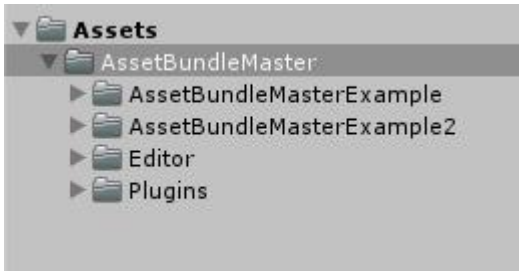
Before we start use it, you'd better import it to an empty project, following the ReadMe to learn this framework step by step, it may take 2 hours.

Let's see the folder structure :



It has 4 folders in the AssetBundleMaster folder:

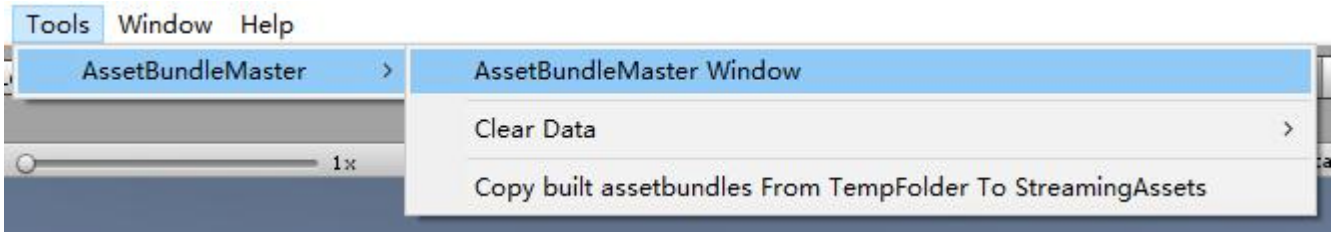
[AssetBundleMasterExample] is Demo for leaning basical APIs, It include load assets, load scenes, and how to do unload.

[AssetBundleMasterExample2] is Demo for testing how Built-In Shader multi compiled and how AssetBundleMaster to optimize it.

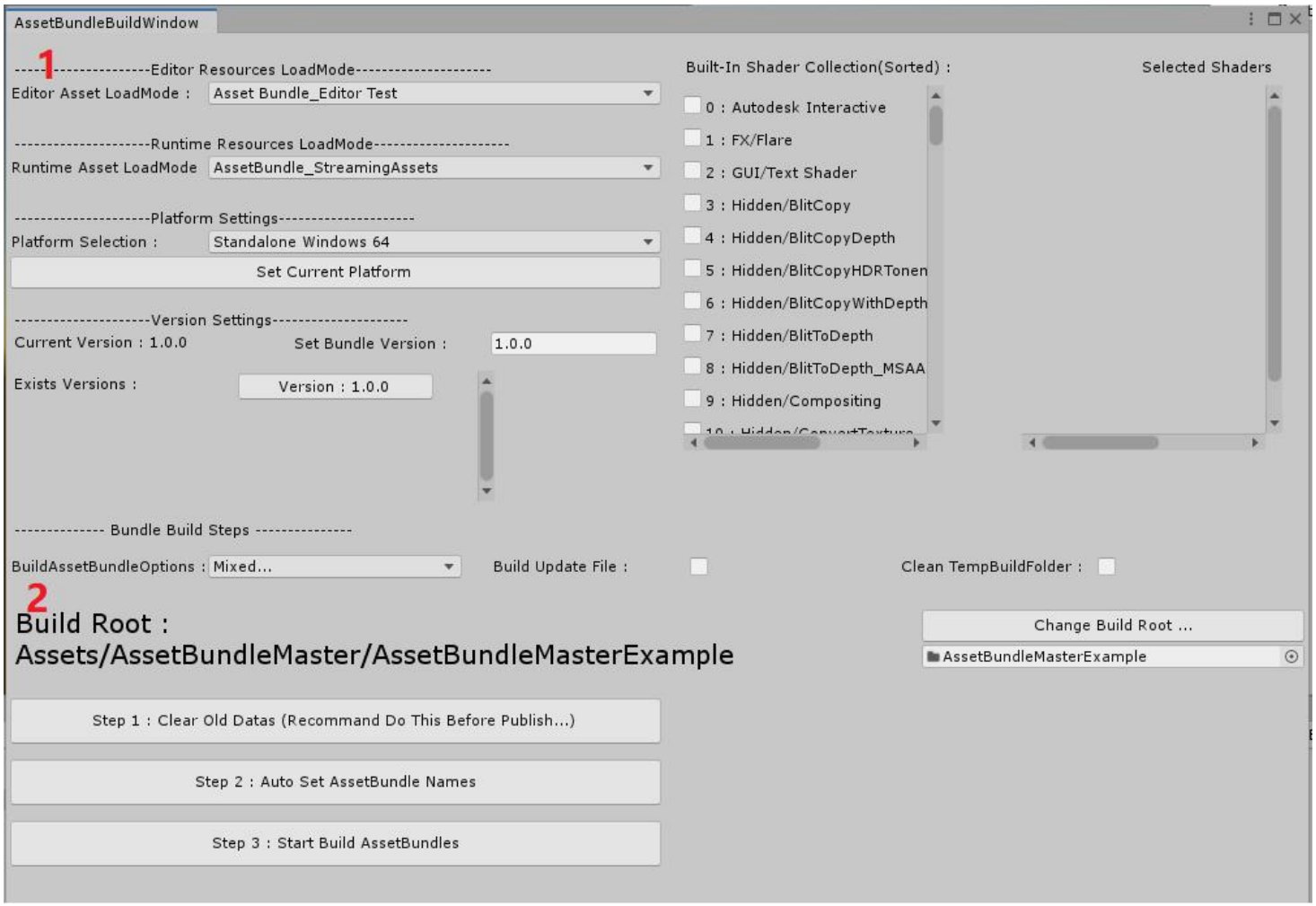
[Editor] is editor scripts.

[Plugins] is runtime scripts, asset load is basic logic for a project, so we put it in Plugins.

Let's try to run API Demo scenes in Editor Mode, open AssetBundleMaster Window first :



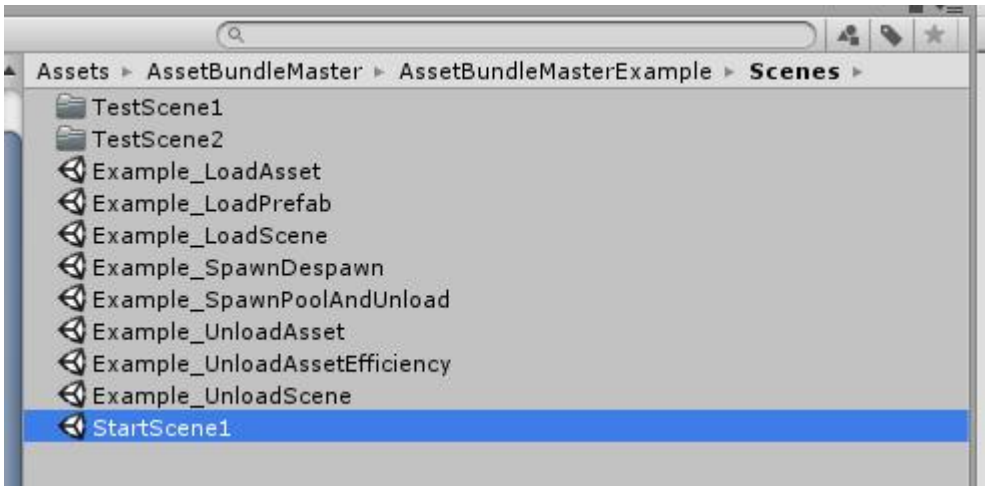
window shows :

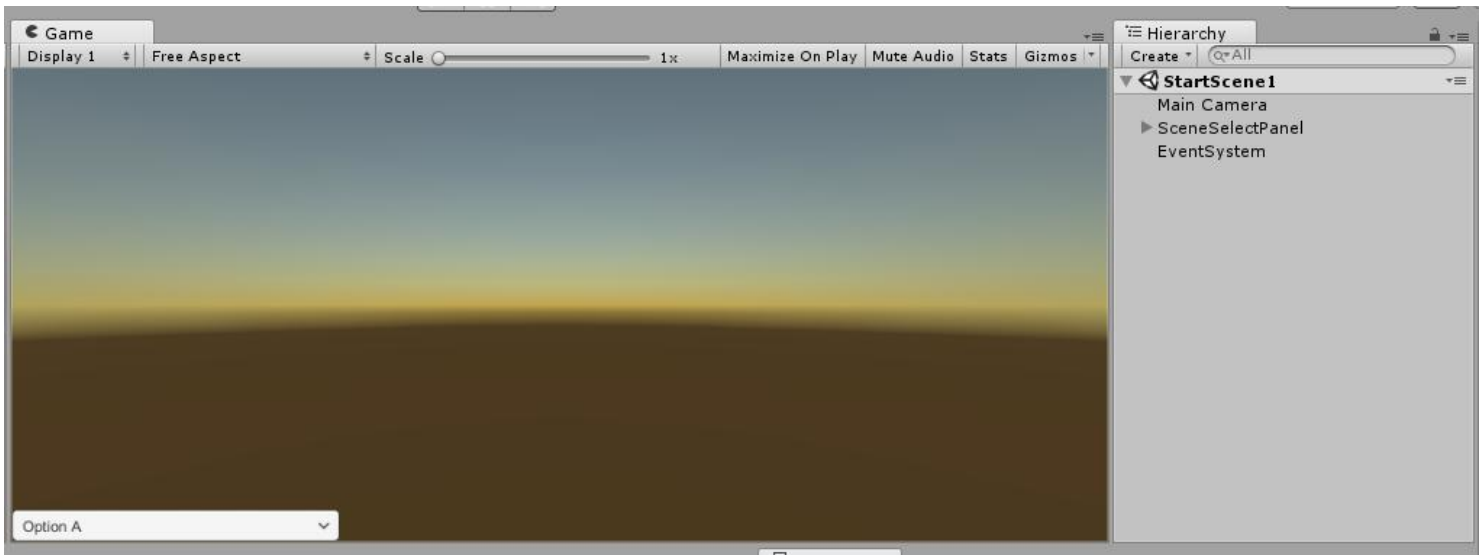


We need to set only 2 variables can make it runable in Editor, developer friendly, ignor other options now :

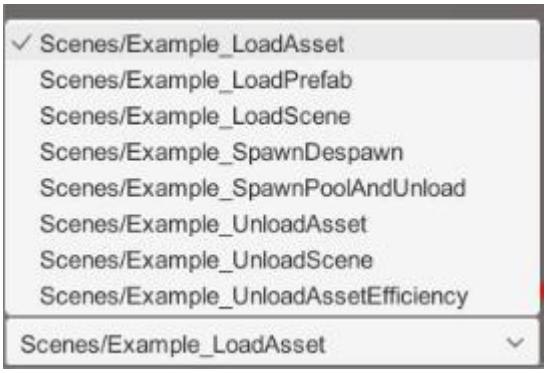
1. Editor Asset LoadMode : load mode in Editor, select AssetDataBase_Editor, it means load asset directly.

2. Build Root : It is the root folder for loading assets, click [Change Build Root...] to change the folder, set to AssetBundleMasterExample folder so that we can run API Demo scenes. The entry scene is StartScene1, you can find it in AssetBundleMasterExample/Scenes folder:

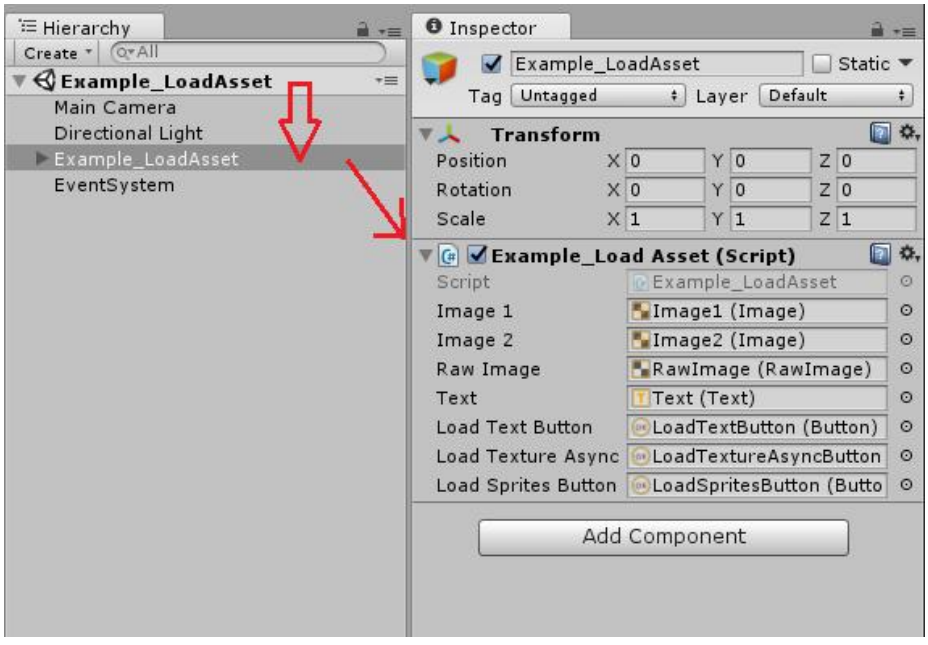




Run StartScene1, the bottom-left shows a dropdown for selecting demo scenes :

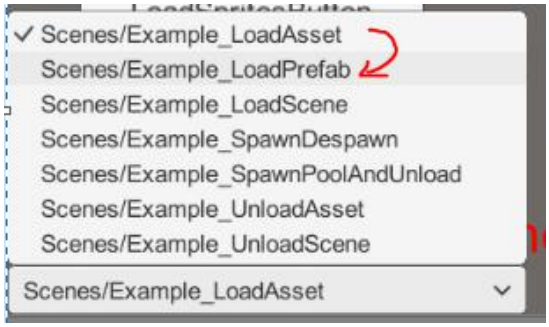


These are all the API demo scenes, select any scene, you can see the hierarchy that a GameObject has the same name with the scene, and the same named script attached on it, it is the demo script.

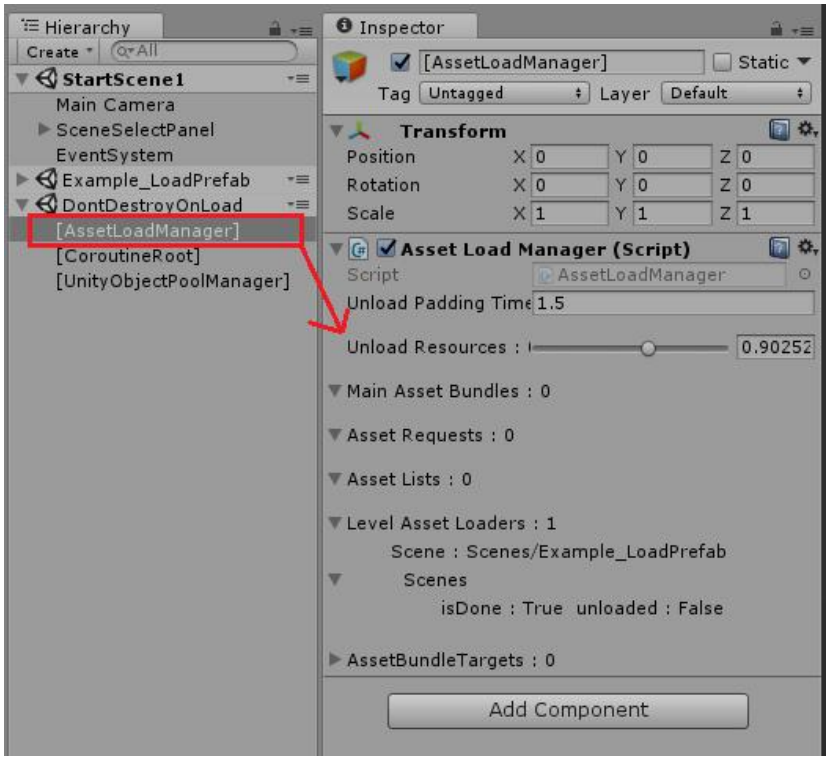


We must explain how the scene load process working, when we select any scene it will unload all assets loaded before, so that any asset will be loaded in the new scene will be new load, this is convenient for testing asset load and unload, you can open

Profiler window to check all the time. The unload process has a padding time when you switch the scene. You can see Hierarchy select [AssetLoadManager] GameObject and the inspector will shows it :



Change the scene, then you can see [AssetLoadManager] Inspector has a count down Slider shows unload padding left time:

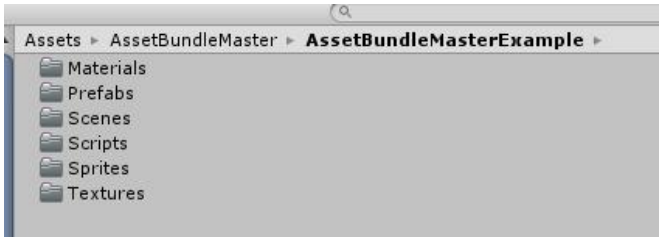


If you use Profiler : Memory > Detailed > Take Sample to check assets in memory, check after the unload process done.

