CardboardCore Pooling

Goal of this package

CardboardCore Pooling gives a very straightforward but flexible approach for re-using GameObjects. This package works with Addressables, making memory management even easier.

Requirements

- Unity 2020.3.16f1 or later
- Addressables 1.16.19 or later

How to use

PoolConfigs

For starters, a `PoolConfig` is required to hold references to one or multiple prefabs. `PoolConfigs` are `ScriptableObjects` and can be created by pressing right-mouse button in the project tab -> Create -> CardboardCore -> Pooling -> PoolConfig.

`PoolConfigs` need to be added to your `Addressable Assets`. To do this, simply check the "Addressable" tick and give it a simple reference name. As you'll be using this name as a key later.

Once you're finished configuring and naming your `PoolConfig`, press the `Save & Generate Names` button to generate a static helper class to easily pick a specific prefab to spawn.

For performance reasons, it's recommended to have at least one custom `MonoBehaviour` attached to your pooled prefabs, and make sure they're the first component found on the `GameObject`, right underneath the `Transform` component. If there's no custom `MonoBehaviour` attached, it'll need to have at least another type of `MonoBehaviour` attached to it, like Unity's `Image` or `NavMeshAgent` to give a few random examples.

PoolManager

To start popping (moving a pooled object from the pool and activate it) and pushing (moving a pooled object back to the pool and deactivate it) your prefabs, a single instance of the `PoolManager` is required. The `PoolManager` is a pure C# class (not a `MonoBehaviour`) to add some extra flexibility. However, it can of course be wrapped in a `MonoBehaviour` if this is what is preferred.

It's possible to have multiple instances working in parallel to each other, but this is not recommended as this would not create any technical or use-case advantages.

Pool

During runtime, instances of `Pools` can be created. One single instance of a `Pool` should be created per `PoolConfig`.

To create a new `Pool`, call `PoolManager.RequestPool("<PoolConfig Addressable Key>")`. Mind that the name of the `PoolConfig` should be the `PoolConfig's` Addressable key (as mentioned earlier under the `PoolConfig` section).

Doing so will cause the `PoolManager` to either set up a new- or get an existing `Pool` asynchronously. a `Task<PoolConfig>` will be returned which has to be `awaited` before the `Pool` is ready and available to use. A bunch of safety features are in place in the `PoolManager` to avoid race conditions or when accidentally trying to request a `Pool` multiple times in parallel, but practically at the same time. But keep an eye on the logs, as they will inform you if something described as such would happen.

During creation of a `Pool` a container specifically assigned to this `Pool` will be added to the `DontDestroyOnLoad` scene in the hierarchy. Here all pooled objects can be found before they're either popped from the pool or after they've been pushed back.

Now you have a reference to the instance of your `Pool`! To pop an object from your `Pool` simply call `pool.Pop<MyMonoBehaviour>(<PoolConfigNames.PooledPrefabName>, transform)`, where `PoolConfigNames` would be the generated static class from your `PoolConfig` and the `PooledPrefabName` is a static string used to find a specific prefab in your `Pool`. And `transform` is the (optional) parent this instance should be attached to in the hierarchy.

After calling this line you'll have an active instance of `MyMonoBehaviour` which is naturally attached to a `GameObject`. You can do with it as you please.

It's recommended to keep a list of active `MyMonoBehaviours` so they can be pushed back in the pool once needed.

To push objects back to the `Pool`, simply call `pool.Push(myMonoBehaviourInstance)`, and remove it from any potential lists or array you've been using to keep track of your objects. This will cause the instance of `MyMonoBehaviour` to be pushed back into the pool and it's `GameObject` will be deactivated. Note that the parent of it's `GameObject` is also set back to the initial pool container in the hierarchy.

IPoolable

One could opt to use the `IPoolable` interface for any `MonoBehaviour` that's added to a `PoolConfig`. This will help knowing when the pooled object was either popped from the pool, via the `OnPop` implementation, or when the pooled object was pushed back into the pool, via the `OnPushed` implementation.

This is a safer approach compared to using i.e. `OnEnable` or `OnDisable`, as during your object's life cycle it might be deactivated but not pushed back in a pool.

Examples

A fairly simple example can be found in the `Demo` folder in this package.

To make the example work: - Go to Assets -> CardboardCore -> Pooling -> Runtime -> Demo -> Configs in your project and select the DemoPoolConfig asset - Check the "Addressable" tick and name it "DemoPoolConfig" - Run the PoolingDemo scene found in Assets -> CardboardCore -> Pooling -> Runtime -> Demo -> Scenes - Press "1" to pop a cube from a pool - Press "2" to pop a sphere from a pool - Press "backspace" to push all cubes and spheres back in the pool

For more advice or feedback, feel free to get in touch via the email address given on the CardboardCore publisher page: https://assetstore.unity.com/publishers/54078