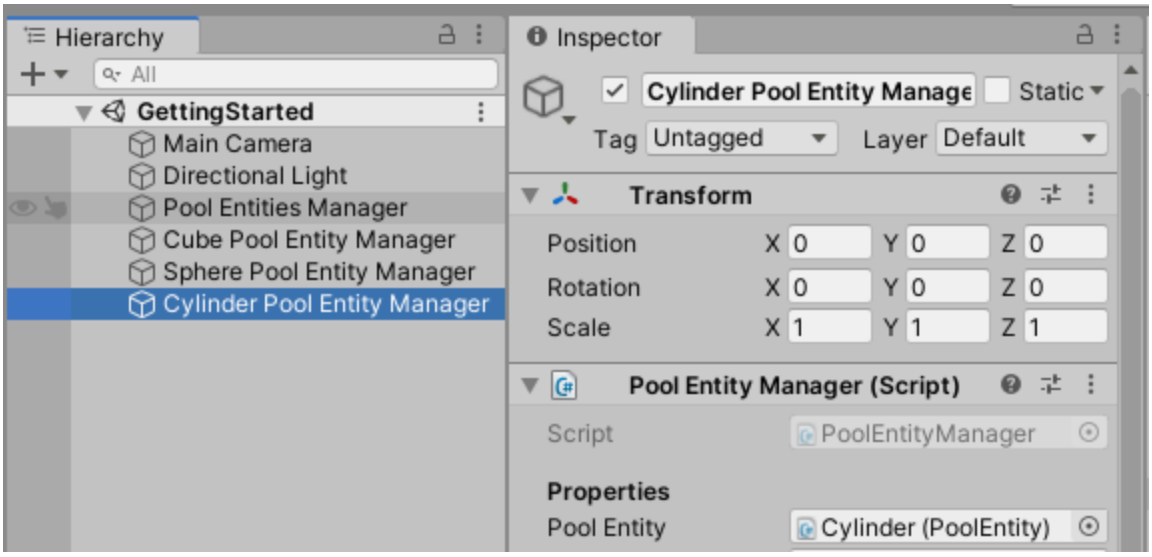
Do the same thing to your Cube Pool Entity Manager.

Make two duplicates and rename them accordingly.

Drag the proper prefab into the Pool Entity field.