Let's see every demo scene and how to use APIs, we set the Build Root path, so all the asset load paths are relative path from it the assets in this folder are called main assets that commonly should be load by your script, like Scenes, and assets not in this folder called referenced assets like Meshes that will not be loaded directly by your code, so you don't need to put all assets into the Build Root. keep this in mind.

PS : when using AssetDataBase_Editor mode, it has no asynchronous loading, all asset loading will be sync load because we use AssetDataBase.LoadAssetAtPath as Editor load function. The unload logic also diferent with AssetBundle mode, while AseetBundle mode using AssetBundle.Unload() and Resources.UnloadUnusedAssets() API.

You can check what kinds of assets we have in the AssetBundleMasterExample folder.



After all we can load assets in Editor, not so much things to learn, if you want it work for your project, only to set Build Root to your asset folder.

Now you can see AssetBundleMaster generate a file in your Resources Folder, this is the basic settings of AssetBundleMaster, you can upload it for other teammates can use it :

Let's start learn how to use API by Demo scenes. How to create AssetBundle written at the end.

0. StartScene1 / StartScene2 (Script : StartScene.cs)

These are 2 entry scene for all demo testing, normally you don't need to do any thing before use AssetBundleMaster APIs, but the only one exception is Remote AssetBundle mode that you load assetbundle from a remote address(Server), you have to wait the AssetLoadManager to download basical files from server, you can see the code in StartScene.cs::Start() function :

```
        void Start()
        {
#if DEVELOPMENT_BUILD
            Application.runInBackground = false;
#else
            Application.runInBackground = true;
#endif

            if(dropdown)
            {
                dropdown.ClearOptions();
                dropdown.AddOptions(loadScenePaths);

                AssetLoadManager.Instance.minUnloadAssetCounter = 0;    // Set mi
                AssetLoadManager.Instance.unloadPaddingTime = 3f;     // Set unlo
                AssetUnloadManager.Instance.maxUnloadPerFrame = 1f;

                // This Is The Game Entry Point!!!
                // if remote-assetbundle mode, we must wait for the AssetLoadMana
                AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
                {
                    // start your game logic after AssetLoadModuleInited
                    dropdown.onValueChanged.AddListener(LoadLevel);
                    LoadLevel(0);
                });
            }
        }
```

API (namespace AssetBundleMaster.AssetLoad):

AssetLoadManager.Instance.OnAssetLoadModuleInited(System.Action callback)

You can make it as your game Entry Point, start game logic after AssetLoadManager.Instance.OnAssetLoadModuleInited() callback. This is the only requirement for Remote AssetBundle mode, and in other mode you can use it also, even the mode don't need any waiting.

1. Example_LoadAsset (Script : Example_LoadAsset.cs)

This Demo shows how to load common assets, loading with ResourceLoadManager .


```
// Use this for initialization
void Start()
{
    LoadTextButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadTextButton Start Load at frame : " + Time.frameCount);

        // Load Single Asset Sync
        var text = ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1");
        if(text)
        {
            Text.text = text.text;
            Debug.Log(Text.text + " Loaded at frame : " + Time.frameCount);
        }
    });

    LoadTextureAsyncButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadTextureAsyncButton Start Load at frame : " + Time.frameCount);

        // Load Single Asset Async
        ResourceLoadManager.Instance.LoadAsync<Texture2D>("Textures/Pic3", (_tex) =>
        {
            RawImage.texture = _tex;
            Debug.Log(RawImage.texture + " Loaded at frame : " + Time.frameCount);
        });
    });

    LoadSpritesButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadSpritesButton Start Load at frame : " + Time.frameCount);

        // load asset with ext name
        var sp1 = ResourceLoadManager.Instance.Load<Sprite>("Sprites/Pic1.tga");
        if(sp1)
        {
            Image1.overrideSprite = sp1;
            Debug.Log(Image1.overrideSprite + " Loaded at frame : " + Time.frameCount);
        }

        var sp2 = ResourceLoadManager.Instance.Load<Sprite>("Sprites/Pic1.png");
        if(sp2)
        {
            Image2.overrideSprite = sp2;
            Debug.Log(Image2.overrideSprite + " Loaded at frame : " + Time.frameCount);
        }
    });

    StartScene.RegisterClickFocus();
}
```

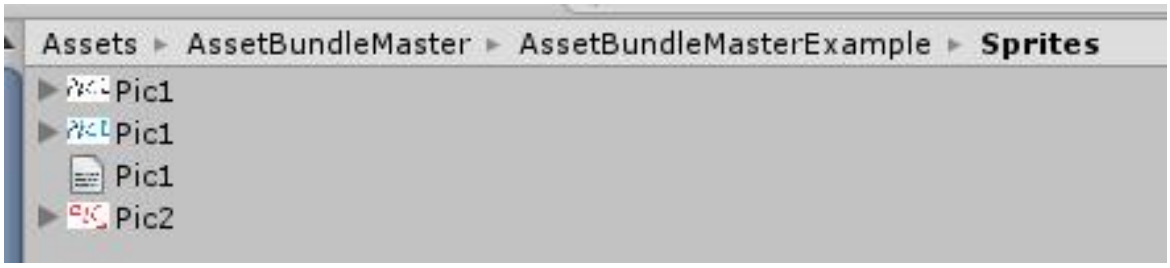
API (namespace AssetBundleMaster.ResourceLoad):

ResourceLoadManager.Instance.Load<T>(string loadPath)

ResourceLoadManager.Instance.LoadAsync<T>(string loadPath, System.Action<T> loaded)

ResourceLoadManager.Instance.LoadAll<T>(string loadPath)

This Demo will load assets from [Sprites] and [Textures] folder, the path "Sprites/Pic1" is kind of special, let's see :



The [Sprites] folder contains 3 assets have the same asset name Pic1.tga / Pic1.png / Pic1.txt (without extention), in this situation our asset load API use no ext name path to load assets(like Resources.Load API), so these 3 assets have the same load path, the API can load asset by limit its type, for example ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1"); load the first TextAsset in the path, and ResourceLoadManager.Instance.LoadAll<Sprite>("Sprites/Pic1"); load all Sprites from this path, if you change "Sprites/Pic1" to "Sprites" that targeting the folder, it will also load asset correctly, the logic is the same as UnityEngine.Resources load API. This feature solved the problem when in Android load from StreamingAssets that has no way to load all assets from target folder.

If you load asset by path without extension, and the file name is not unique like Pic1 above, the asset loader will load assets iteratively until target asset is loaded. that's unnecessary overhead. If a load path with extension the path must be unique, so we enhanced the load APIs, support the load path with extension :

ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1").text;

can be rewrite to :

ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1.txt").text;

We don't have any load overhead now. you should use load path with extension when the asset name is not unique while loading a single asset for preventing the overhead.

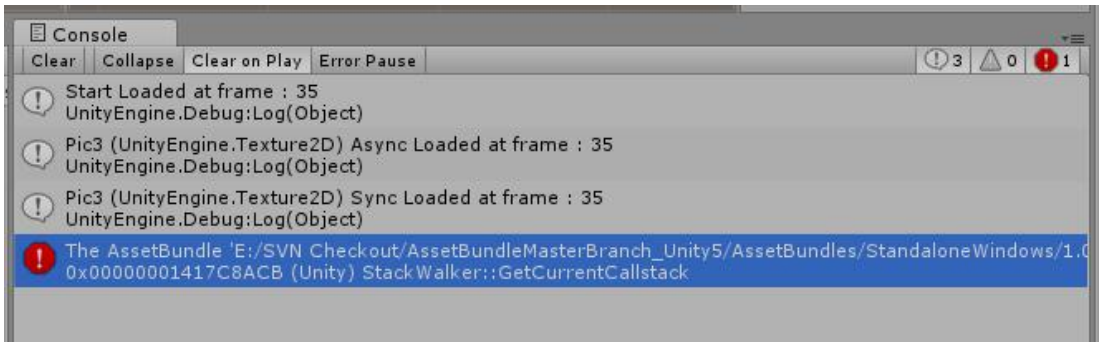
Tips for AssetBunlde load mode : if you are loading asset async, and a sync requst happened while the asset is still loading, the Console will pop an error message, but the function still work correctly. see what happened:

```
void AsyncAndSync()
{
    Debug.Log("Start Loaded at frame : " + Time.frameCount);

    ResourceLoadManager.Instance.LoadAsync<Texture2D>("Textures/Pic3", (_tex) =>
    {
        RawImage.texture = _tex;

        Debug.Log(RawImage.texture + " Async Loaded at frame : " + Time.frameCount);
    });

    var tex2D = ResourceLoadManager.Instance.Load<Texture2D>("Textures/Pic3");

    Debug.Log(tex2D + " Sync Loaded at frame : " + Time.frameCount);
}
```

1. Request load "Textures/Pic3" Async

2. Request load "Textures/Pic3" Sync before Async is not yet completed

Load Sync while Async loading, this code was not written in demo, just for tips.



This is because Unity has no provide any API for stop an AssetBundle async load, and load sync while async loading will pop error, the AssetBundleMaster's Low-Level API handles it to ensure all call backs and returns working correctly. you don't need to worry about it.

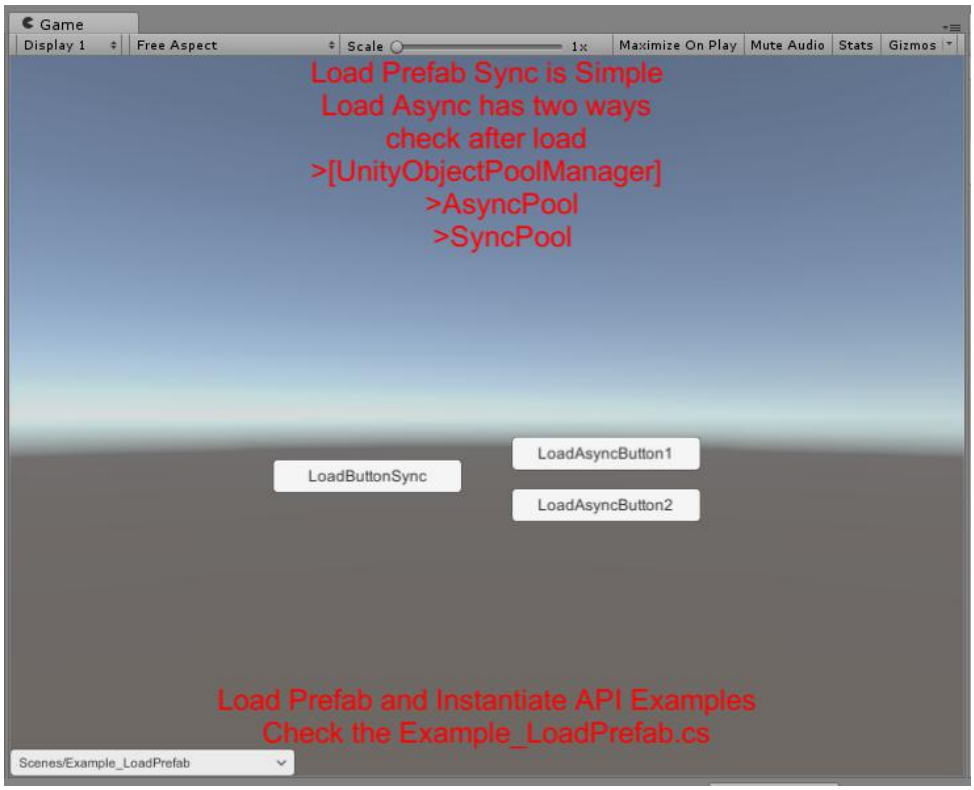
Other APIs can be check in ResourceLoadManager.cs, include Generic and Non-Generic version.

```
#region Main Funcs
/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public UnityEngine.Object Load(string loadPath, System.Type systemTypeInstance)...
/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public UnityEngine.Object Load(string loadPath)...
/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public T Load<T>(string loadPath) ...

/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
public UnityEngine.Object[] LoadAll(string loadPath, System.Type systemTypeInstance)...
/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
public UnityEngine.Object[] LoadAll(string loadPath)...
/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
public T[] LoadAll<T>(string loadPath) ...

/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public void LoadAsync(string loadPath, System.Type systemTypeInstance, System.Action<UnityEngine.Object> loaded)...
/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public void LoadAsync(string loadPath, System.Action<UnityEngine.Object> loaded)...
/// <summary> Load asset from a path. Notice :    if in Resources mode, the load ...
public void LoadAsync<T>(string loadPath, System.Action<T> loaded) ...

/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
private void LoadAllAsync(string loadPath, System.Type systemTypeInstance, System.Action<UnityEngine.Object[]> loaded)...
/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
public void LoadAllAsync(string loadPath, System.Action<UnityEngine.Object[]> loaded)...
/// <summary> Load all assets from a path. Notice :    if pass a directory path, ...
public void LoadAllAsync<T>(string loadPath, System.Action<T[]> loaded) ...

/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync(string loadPath, System.Type systemTypeInstance)...
/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync(string loadPath)...
/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync<T>(string loadPath) ...

/// <summary> Unload Asset with target type and load path, Recommand use this AP ...
public void UnloadAsset(string loadPath, System.Type systemTypeInstance, bool inherit = false)...
/// <summary> Unload Asset with target type and load path, generic version
public void UnloadAsset<T>(string loadPath, bool inherit = false) ...
/// <summary> Unload All Assets
public void UnloadAllAssets()...
#endregion
```

Tool tips : In this version we added code generate tools for you, you don't need to coding the load/unload script, see how to use it in the Appendix.

2. Example_LoadPrefab (Script : Example_LoadPrefab.cs)

This demo shows how to load and Instantiate GameObjects. load with PrefabLoadManager. the sctips shows 2 ways to Instantiate GameObject async, with a little overhead diference.




```
void Start()
{
    LoadButtonSync.onClick.AddListener(() =>
    {
        for(int i = 0; i < spawnCount; i++)
        {
            var cube = PrefabLoadManager.Instance.Spawn(LoadPath, "SyncPool");  // spawn Sync
            RandomCubePos(cube);
        }
    });

    /*
    * These are two kinds of Async spawn way, The first (LoadAsyncButton1) is take a little overhead GC,
    * ortherwise 2nd (LoadAsyncButton2) is less GC, see what is diferent
    */
    LoadAsyncButton1.onClick.AddListener(() =>
    {
        Debug.Log("Start Load at frame : " + Time.frameCount);
        for(int i = 0; i < spawnCount; i++)
        {
            // if Asset not yet loaded, this will call multi times the asset load Low-Level API
            PrefabLoadManager.Instance.SpawnAsync(LoadPath, (_cube) =>
            {
                RandomCubePos(_cube);
                Debug.Log("Loaded at frame : " + Time.frameCount);

            }, "AsyncPool");
        }
    });

    LoadAsyncButton2.onClick.AddListener(() =>
    {
        Debug.Log("Start Load at frame : " + Time.frameCount);
        // only call Low-Level API Once, wait until asset loaded
        PrefabLoadManager.Instance.LoadAssetToPoolAsync(LoadPath, (_prefab, _pool) =>
        {
            for(int i = 0; i < spawnCount; i++)
            {
                var cube = _pool.Spawn(LoadPath, _prefab);      // you got the pool holds the Prefab
                RandomCubePos(cube);
                Debug.Log("Loaded at frame : " + Time.frameCount);
            }

        }, "AsyncPool");
    });
}
```

API (namespace AssetBundleMaster.ResourceLoad):

PrefabLoadManager.Instance.Spawn(string loadPath, string poolName = null, bool active = true) // Spawn is to load and Instantiate GameObject.

PrefabLoadManager.Instance.SpawnAsync(string loadPath, System.Action<GameObject> loaded = null, string poolName = null, bool active = true) // SpawnAsync is to load Async and Instantiate GameObject.

PrefabLoadManager.Instance.LoadAssetToPoolAsync(string loadPath, System.Action<GameObject, GameObjectPool> loaded = null, string poolName = null) // LoadAssetToPoolAsync is to load prefab asset Async to target pool.

