User Manual

Contents

1. Getting Objects with Pooling

Since pooling is a technical subject, this package reduces its footprint in the Unity Editor as much as possible. There are no prefab lists and no manager objects required. Instead, replace your existing `Instantiate` calls with calls to `Pooling.GetObject`.

Don't forget to import the proper namespace using `using ThirteenPixels.Pooling;`.

```
[SerializeField]
private GuidedMissile guidedMissilePrefab;
private GuidedMissile guidedMissileInstance;

private void Shoot()
{
    guidedMissileInstance = Pooling.GetObject(guidedMissilePrefab,
                                              transform.position,
                                              transform.rotation);
    guidedMissileInstance.Accelerate(transform.forward);
}
```

As pooling works, this method will recycle an instance of the prefab referenced by `guidedMissilePrefab` if available. If there are no instances in the pool, a new one will be created and returned.

2. Returning Objects to their Pool

Instead of destroying an object, it must be returned to the pool it was taken from. There are two ways to this. Which way to choose depends on the following question.

> **Who** will put the object back into the pool?
>
> a) Only the same object that spawned it.
>
> b) Potentially any other object.

If the answer is a), the object that requests an instance from the pool can use the prefab reference to call `Pooling.ReturnObject`.

```
public void ExplodeMissile()
{
    Pooling.ReturnObject(guidedMissilePrefab, guidedMissileInstance);
}
```

Often, the answer is b) as an instance destroys itself or can be destroyed by some other object in the scene. In this case, the prefab must be modified by adding a **PoolReturner** component to it. This component remembers the pool it was taken from and offers methods to return it there.

The following is an example for a component that uses a PoolReturner attached to the same prefab to destroy itself.

```
using UnityEngine;
using ThirteenPixels.Pooling;

[RequireComponent(typeof(PoolReturner))]
public class GuidedMissile : MonoBehaviour
{
    private PoolReturner poolReturner;

    private void Awake()
    {
        poolReturner = GetComponent<PoolReturner>();
    }

    private void OnCollisionEnter()
    {
        // TODO Apply damage in area
        poolReturner.Return();
    }
}
```

In addition to `Return`, the PoolReturner component also offers a method called `ReturnAfter` that allows you to return the object after a delay in seconds.

3. Custom Behavior for Pooling Instances

By default, the `Pooling` class will use `GameObject.SetActive` to entirely disable a GameObject that is returned to its pool, and to reactivate it when taken out again. This works well for everything, but while it's faster than `Instantiate` and `Destroy`, it's still not perfect in terms of performance.

In addition to optimization, some prefab types require some resetting after being recycled. For example, the velocity of a Rigidbody should be reset. Otherwise, it will still have the velocity it had before returning it to the pool.

For both resetting and optimization on a per-prefab basis, use the `IPoolingBehaviour` interface. Any script on your prefab can implement it to **override** the default `GameObject.SetActive` behavior, including the component using the PoolReturner.

```
using UnityEngine;
using ThirteenPixels.Pooling;

public class PoolingBehaviorRigidbodyReset : MonoBehaviour, IPoolingBehaviour
{
    public void OnPoolingSetActive(bool active)
    {
        rigidbody.velocity = Vector3.zero;
        gameObject.SetActive(active);
    }
}
```

Note that this component manually calls `GameObject.SetActive` as it's not overriding the default behavior, but rather adds an additional action to perform.

`IPoolingBehaviour.OnPoolingSetActive` is called both when taking an instance from the pool and when returning it, with the `active` parameter being `true` when taking it out.

The following is an example for using `IPoolingBehaviour` for creating custom pooling behavior, replacing `GameObject.SetActive`.

```
using UnityEngine;
using ThirteenPixels.Pooling;

public class PoolingBehaviourPositioning : MonoBehaviour, IPoolingBehaviour
{
    public void OnPoolingSetActive(bool active)
    {
        if (!active)
        {
            transform.position = Vector3.down * 1000000;
        }
    }
}
```

This component just places the object very far down upon being returned to the pool – out of the active area of the scene. Of course, a behavior like this is desirable only in rare cases and thus is a rather extreme example. However, wherever you can implement such behavior, it will make pooling even faster!

4. (Pre-)Filling Pools

The `Pooling` class offers methods to fill pools with instances before they are needed. You can use this at the start of a scene to reduce hiccups during the game.

`Pooling.CreateObjects(prefab, amount)` creates the given `amount` of instances of the given `prefab` and stores them in the prefab's pool. This can be used to allow every object to create as many instances as it assumes it will need at once at any given time.