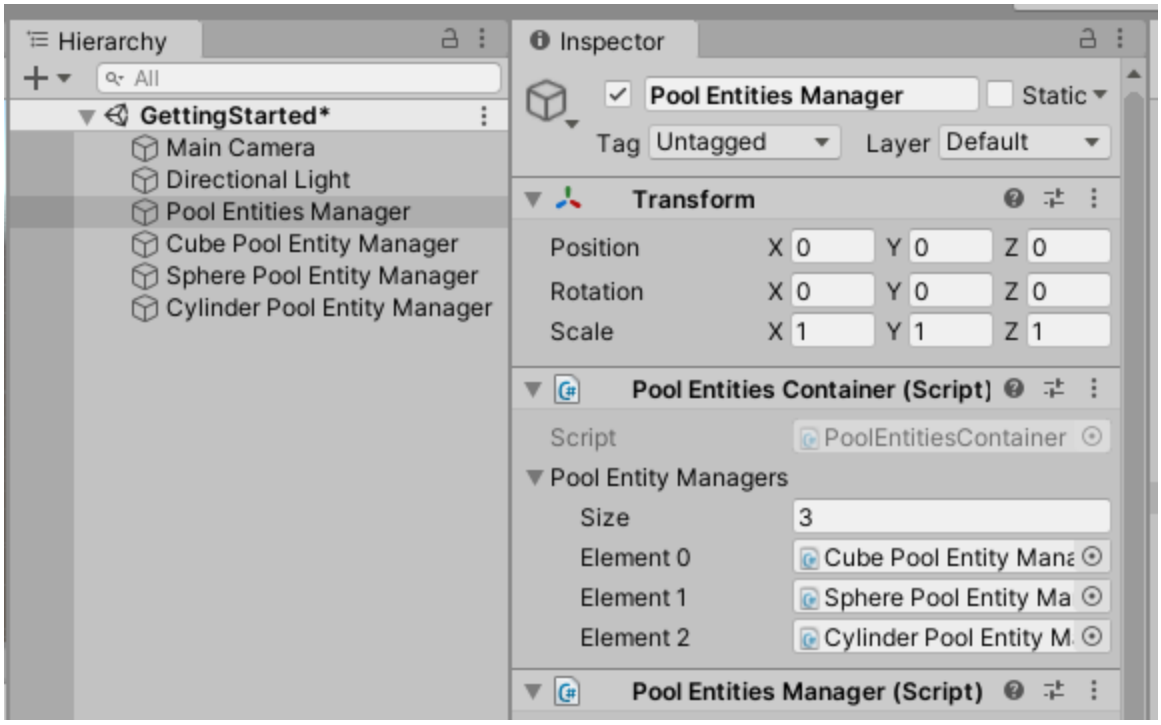




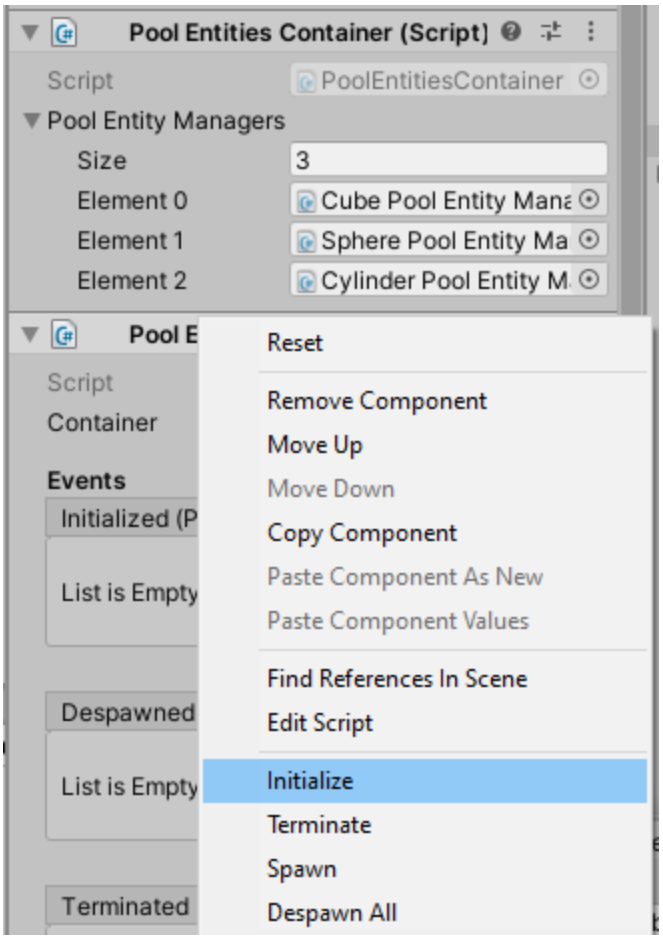


There are now three pool entity managers created and successfully able to be used properly.