The script shows 2 Async load logic, if the prefab asset is not yet loaded, the second one is more efficiency than the first one, but is more coding work(See scope in LoadAsyncButton2.onClick). the overhead of PrefabLoadManager.Instance.SpawnAsync is acceptable. You should always load and instantiate GameObject with PrefabLoadManager (Character, UI, Particle Effect, Weapon...etc, all your GameObject Prefabs), you are not only get benefit from ObjectPool, but also get auto resources control. you can also use LoadAssetToPoolAsync as a pre-load like this :

PrefabLoadManager.Instance.LoadAssetToPoolAsync("Prefabs/Cube", null, "Characters");

This is only load Cube asset to pool, as you know the asset load process causes I/O, Decompress, Deserialize, Shader Parse....ect, do pre-load while CPU is not busy can have a better game experience.

The auto resource contrl logic can be check in the appendix.

you can use file extension also.

3. Example_LoadScene (Script : Example_LoadScene.cs)

This demo shows how to load scenes, the scene load is like asset load which using relative load path. Let's see the scene Example_LoadScene.


```
    public const string SceneLoadPath1 = "Scenes/TestScene1/TestScene";
    public const string SceneLoadPath2 = "Scenes/TestScene2/TestScene";

    private List<int> m_loadedScenes = new List<int>();
```

```
void LoadOneScene(string loadPath, AssetLoad.LoadThreadMode mode)
{
    Debug.Log("Start Load scene[" + loadPath + "] at frame : " + Time.frameCount);

    var id = SceneLoadManager.Instance.LoadScene(loadPath,
        mode,
        UnityEngine.SceneManagement.LoadSceneMode.Additive,
        (_id, _scene) =>
        {
            Debug.Log("Scene [" + _scene.path + "] Loaded at frame : " + Time.frameCount);
        });
    m_loadedScenes.Add(id);
}
```

```
LoadSceneSyncButton1.onClick.AddListener(() =>
{
    LoadOneScene(SceneLoadPath1, AssetLoad.LoadThreadMode.Synchronous);
});
LoadSceneAsyncButton2.onClick.AddListener(() =>
{
    LoadOneScene(SceneLoadPath2, AssetLoad.LoadThreadMode.Asynchronous);
});
```

API (namespace AssetBundleMaster.ResourceLoad):

SceneLoadManager.Instance.LoadScene(string sceneLoadPath,

AssetBundleMaster.AssetLoad.LoadThreadMode loadMode =
AssetBundleMaster.AssetLoad.LoadThreadMode.Asynchronous,

UnityEngine.SceneManagement.LoadSceneMode loadSceneMode = LoadSceneMode.Single,

System.Action<int, Scene> loaded = null)

It do load scene like an asset using its relative load path, can load with async/sync mode and LoadSceneMode. Please notice it that load scene must wait for the call back UnityEngine.SceneManagement.SceneManager.sceneLoaded that means no matter load scene Sync or Async, you'd better wait for loaded call back.

It returns a HashCode as handler of this scene, and the callback [System.Action<int, Scene> loaded = null] is also give you the same HashCode.

4. Example_SpawnDespawn (Script : Example_SpawnDespawn.cs)

This demo shows how the objectpool do spawn and despawn GameObjects.

API (namespace AssetBundleMaster.ResourceLoad):

PrefabLoadManager.Instance.Spawn(string loadPath, string poolName = null, bool active = true);

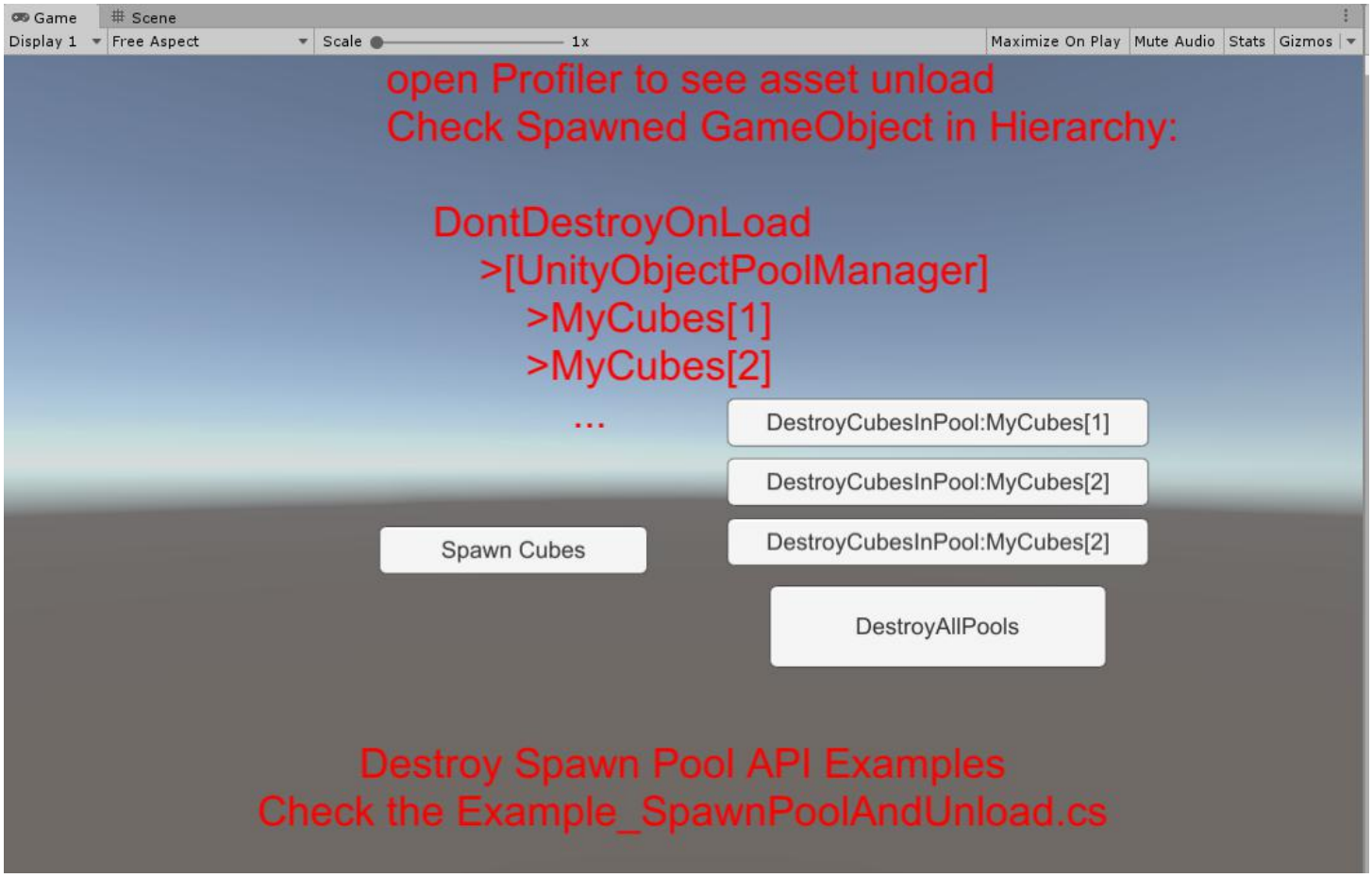
PrefabLoadManager.Instance.Despawn(GameObject go, string poolName = null);

We do auto load and Instantiate prefab, and set active when spawn a GameObject, with the poolName parameter, you must despawn GameObject to the same poolName where it spawned from. The GameObject will be set active false while despawn to pool, and its parent will also set to the pool, if you have spacial needs, rewrite the code at AssetBundleMaster.ObjectPool.GameObjectPool.Despawn function.

PS : if you call the PrefabLoadManager.Instance.Despawn(GameObject go, string poolName = null); poolName is incorrect, it will search all pools to find the correct pool where the gameobject is spawned from, if you have many pools you'd better fill the correct poolName for less search. else if you don't fill any poolName, it will use a default pool name.

5. Example_SpawnPoolAndUnload (Script : Example_SpawnPoolAndUnload.cs)

This deom shows how to destroy spawn pool, and how's the auto unload working.



```
// Use this for initialization
void Start()
{
    SpawnButton.onClick.AddListener(() =>
    {
        CreateCube(LoadPath, PoolName1, 12);
        CreateCube(LoadPath, PoolName2, 15, m_caches2);
        CreateCube(LoadPath, PoolName3, 10, m_caches3);
    });

    // how to destroy spawned targets
    DestroyPoolButton1.onClick.AddListener(() =>
    {
        PrefabLoadManager.Instance.DestroyTargetInPool(LoadPath, PoolName1, true);
    });
    DestroyPoolButton2.onClick.AddListener(() =>
    {
        foreach(var go in m_caches2)
        {
            PrefabLoadManager.Instance.DestroySpawned(go, PoolName2, LoadPath, true);
        }
        m_caches2.Clear();
    });
    DestroyPoolButton3.onClick.AddListener(() =>
    {
        foreach(var go in m_caches3)
        {
            PrefabLoadManager.Instance.DestroySpawned(go);
        }
        m_caches3.Clear();
    });

    // this will unload the asset
    DestroyPoolAndUnloadButton.onClick.AddListener(() =>
    {
        PrefabLoadManager.Instance.DestroyAllPools(true);
        StartScene.FocusAssetLoadManager();
    });
}
```

API (namespace AssetBundleMaster.ResourceLoad):

PrefabLoadManager.Instance.DestroyTargetInPool(string loadPath, string poolName = null, bool tryUnloadAsset = true);

PrefabLoadManager.Instance.DestroySpawned(GameObject target, string poolName = null, string loadPath = null, bool tryUnloadAsset = true);

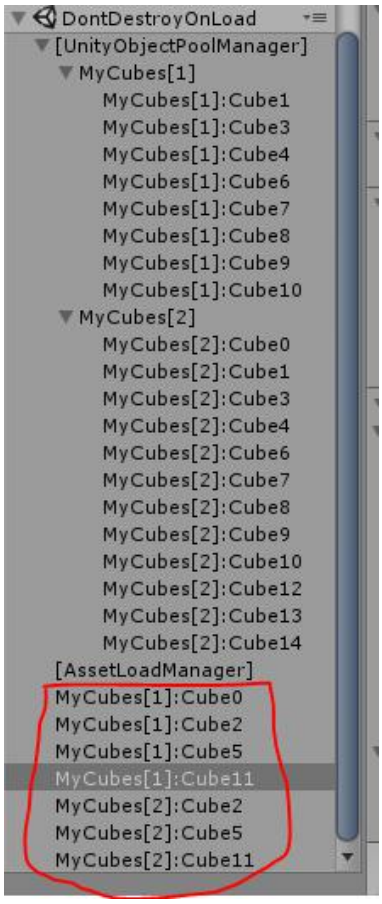
The first API (DestroyTargetInPool) is for you certainly know when to unload a target pool(means no need all spawned GameObject any more).

The second one (DestroySpawned) is for you don't want to control the pool, let it auto controlled. For example you load many nameplate UI for showing HP on mobs, use DestroySpawned to destroy whitch one no need to use any more(died, out of range ect...).

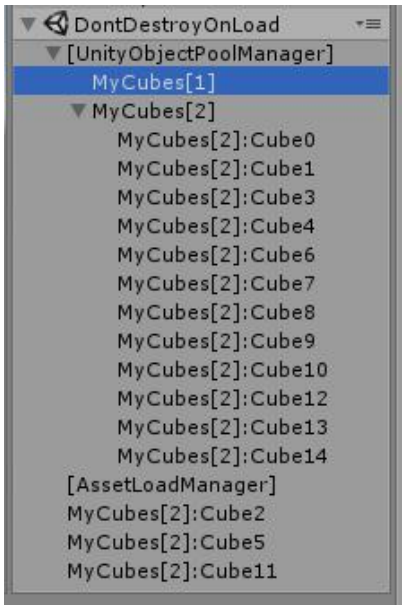
Both of them If tryUnloadAsset is true, PrefabLoadManager will do check referenced assets, if all pools do not reference to the prefab asset, it will do unload asset process later.

Notice : the DestroySpawned API you can ignor the poolName and loadPath parameters, but if you have many pools and many diferent prefab loaded in runtime, you'd better fill them correctly, otherwise it will search all the pools to find out where it spawned from.

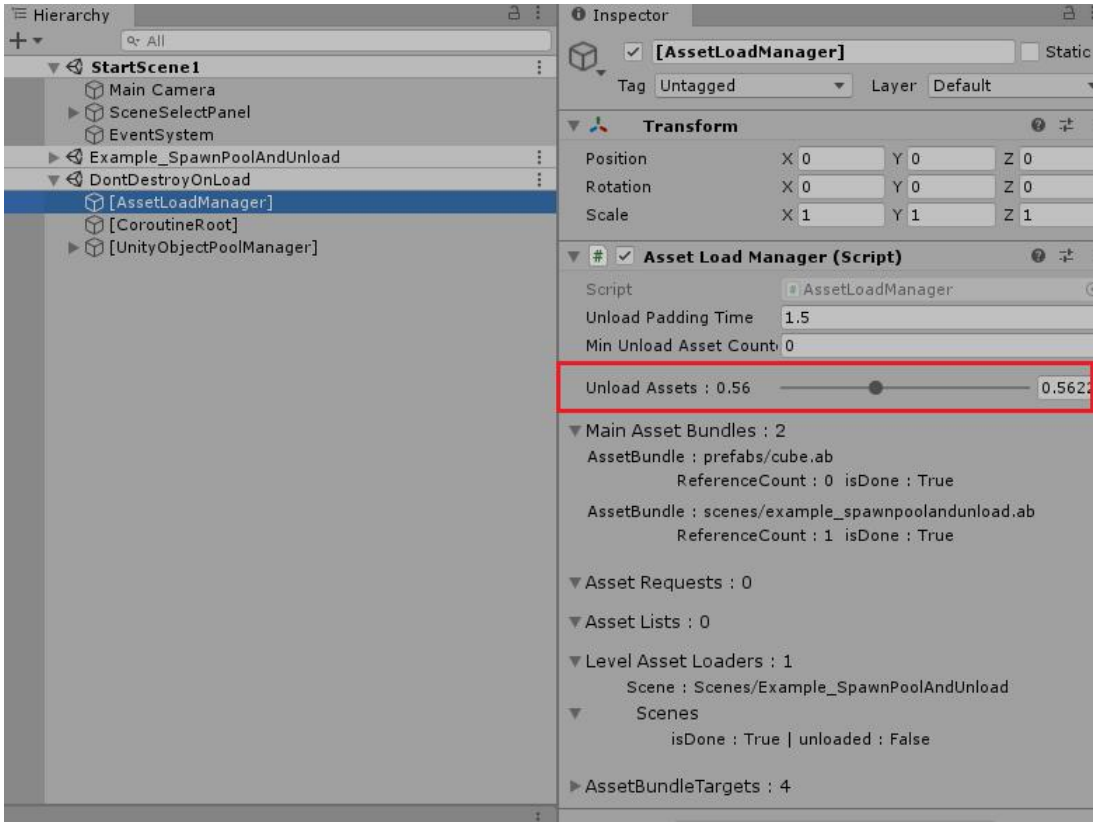
PS : click [Spawn Cubes] button in scene, there are 3 pools will be created, and every pool create some Cubes, see Hierarchy there are some Cubes was set no parent, it just for telling you that the connection of spawned GameObject to pool will not be break from parent changed. GameObject still controlled by pool even it is not child of the pool in Hierarchy.



click [DestroyCubesInPool:MyCubes[1]] button, the MyCubes[1] pool will destroy all Cubes controlled by it, we can see the pool of MyCobes[1] is alive but spawned cubes was destroied.



The asset(prefab) is still referenced by MyCubes[2] and MyCubes[3] pools, so it will not trigger the unload asset process, click [DestroyCubesInPool:MyCubes[2]] and [DestroyCubesInPool:MyCubes[3]] button, now all GameObjects was destroied, see [AssetLoadManager] inspector, unload asset was triggered.