`Pooling.Fill(prefab, target)` creates as many instances as needed to get to a count of `target` instances in the pool. If you predict that multiple objects will probably not drain the pool they collectively use, use this method.

Of course, using these methods is connected to estimating runtime behavior. Luckily, it's not necessary to make perfect predictions, as the system will just create more instances when needed.

5. Event Order

When taking an object out of a pool, the following events occur in the listed order.

1. `Awake` is called if the object was freshly created.
2. `OnEnable` is called.
3. `IPoolingBehaviour.OnPoolingSetActive(true)` is called.
4. Any code after the `Pooling.GetObject` call is run.
5. `Start` is called if the object was freshly created.

This order is the same for objects created while calling `Pooling.GetObject` as well as during pre-filling.

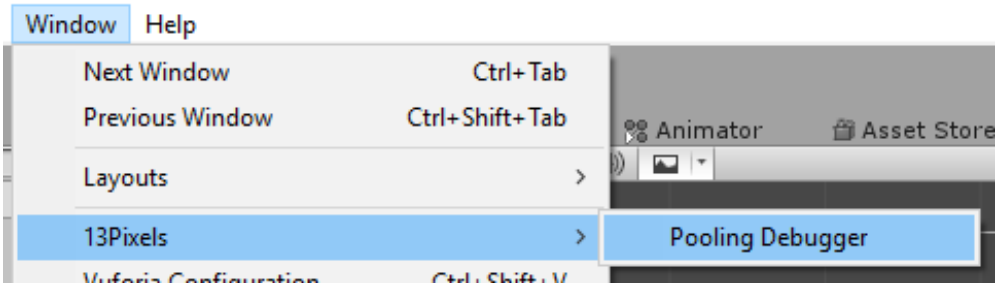
When returning an object to its pool, these events occur in the listed order.

1. `IPoolingBehaviour.OnPoolingSetActive(false)` is called.
2. `OnDisable` is called.

Note that `OnEnable` and `OnDisable` are only called if the components or the entire GameObject are deactivated and reactivated as part of the pooling behavior.

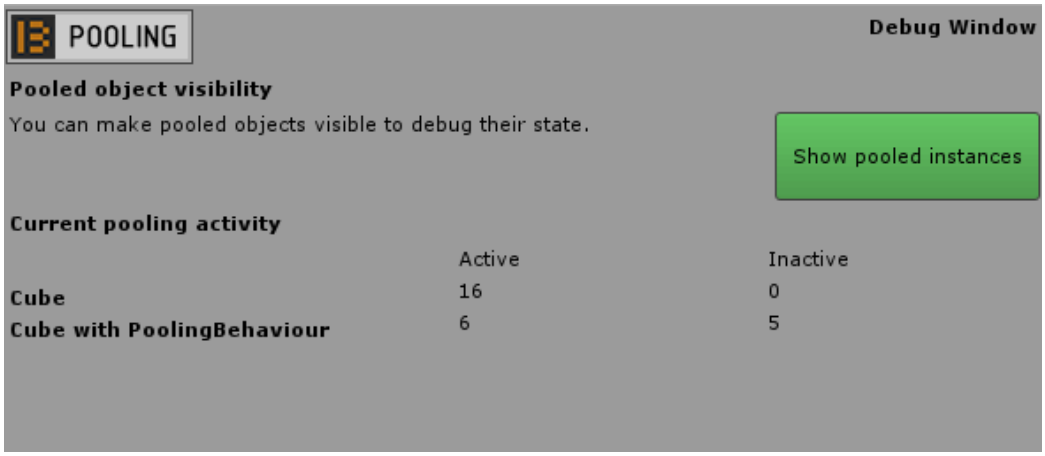
6. The Debug Window

While in the Unity Editor, you can debug the behavior of both the pooling system and your pooled objects by using the **Debug Window**. Open it through the "Window" menu:



You will see a Debug Window that can be dragged around and added to your editor layout. It displays all currently existing pools and how many active and inactive objects they have.

By default, all objects instantiated by the pooling system will be hidden in the hierarchy. If you want to inspect a pooled instance for debugging purposes, you can make them visible with the green button located in the Debug Window.



7. Support

If you have questions, issues, proposals or want to get to know other developers using 13Pixels Pooling, feel free to join the 13Pixels slack workspace: 13pixels.de/slack.

Alternatively, you can write a mail to assetstore@13pixels.de.

Have a productive time!