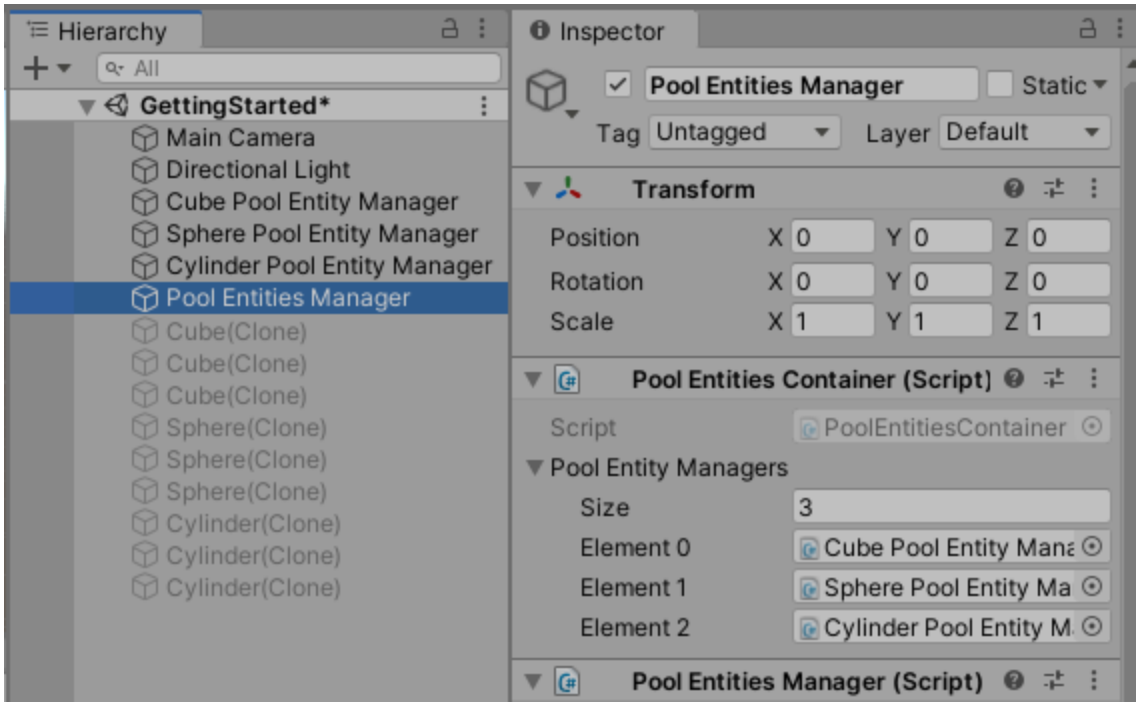
Return to the Pool Entities Manager GameObject and in the Pool Entities Container script, add the three Pool Entity Managers scripts to its list.



The Pool Entities Manager script contains the same context menu options as the Pool Entity Manager.



Now enter play mode. Select the Initialize context menu option.



With the click of a single button, all three Pool Entity Managers are now Initialized.

Similarly, if you select the Spawn context menu option on the Pool Entities Manager, it will now spawn randomly from the Cube, Sphere, or Cylinder Pool Entity Manager.

That's it for the Pool Entities Manager! It's that simple to expand from a single Pool Entity Manager.

To keep cleaner code and to allow for easier expansion, there is something we should do.

Currently each Pool Entity Manager sets its spawned pool entity’s position.

We can easily put this work onto the Pool Entities Manager now!

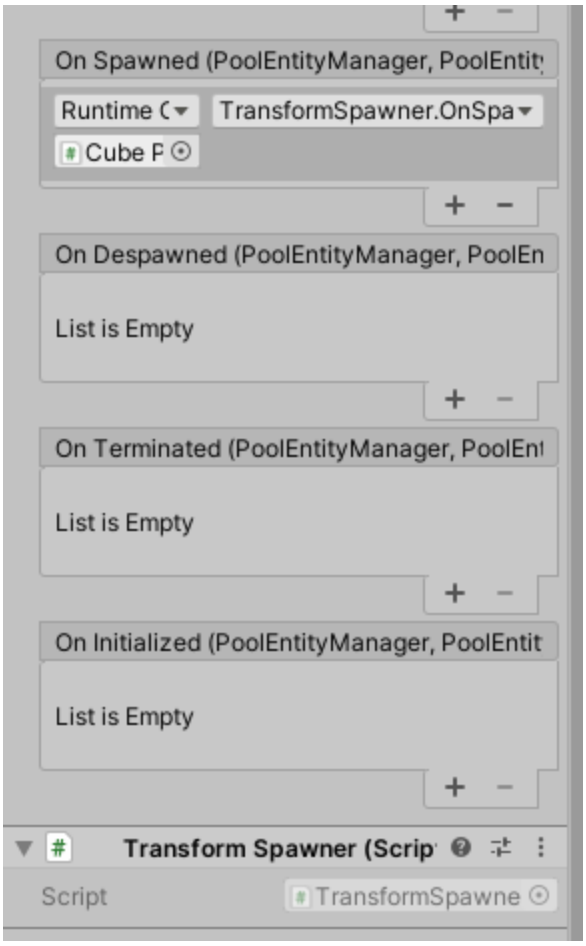
Add the following method to your TransformSpawner script.