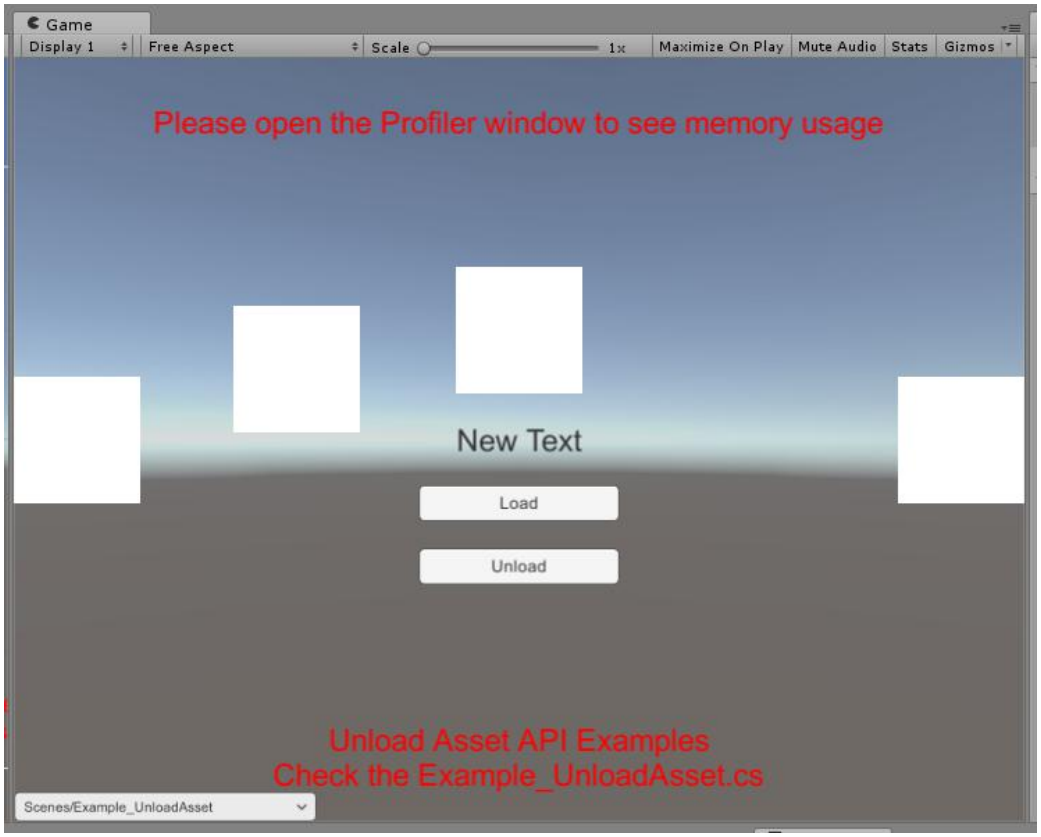


So that PrefabLoadManager can auto control the resources. every one just to control their pools to load and unload, the auto resource control will handle when to unload the assets.

PS : here we using the AssetDataBase_Editor mode, it will unload the assets depending on Resources.UnloadUnusedAssets(); while the AssetBundle mode will use AssetBundle.Unload(true); to unload assets (more about it see the appendix).

6. Example_UnloadAsset (Script : Example_UnloadAsset.cs)

This deom shows common asset unload process. the API is to unload asset :




```
UnloadButton.onClick.AddListener(() =>
{
    // Pic1 loaded Sprite and Text, unload type <Object> means all these types
    /* Notice : Here the TextAsset is also unloaded */
    ResourceLoadManager.Instance.UnloadAsset<Object>("Sprites/Pic1", true);     // ture means <T> is base type

    // Pic2 is loaded as Sprite only
    ResourceLoadManager.Instance.UnloadAsset<Sprite>("Sprites/Pic2.png", false);    // false means unload only type == <T>

    // Pic3 is loaded as Texture only, we can call unload like Resources.UnloadAsset
    ResourceLoadManager.Instance.UnloadAsset(RawImage.texture);

    // unload request is tick later, so you can clear reference after call UnloadAsset
    // Notick : unload request is not a force unload you should set reference to null
    Image1.overrideSprite = null;
    Image1_2.overrideSprite = null;
    Image2.overrideSprite = null;
    RawImage.texture = null;
});
```

API (namespace AssetBundleMaster.ResourceLoad):

ResourceLoadManager.Instance.UnloadAsset(UnityEngine.Object target); // recommand not use this

ResourceLoadManager.Instance.UnloadAsset(string loadPath, System.Type systemTypeInstance, bool inherit = false); // use this

ResourceLoadManager.Instance.UnloadAsset<T>(string loadPath, bool inherit = false); // use this

We have many ways to unload asset, you can call [ResourceLoadManager.Instance.UnloadAsset(UnityEngine.Object target)] like UnityEngine.Resources.UnloadAsset, or others the parameter is the load path + load type, this is because to load the same asset it may have diferent load types, for example a picture can be load as Texture2D or Object, so the unload logic is also the same. The unload type can be inherit, like the picture shows

above, ResourceLoadManager.Instance.UnloadAsset<Object>("Sprites/Pic1", true); means to unload all type inherit from Object that load from path "Sprites/Pic1", and the ResourceLoadManager.Instance.UnloadAsset<Sprite>("Sprites/Pic2", false); means only to unload Sprite type load from "Sprites/Pic2". these APIs were provided for your special need. UnloadAsset with Object target is not a good use, because the "Resource" you want to unload maybe incorrcet(for example you load a GameObject and get a texture from it, UnloadAsset(texture) is incorrect, you should unload the GameObject),

This API is privide a flexible asset unload logic that it do reduce the reference count of assets, if any asset reference count was set to 0, the unload process will be triggered. if you set all references of assets to null, the assets will be unloaded, but if not, the ResourceLoadManager holds weakreference to the assets too, and someone request the same asset next

time, ResourceLoadManager will return the weakreferenced target and no need to reload asset, this is the core logic to make sure asset will not duplicated in memory.

Unload procccess will be diferent by diferent asset load mode, the AssetDataBase_Editor is totally

use Resources.UnloadUnusedAssets(); and the AssetBundle mode may use AssetBundle.Unload(true) and Resources.UnloadUnusedAssets() it depends on the auto resource control logic.

If you do request unload asset by ResourceLoadManager, the AssetLoadManager will increase the unload counter number, if the number is greater than AssetLoadManager.Instance.minUnloadAssetCounter it will enter the unload process, finally will trigger the Resources.UnloadUnusedAssets(). you can adjust the value of it. AssetLoadManager have only 2 variables can be access you can see StartScene.cs::Start() function :

1. AssetLoadManager.Instance.minUnloadAssetCounter : the minimal unload number to enter unload process. change this value for your needs.

2. AssetLoadManager.Instance.unloadPaddingTime : the padding time between unload process started and unload function called, this is to reduce the unload-reload happening times, in runtime the asset load / unload will happened often, so you don't have to mind it. change this value for your needs.

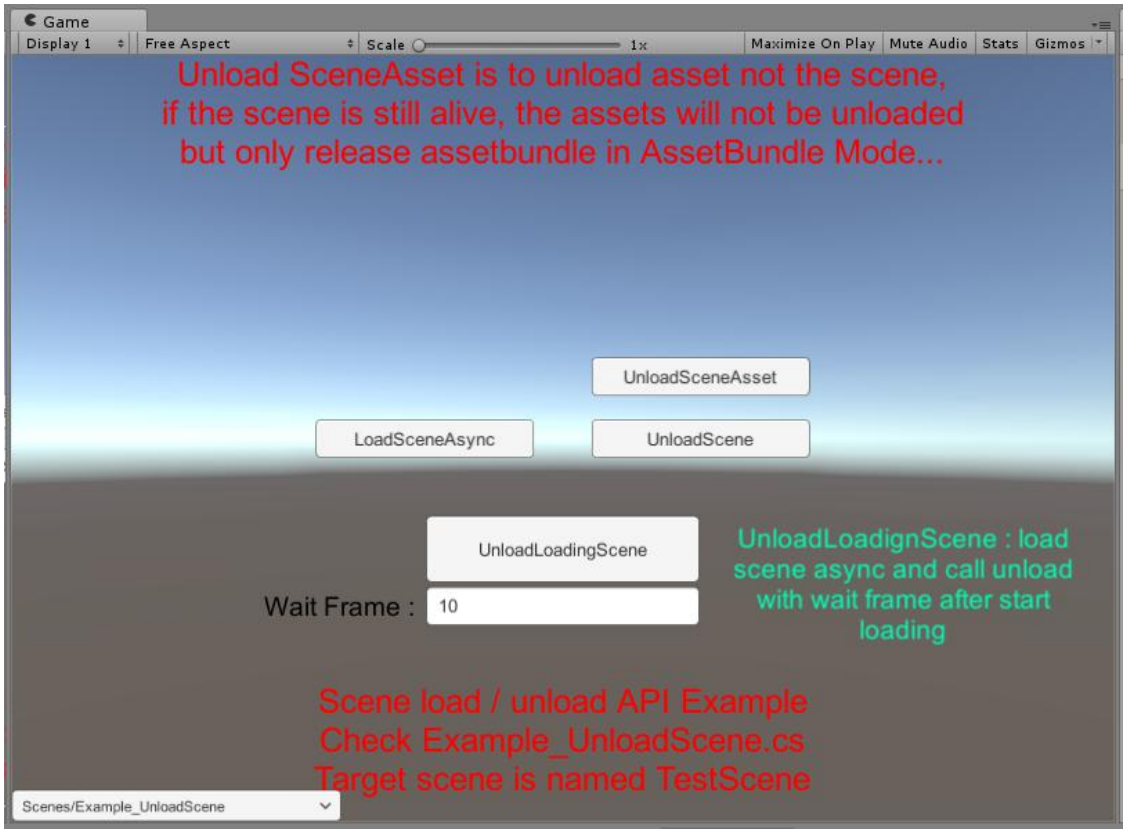
PS: both of them were set to a small value for editor proflier testing in StartScene.cs script.

3. In StartScene.cs::Start() function you can also see the other variable

--> AssetUnloadManager.Instance.maxUnloadPerFrame = 1f; It holding the unload assetbundles and ticks every frame, the number is how many assetbundles can be unload per frame, ranged [0.01f, 1000f] means it can be less then 1, if set it less then 1, It means 1/maxUnloadPerFrame frames to unload an assetbundle, for example 0.5 is every 2 frames to unload 1 assetbundle, and 0.1 is every 10 frames to unload 1 assetbundle, this is because different platform has different performance on it. for example in Editor it takes 200ms to unload 87 assetbundles at a time, but in Built windows exe, it takes 13ms to unload 87 assetbundles, however we can separate the unload to different frames, not spike the runtime.

7. Example_UnloadScene (Script : Example_UnloadScene.cs)

This demo shows unload scene features, scene loading can have both async and sync process, the scene loaded event is triggered by SceneManager.sceneLoaded, so to unload scene is a little complex(but simple for user side) :

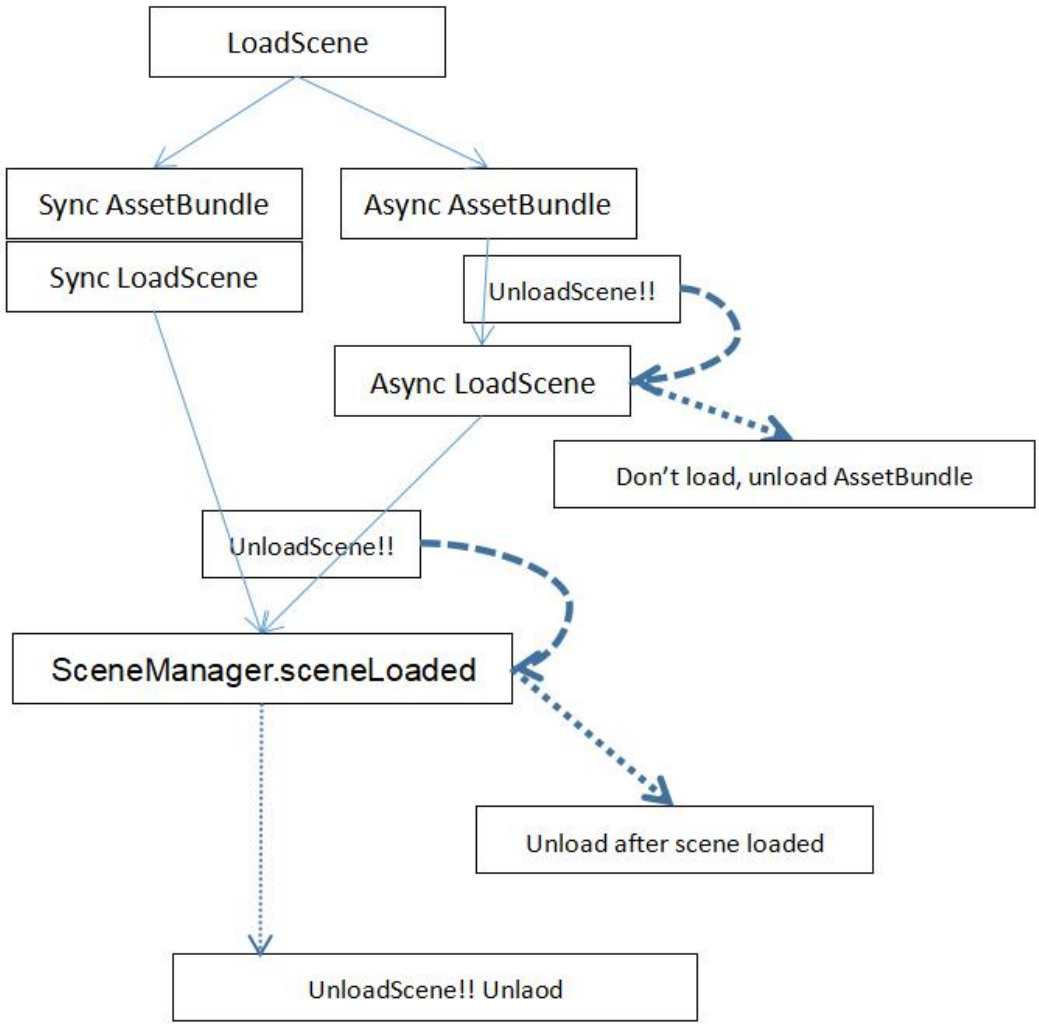


```
void UnloadOneScene()
{
    if(m_loadedScenes.Count > 0)
    {
        var sceneId = m_loadedScenes[0];
        m_loadedScenes.RemoveAt(0);
        Debug.Log("Request unload at frame : " + Time.frameCount);
        // Please Unload Scene In This Way
        SceneLoadManager.Instance.UnloadScene(sceneId, (_id, _scene) =>
        {
            Debug.Log("Unloaded " + _id + " [" + _scene.path + "] at frame : " + Time.frameCount);
        });
    }
}
```

API (namespace AssetBundleMaster.ResourceLoad):

SceneLoadManager.Instance.UnloadScene(int id, System.Action<Scene> unloaded = null);

The scene load API (SceneLoadManager.Instance.LoadScene) retuen a HashCode, UnloadScene is use this HashCode. and the unload asset process will triggered if any scene asset can be unload. it is also auto resource control. The scene load has async process, the unload flow chart is like this(if not AssetBundle mode, no AssetBundle step) :



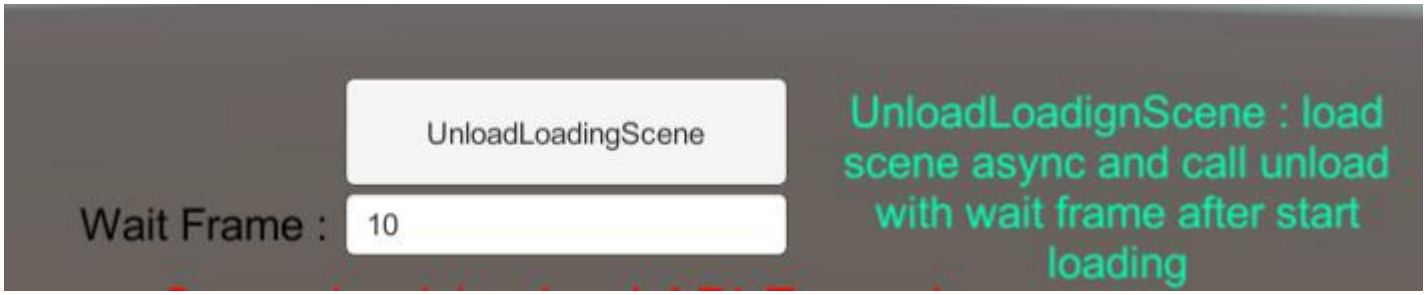
The Async loading may be :

1. Async load AssetBundle

2. Async load Scene

3. Wait SceneManager.sceneLoaded called no matter load scene Async/Sync

So the UnloadScene function wrapped all situations to make sure unload request work at any loading step.



This is to test UnloadScene function, it do load scene Aync and unload after Wait Frames, change to 1,2,3...etc, can see in all step it works.

Thats all the Asset load, GameObject load, Scene load and unload logic, controlled by only 3 managers:

1. ResourceLoadManager

2. PrefabLoadManager

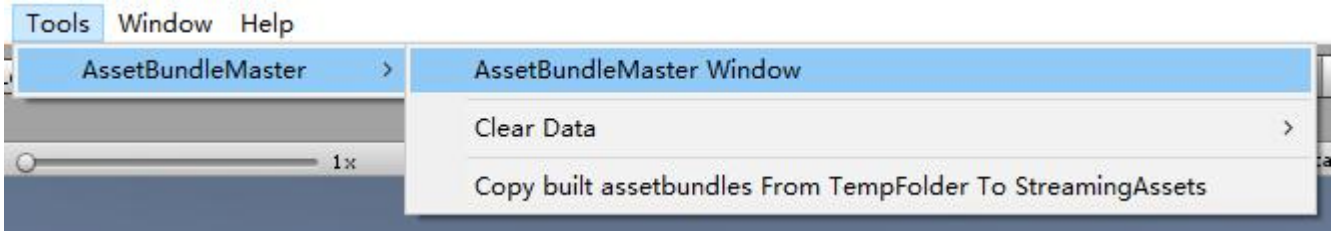
3. SceneLoadManager

if you run in editor mode, some tests may not work as well such as Async load and asset duplicate. Let's start build AssetBundle, and runs it in Editor and published app.

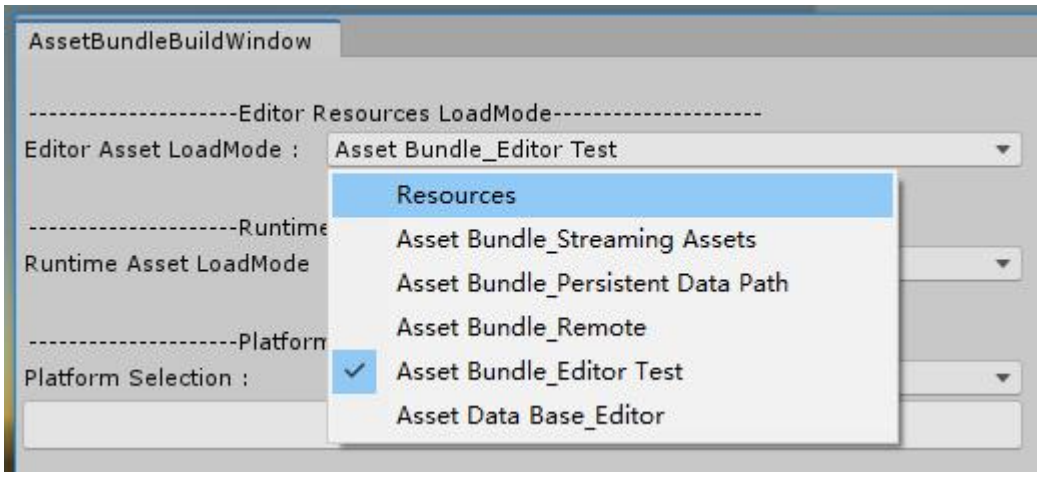
PS : StartScene1 Dropdown has a scene Example_UnloadAssetEfficiency is to explain how auto resources unload logic work with diferent Load Managers, see Appendix.

---------- Build AssetBundles ----------

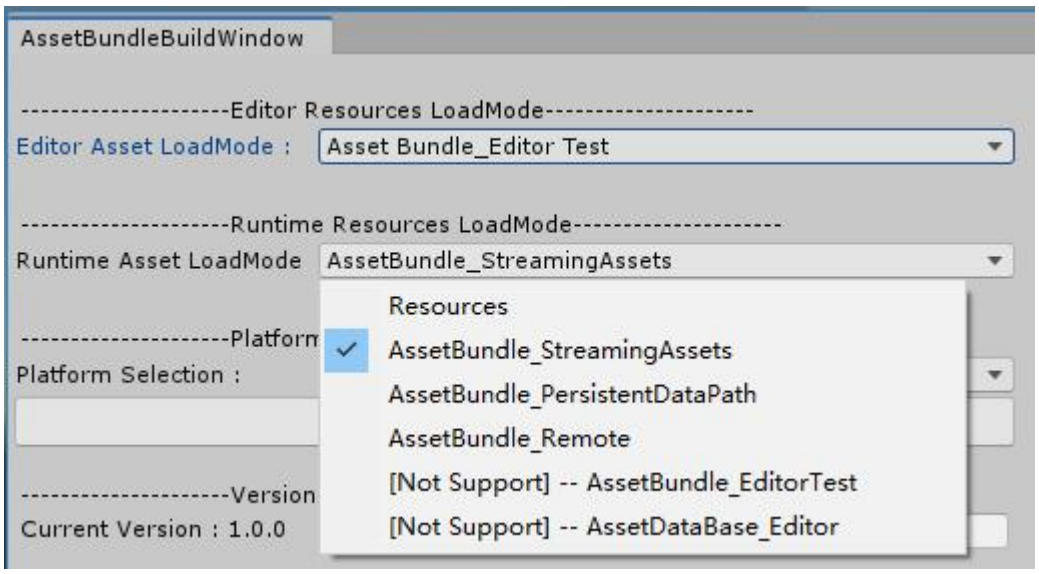
Open the AssetBundleMaster Window



We have 6 load modes, and have 4 published asset load modes.



Editor Modes



Publish modes

1. Resoueces mode: used as default Resources load mode, if you use it, you don't have to use AssetBundleMaster.

2. AssetBundle_StreamingAssets mode: load AssetBundle from StreamingAssets.

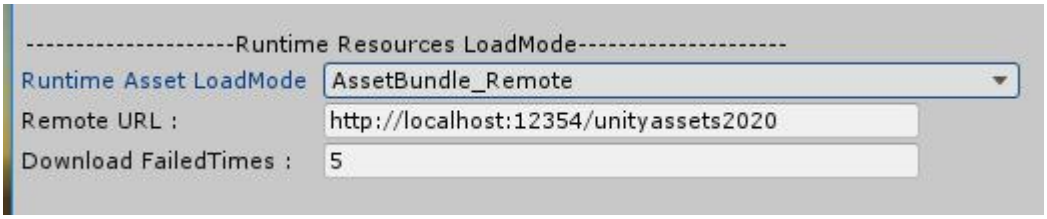
3. AssetBundle_PersistentDataPath mode : updatable resource mode, load from PersistentDataPath, if no asset in PersistentDataPath, load from StreamingAssets. (we don't implement any update logic, but generate patch file for you)

4. AssetBundle_Remote : remote assetbundle mode, you can put your assetbunldes to any server, and load from it.

5. AssetBundle_EditorTest mode: load from temp AssetBundle folder, build AssetBundle will build to temp folde (Editor Only).

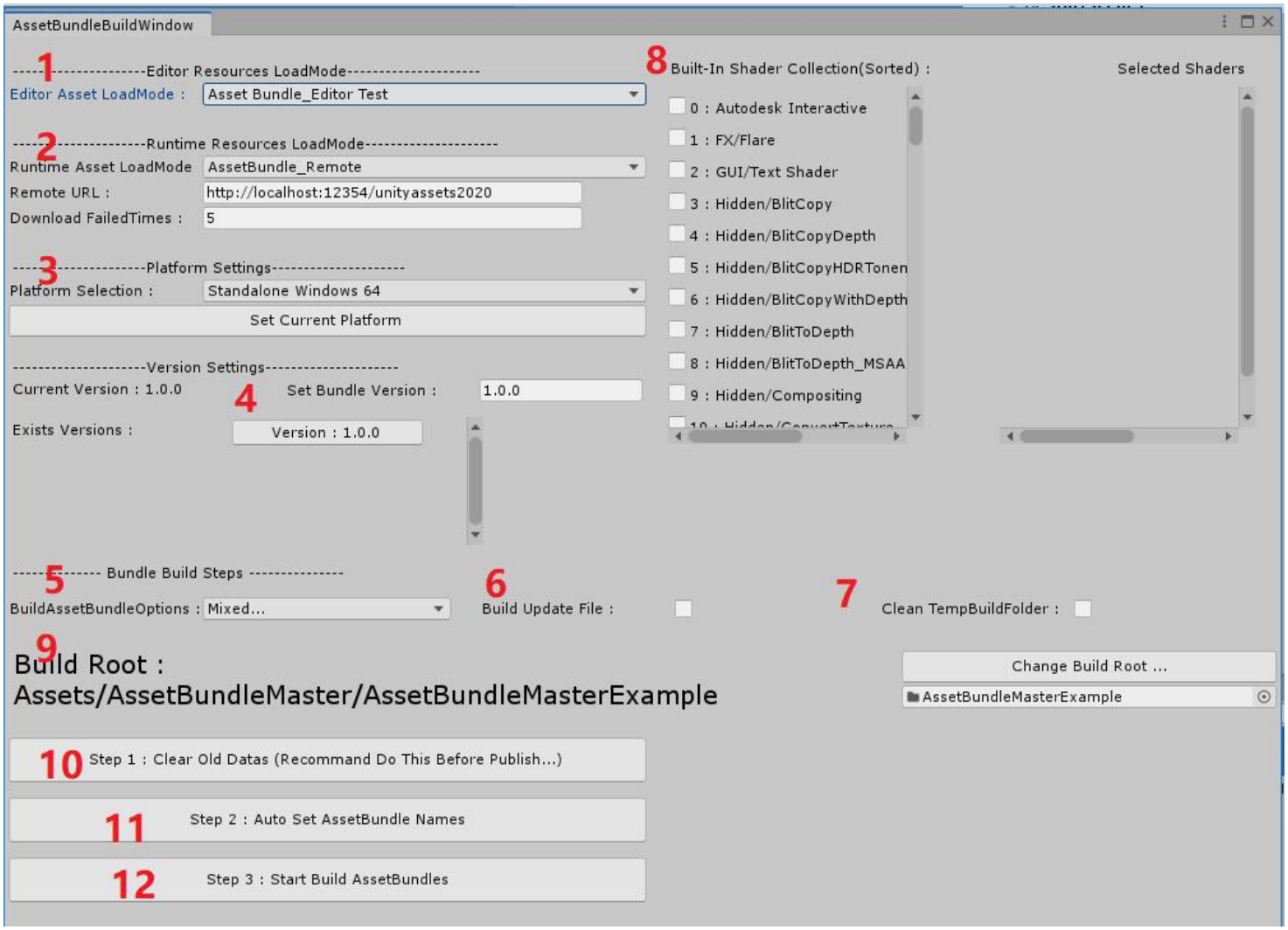
6. AssetDataBase_Editor mode: no need to build AssetBundle, load asset directly (Editor Only).

If you use the AssetBundle_Remote mode, the following options will shows up :



Remote URL is the server url, is the same as your root folder. Download FailedTimes is the retry times if assetbundle download failed, the download use the lcoal cache, the APIs are UnityWebRequestAssetBundle / UnityWebRequest / LoadFromCacheOrDownload based on the different Unity Version. if you are testing with a http server, it may have some CORS or MIME issues, check it.

Let's check all the settings on AssetBundleMaster Window :



1. Editor Asset LoadMode (EnumPop) : Asset load mode in Editor.

2. Runtime Asset LoadMode (EnumPop) : Asset load mode in Published.

3. Platform Selection (EnumPop) : AssetBundle target platform, click [Set Current Platform] can set to current platform.

4. Set Bundle Version (Text) : The version of AssetBundle.

5. BuildAssetBundleOptions (EnumFlagPop) : BuildAssetBundleOptions.

6. Build Update File (CheckBox) : Create Patch between versions, Patch file is Josn, serialized from AssetBundleMaster.AssetLoad.LocalVersion.UpdateInfo.

7. Clean TempBuildFolder (CheckBox) : Clear any file in temp folder not in this build.

8. Built-In Shader Collection (Scroll View CheckBox) : Deal with Built-In Shader, no multi compiled.

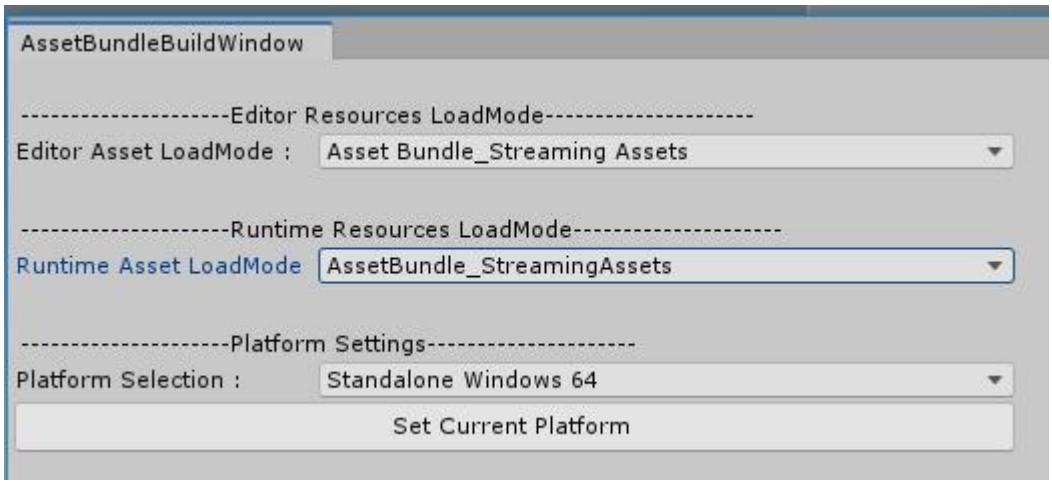
9. Build Root (Label, Button) : Root folder, asset in this folder can be load.

10. Step1 Clear Old Datas (Button) : Clear all AssetImporter.AssetBundleName.

11. Step 2 Set AssetBundle Names (Button) : Auto set AssetBundleNames, some asset auto process. see appendix.

12. Step 3 Start Build AssetBundles (Button) : Start Build AssetBundles.

That's all settings. Let's set editor load mode and runtime load mode to AssetBundle_StreamingAssets, this is the most simple publish mode. and click :

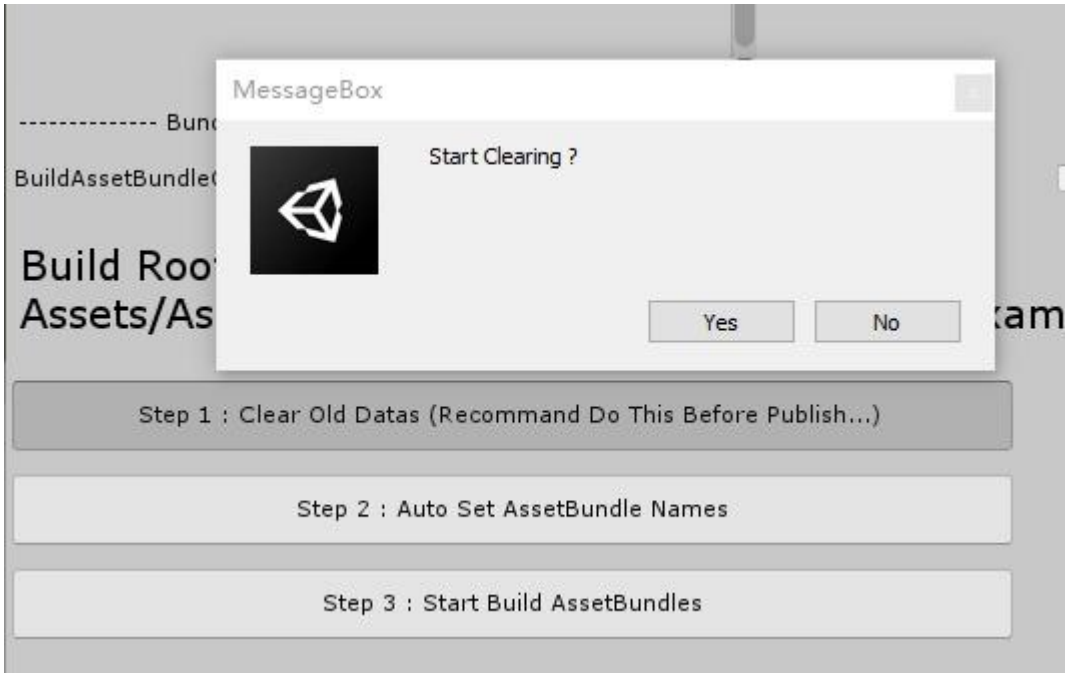


Step1...