Now, for whichever case your is more convenient for you, your TransformSpawner script can set the spawned entity's position whether you are dealing with a single Pool Entity Manager or a Pool Entities Manager.

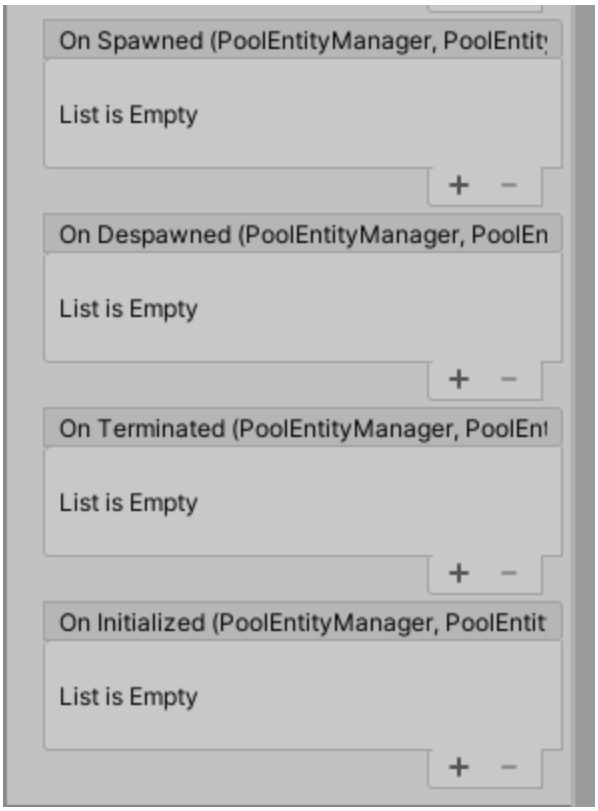
```
public void OnPoolEntitiesManagerSpawned
      (
            PoolEntitiesManager poolEntitiesManager,
            PoolEntityManager manager,
            PoolEntity entity)
      {
            entity.transform.position = transform.position;
      }
```

Go to each Pool Entity Manager and remove the TransfromSpawner script, remove the listeners in their OnSpawn method associated with the script.

Before



After



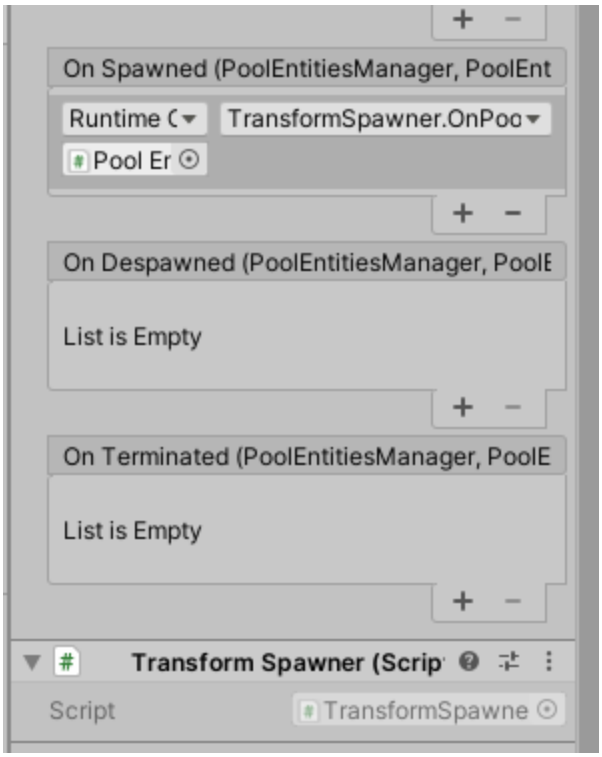
Return to your Pool Entities Manager script.

Add the TransformSpawner script.

Add the TransformSpawner's OnPoolEntitiesManagerSpawned method DYNAMICALLY as a listener to the Pool Entities Manager OnSpawned UnityEvent.

If you test in play mode, you will experience no difference visually. The shapes still spawn at the same point. But now the work is done on the Pool Entities Manager.

Each Pool Entity Manager does not need to worry about this and this will allow for easier modification of functionality if needed.