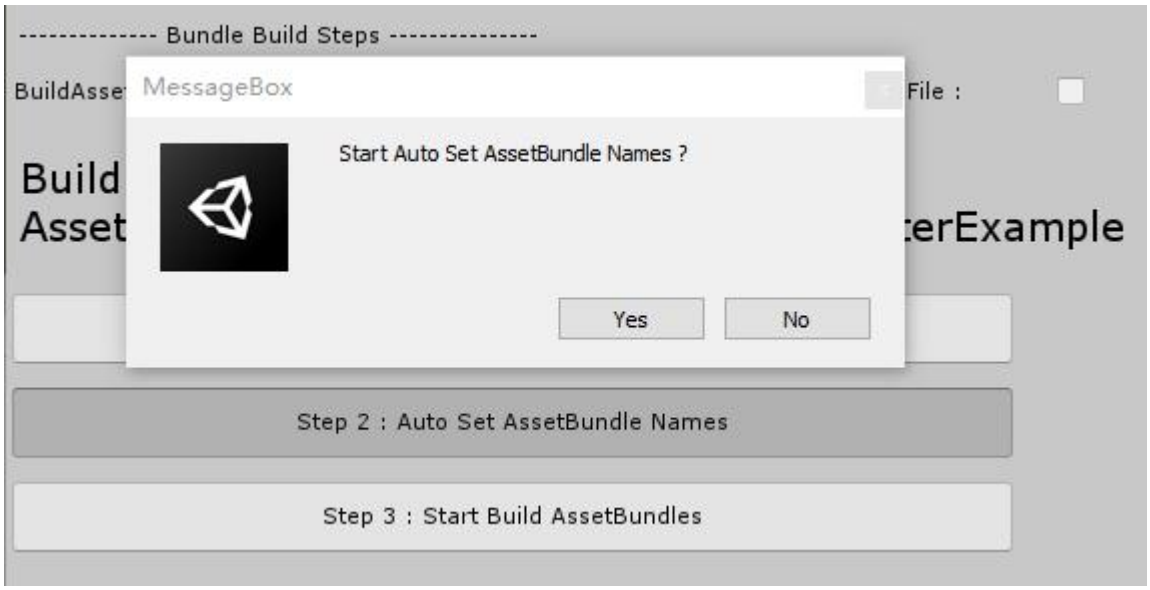
Step2...

Step3...

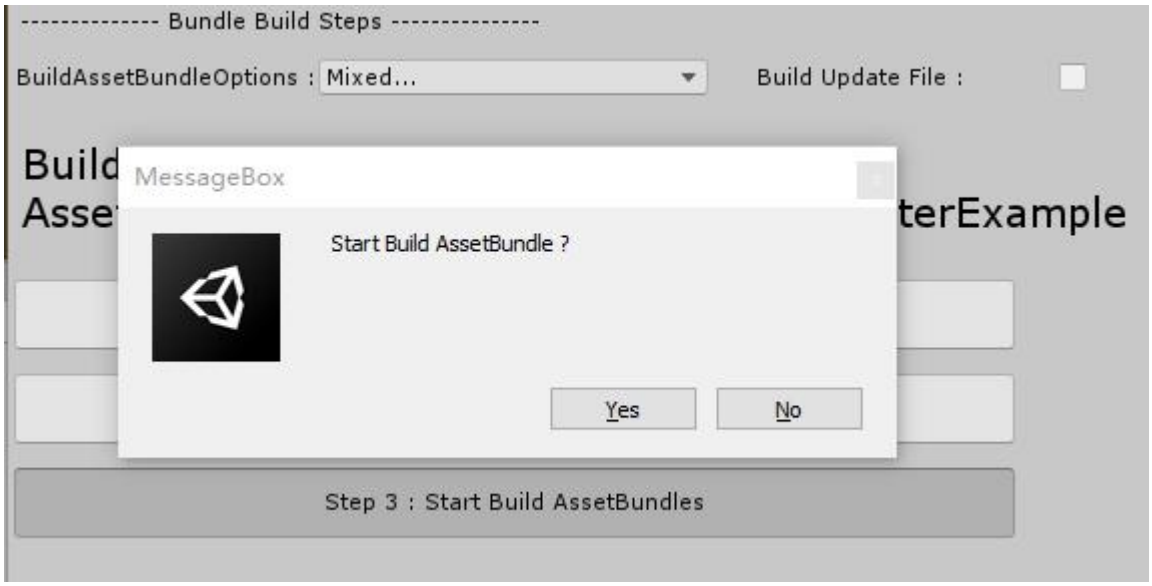
It will auto build AssetBundle for you. every step shows tips :



Step1

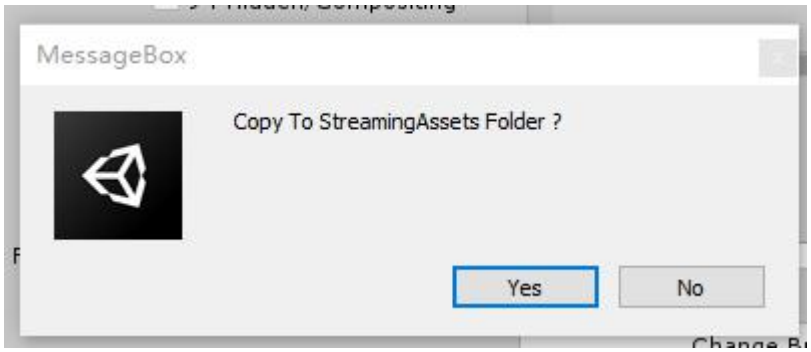


Step2

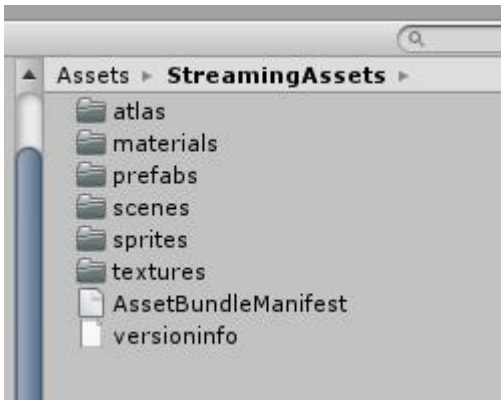


Step3

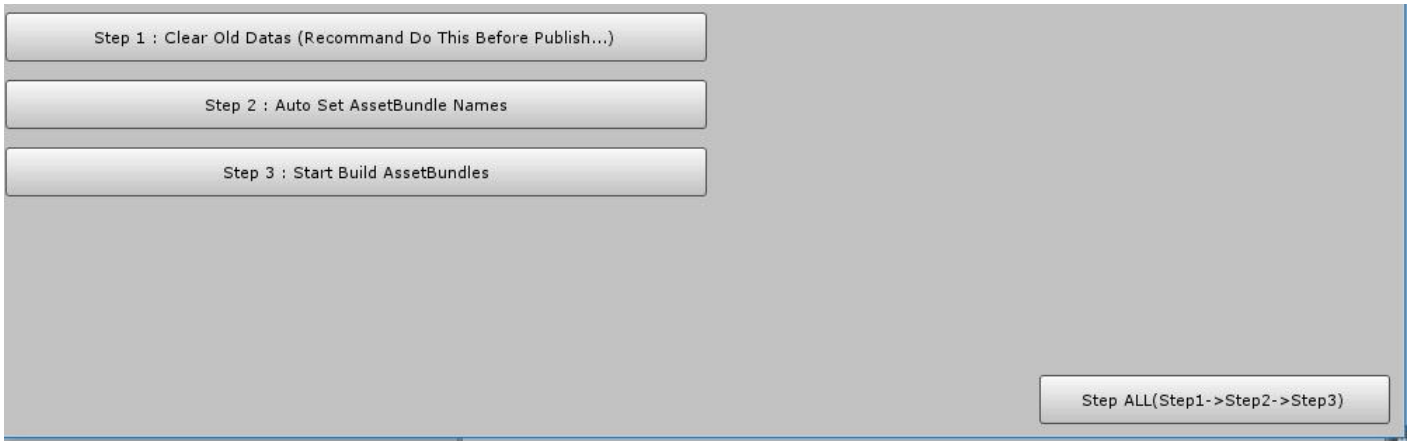
When build finished, It will ask you to copy built AssetBundles from temp folder to StreamingAssets? Yes.



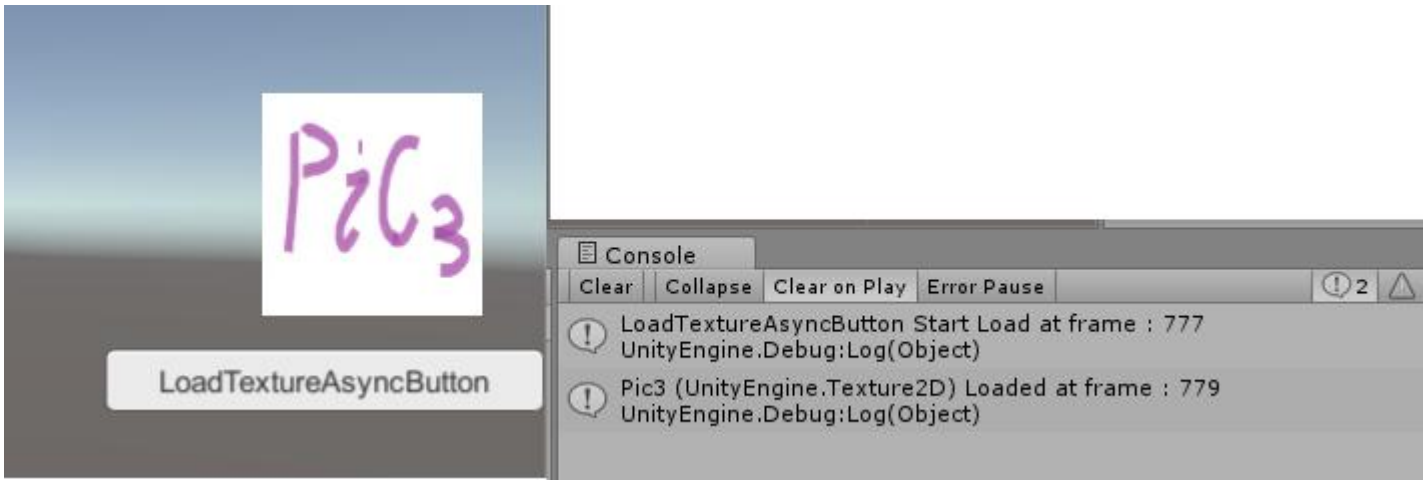
You can see StreamingAssets got built AssetBundles :



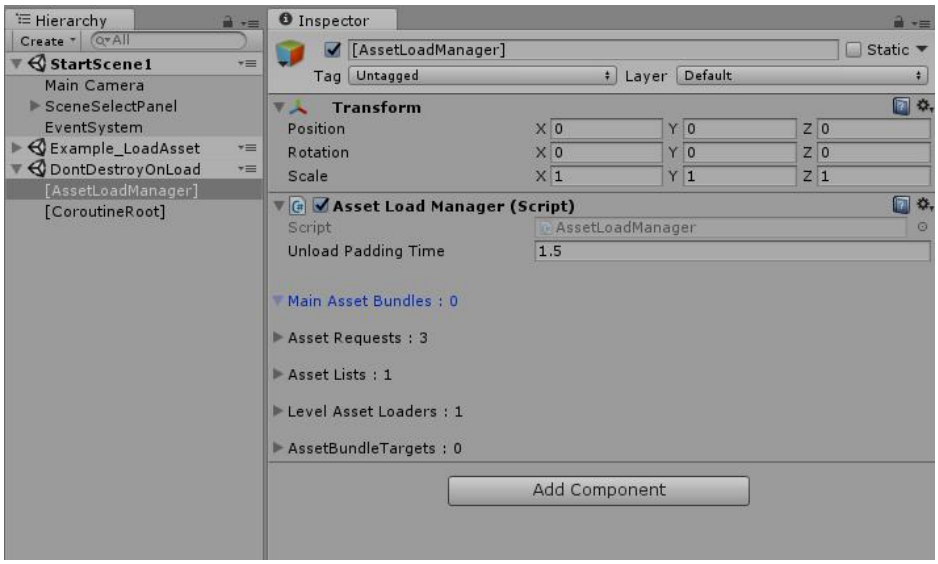
We added a wrap build function button at bottom-right, it wrap step1, step2, step3 steps:



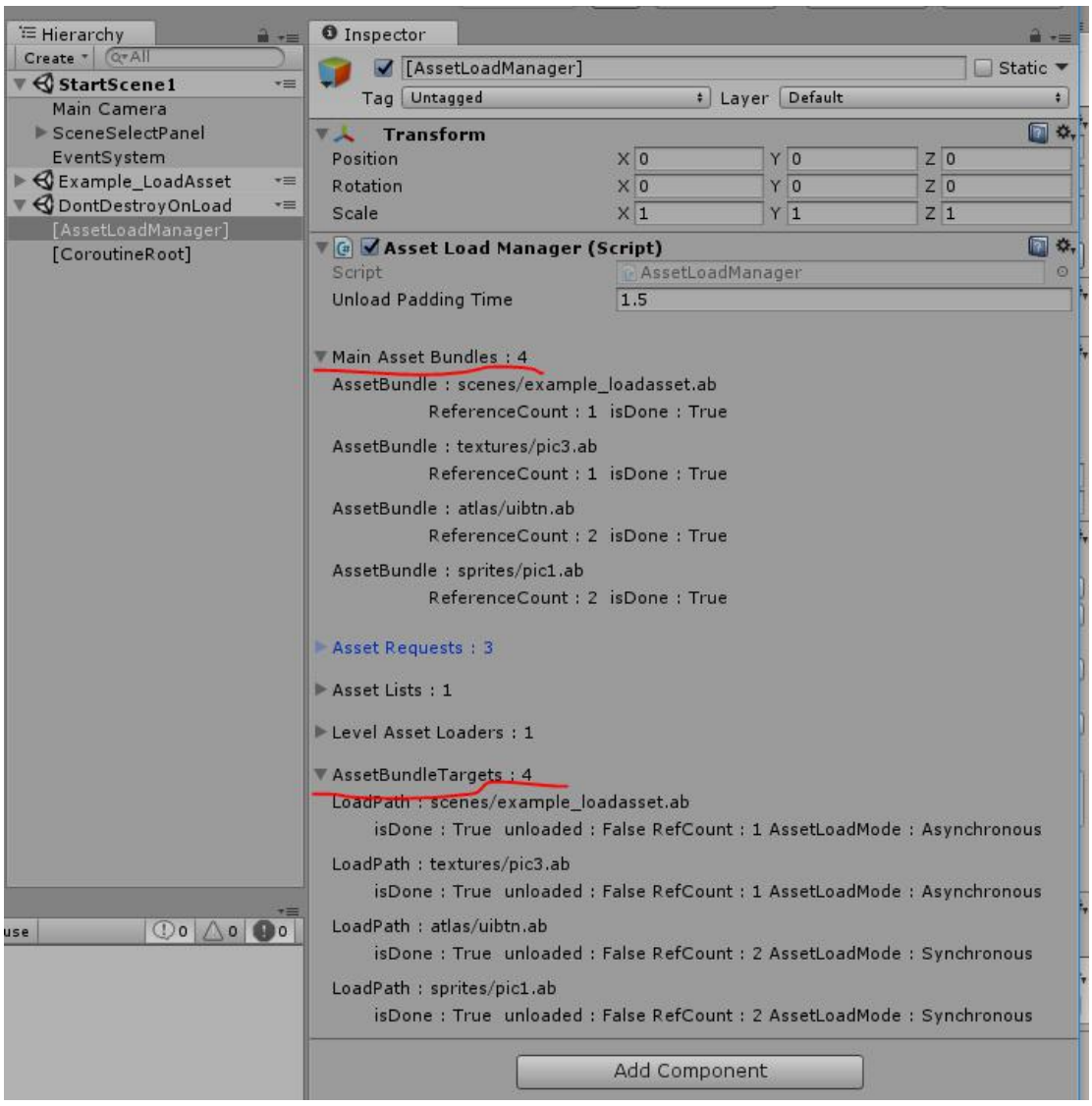
So the AssetBundle build successed, if you run in Editor, just open the StartScene1, repeat the API tutorial again. This time the Async load is working.



And the [AssetLoadManager] shows different if you noticed :



AssetDataBase_Editor no AssetBundle was loaded



AssetBundle_StreamingAssets load AssetBundles

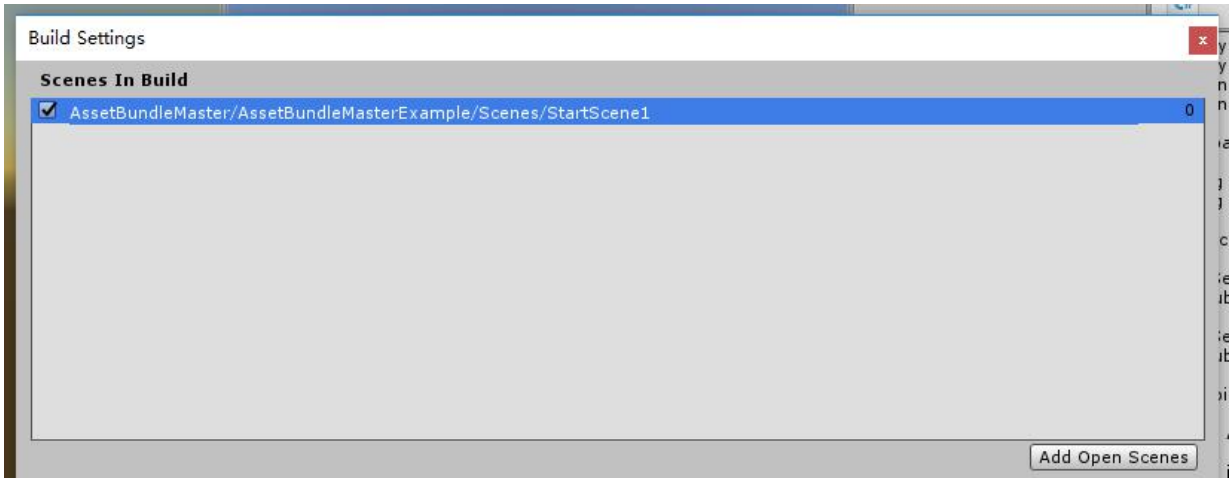
We did successed building AssetBundles and load AssetBundle in Editor. You must notice in Editor Play, it may referencing assets unexpect, such like SpriteAtlas, it may cause the resources unload error, but it only happened in Editor, you don't have to worry about the publish build. The best development mode is use AssetDataBase_Editor in editor mode,

and running on target platform after built AssetBundles. We only found out the error appeared when Editor SpriteAtlas runtime caching data works, if you want to test AssetBundle mode in Editor, you can delete .spriteatlas files before Play(SpriteAtlas was auto generated by AssetBundleMaster in Assets/AssetBundleMasterSpriteAtlas folder).

If you want to publish it, make sure your Platform Selection correct, the next one is very important, scene and the Build Settings :

When you pubilsh your project, there is no need to add any scene to build Settings, because the scenes in Build Root folder will be built to AssetBundle, and any scene added to Build Settings will be packed to Resources too. So how to publish the app and load the StartScene1 as default scene? there are 2 ways :

1. Add entry scene to scene build as default scene



This is the most commonly used, but notice if you add StartScene1 to build settings, and the scene file [StartScene1.unity] is under the Build-Root directory, it will be duplicated after build. if you move the file out of Build-Root directory, you can't use SceneLoadManger API to load StartScene1, you must load it by UnityEngine.SceneManagement API as usual.

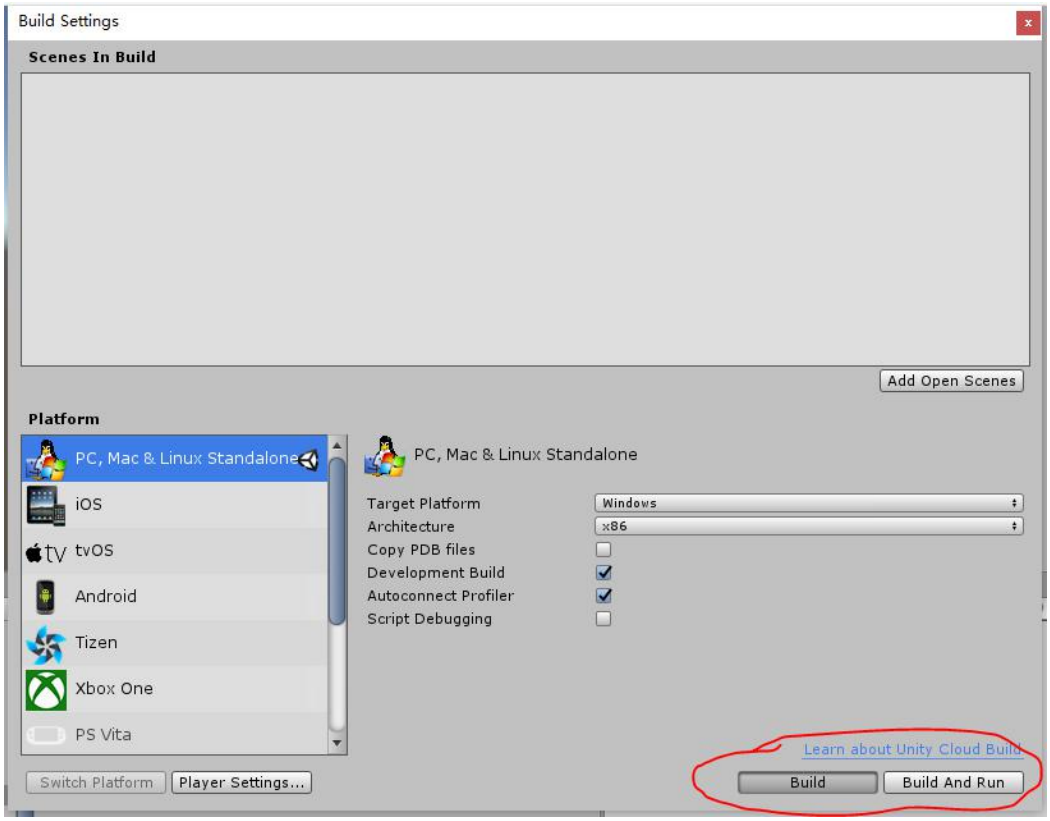
2. Or you can use the [UnityEngine.RuntimeInitializeOnLoadMethod] attribute to load the entry scene.

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    SceneLoadManager.Instance.LoadScene("Scenes/StartScene1");
}
```

Or you are useing AssetBundle_Remote Mode :

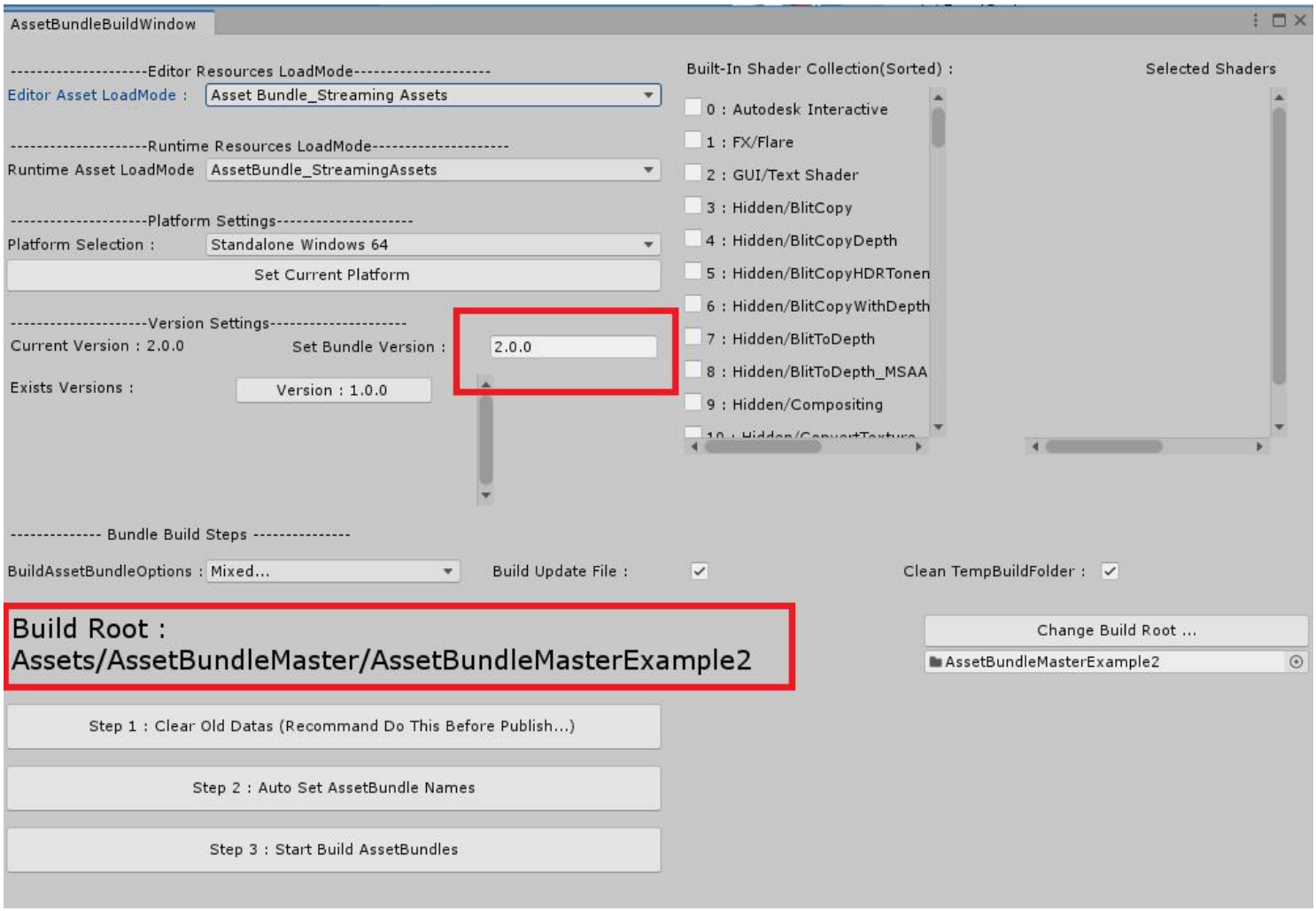
```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    // If AssetBundle_Remote mode, muse wait for Inited
    AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
    {
        SceneLoadManager.Instance.LoadScene("Scenes/StartScene1");
    });
}
```

You should add this code to any script yourself. This is pretty good to avoid scene build duplicated, but need coding. In this way your Build Settings is stay empty.



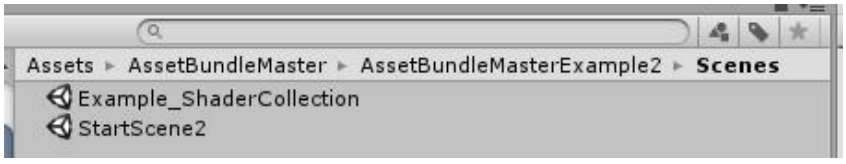
No need to add any scene, just build it and run.

Now let's look at the function 8. Built-In Shader Collection. Let's change the Build Root to AssetBundleMasterExample2 :

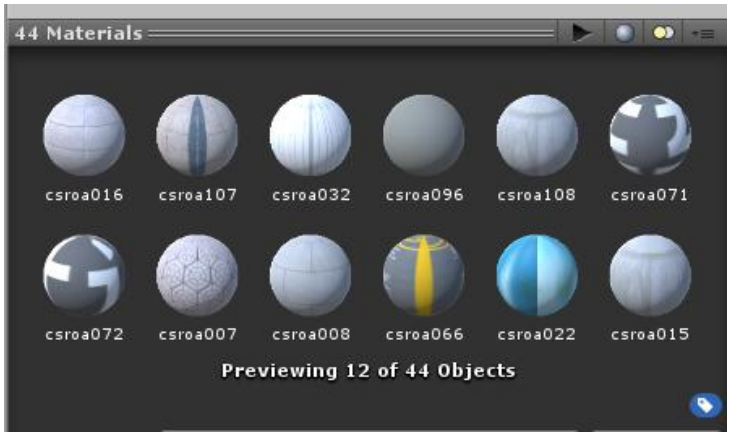


Click [Change Build Root] button to change it.

And chagne the version to 2.0.0, We Built 1.0.0 before, so Exists Versions shows 1.0.0 Exists, check the Build Update File, we can see Patch file will be generated later. Build Root is AssetBundleMasterExample2 now, let's see what is in this folder :

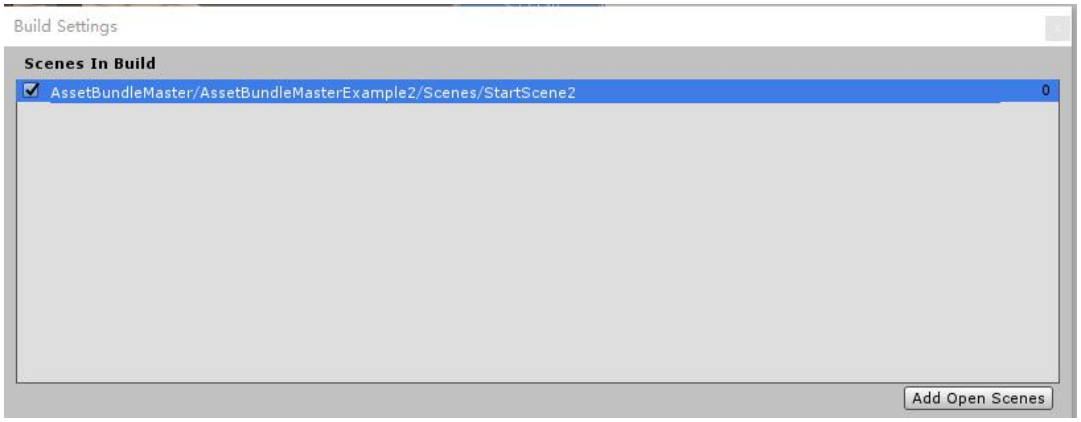


There are 44 materials were used in the Example_ShaderCollection scene, they all used the Standard Shader :



These materials may cause the issue. We need to change the entry scene to StartScene2, it will load Example_ShaderCollection, now change the entry scene :

1. Change the entry scene by Build Settings



2. Or change code

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    SceneLoadManager.Instance.LoadScene("Scenes/StartScene2");
}
```

Or you are useing AssetBundle_Remote Mode :