Generally it is a good idea to put the same functionality on the Pool Entities Manager if all Pool Entity Managers exhibit the same functionality.

……..

Ok….but I just heard you say "Well I think a Cube and a Cylinder are rare shapes. They shouldn't spawn as often as the sphere."

This is where the Weighted Pool Entities Container comes in. It allows for weight to be added to the spawning of Pool Entity Managers.

Remove the Pool Entities Container from your Pool Entities Manager GameObject.

Add the Weighted Pool Entities Container script to the same GameObject.

Set the size of its Weighted Pool Entities list to 2.

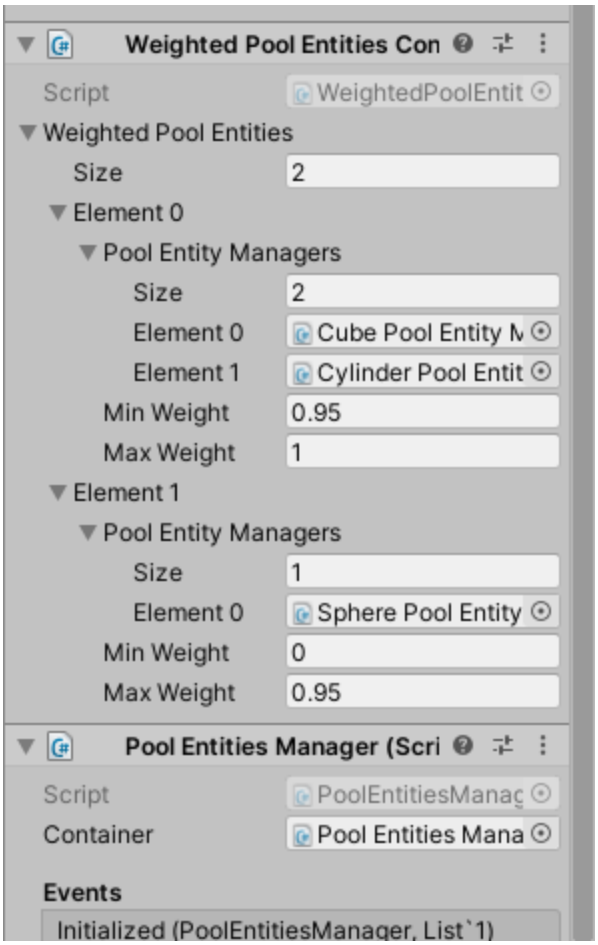
Add the Cube and Cylinder Pool Entity Managers into Element 0's Pool Entity Managers list.

Add the Sphere Pool Entity Manager into Element '1s Pool Entity Manager list.

Set Element 0's Min Weight to 0.95 and its Max Weight to 1.0.

Set Element 1's Min Weight to 0 and its Max Weight to .95f.

Also update the container field in the Pool Entities Manager to the Weighted Pool Entities Container script.



The weights work in the form of a normalized percentage. When the Pool Entities Manager's Spawn method is called, it will generate a random number and select a random Pool Entity Manager from within the Weighted Pool Entities list that the random number is found within their weight.

The Pool Entities Manager has a 5% chance to spawn a Cube or a Cylinder.

It also has a 95% chance to spawn a sphere.

Test it out in play mode! You will most likely get a sphere to spawn, but on the off chance, a Cube or Cylinder will be spawned!

Report a Bug

You may send all bugs to the email:
homanicsjake@gmail.com
You can send whatever you believe is necessary for me to examine and solve the bug. This may include projects, scripts, screenshots, videos.

Request a feature

I believe that this system fits most cases for the needed pool management system, although not every scenario is being accounted for. However it is my goal for this system to account for ALL pool management scenarios. I have set it up to be easily expanded upon where you can easily implement features. If you believe your own implemented feature or you think a feature fits the package, then send any information regarding that to:
homanicsjake@gmail.com

Contact Me

I believe this package has a lot of potential. I see it tried and proven in my own projects. Currently any time I require pool management I go to this project. This is why I believe a solid community can be created around this. I am open to all opportunities regarding this project. Potential collaborators, criticizers, future opportunists, celebrators, are all welcome to email me at:
homanicsjake@gmail.com.