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    // If AssetBundle_Remote mode, muse wait for Inited
    AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
    {
        SceneLoadManager.Instance.LoadScene("Scenes/StartScene2");
    });
}
```

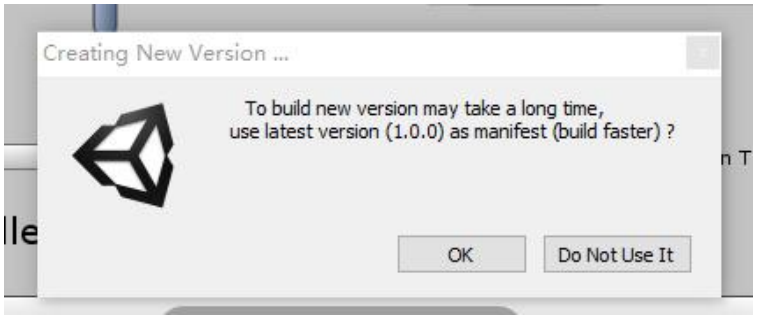
Do build AssetBundle Steps again,

Step1...

Step2...

Step3...

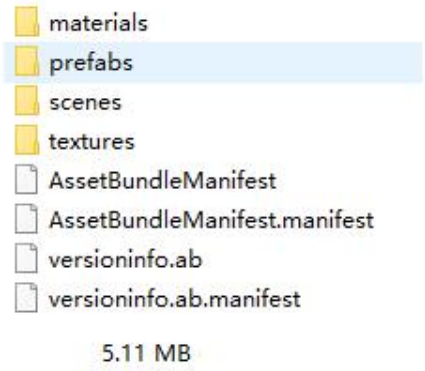
This time the Step3 shows new tips :



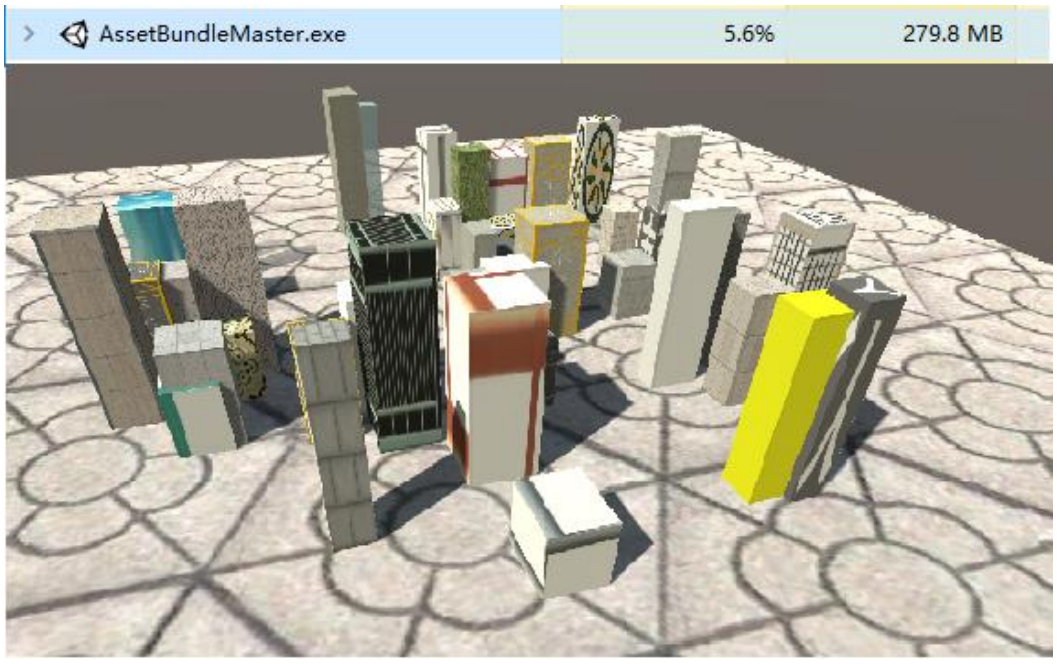
This is because we built 1.0.0 version bundles already, and if we create 2.0.0 version it can be a incremental build, it will be faster than build a totally new version. But this build is to build AssetBundleMasterExample2, is totally diferent with AssetBundleMasterExample, no shared assets, so whatever you choose, it will build a totally new version.

So the function 7. Clean TempBuildFolder (CheckBox) is used for it, if checked, the temp folder will do clean files that not in this version.

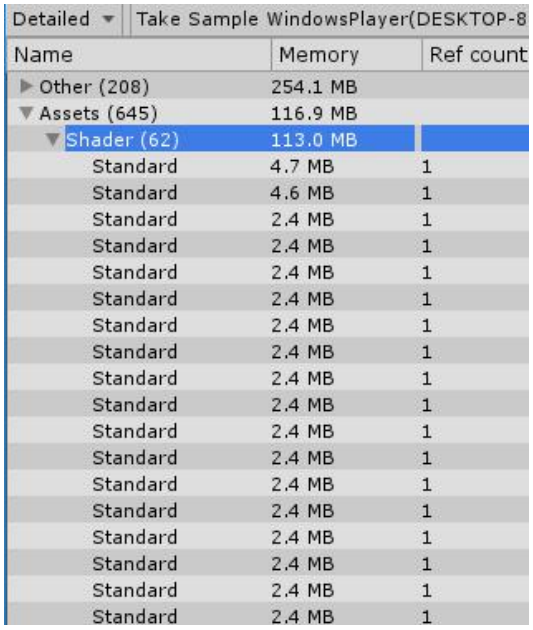
After the build AssetBunlde finished, we build app to check what happened at the runtime. don't forget to select the [Development Build] and [Autoconnect Profiler] in Build Settings. My platform is windows 10. Screen shot from Unity2020.



Total AssetBundle size : 5.11MB(With .manifest files)

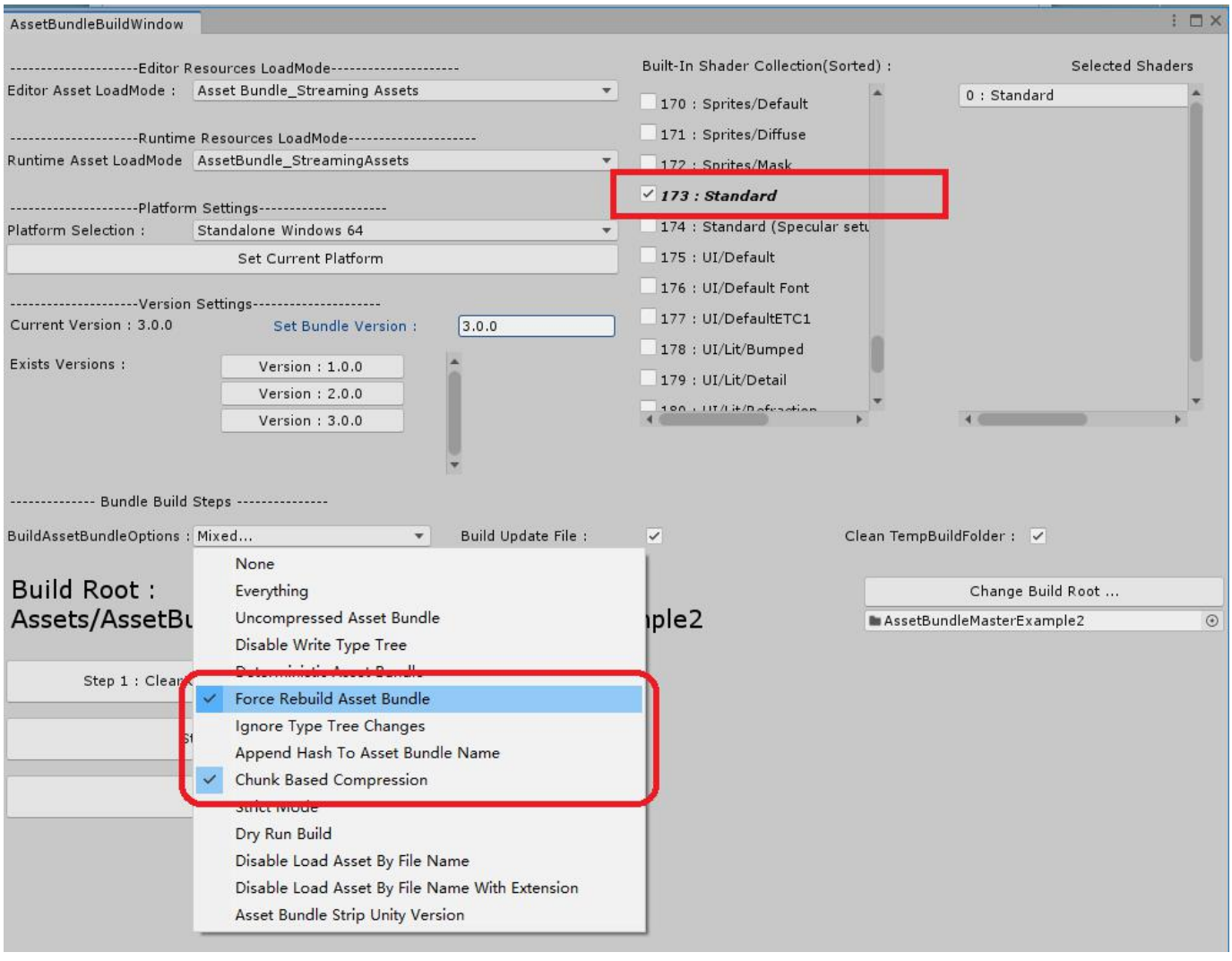


The Task Manager shows the app runtime memory : 279.8MB

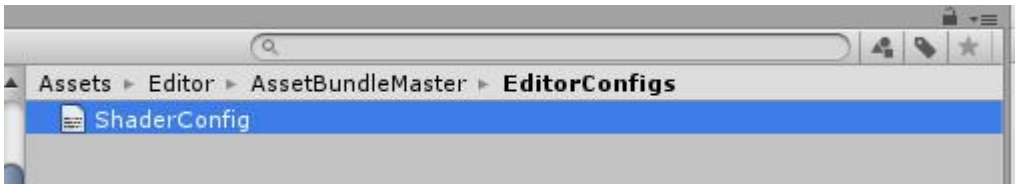


Profiler Shader 113MB, Standard was compiled many times

This issue happend on Standard Shader, let's use the Shader Collection to collect Standard Shader, and build a 3.0.0 version, the shader changes will not cause the changes of referencing assets, so we must select the ForceRebuildAssetBundle on BuildAssetBundleOption, otherwise we will lost the reference of materials :



PS : the default setting of BuildAssetBundleOption is ChunkBasedCompression, it is just for building AssetBundle faster, you can change the settings by your need, and the settings serialized files is in the editor folder shows below, you can upload to SVN share to your teammate :



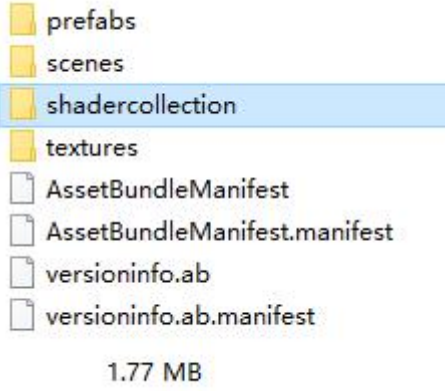
Do build AssetBundle Steps again,

Step1...

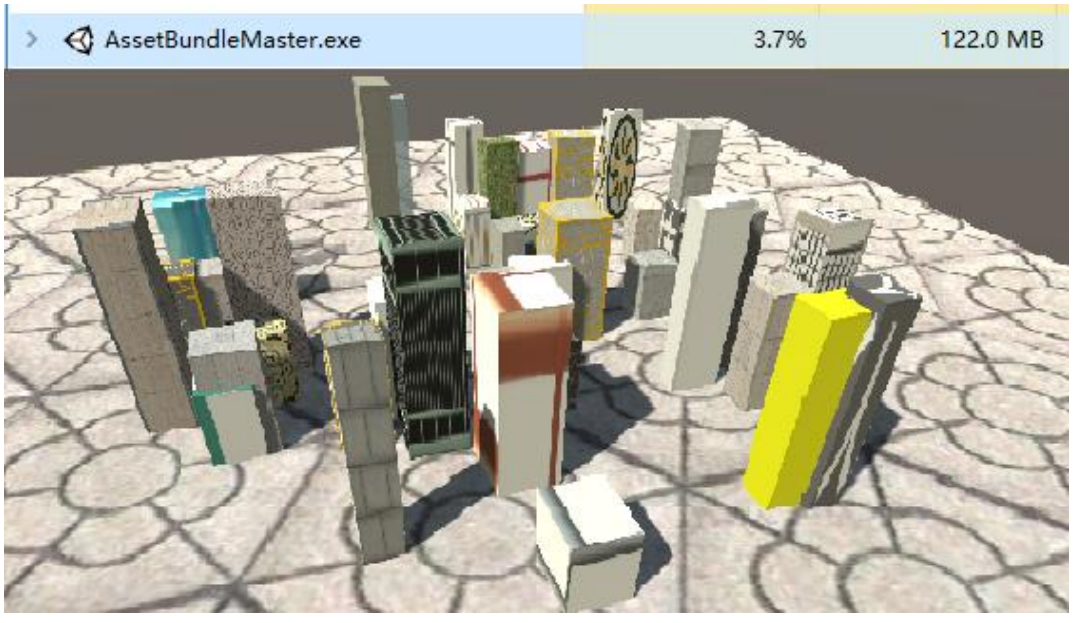
Step2...

Step3...

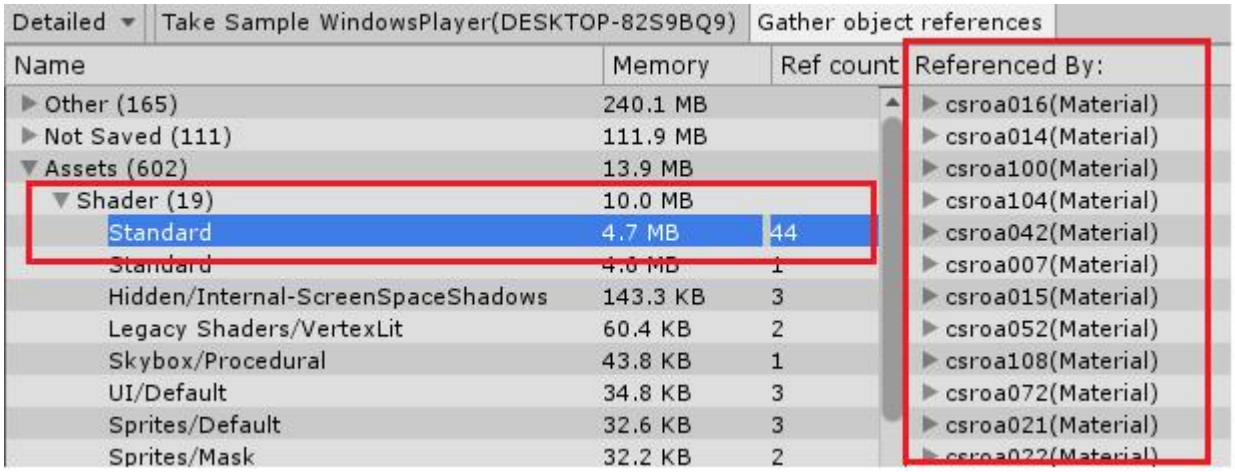
And build app to run :



Total AssetBundle size : 1.77MB VS 5.11MB



Task Manager shows runtime memory 122.0MB VS 279.8MB



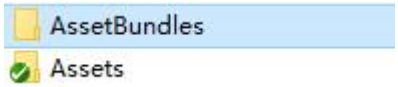
Profiler : Shader 10M VS 113MB, Standard compiled once only.

You can see the big different between 2.0.0 and 3.0.0, in 2.0.0 we are wasting I/O time and runtime memory. this issue mainly caused by Built-In Shader(based on your usage, for exp the UGUI using built-in shaders but have the same built-in material, it will not cause this problem).

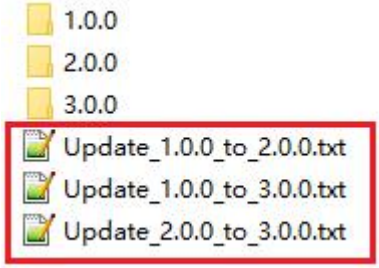
All tests were tested on Unity5/Unity2017/Unity2018/Unity2019/Unity2020, nearly got the same result.

PS : Shader collection can be used for small project to solve these issues, but if a middle or large-class project using the Standard material largely, it will not be a good solution, even more we strictly recommand don't select any shader collection. Please follow the Unity's suggestion.

Let's take a look at AssetBundle temp folder, it is at the same layer as the Assets folder :



Assets/../AssetBundles/StandaloneWindows64/



You can see Version based AssetBundle Temp Folders and Update Patch files, patch file is Json format looks like this (Update_2.0.0_to_3.0.0.txt) :

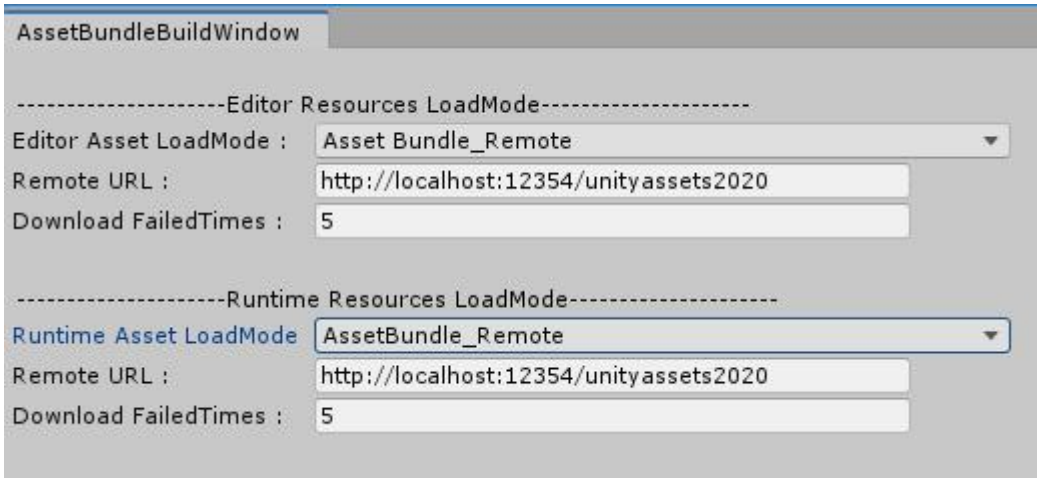

```
{
    "buildTime":"2021-04-19 10:18:32",
    "updateList":[
        "prefabs/city.ab",
        "shadercollection/standard.ab",
        "versioninfo.ab"
    ],
    "deleteList":[
        "materials/csroa000.ab",
        "materials/csroa001.ab",
        "materials/csroa002.ab",
        "materials/csroa003.ab",
        "materials/csroa004.ab",
        "materials/csroa007.ab",
        "materials/csroa008.ab",
        "materials/csroa012.ab",
        "materials/csroa014.ab",
        "materials/csroa015.ab",
        "materials/csroa016.ab",
        "materials/csroa019.ab",
        "materials/csroa020.ab",
        "materials/csroa021.ab",
        "materials/csroa022.ab",
        "materials/csroa023.ab",
        "materials/csroa026.ab",
        "materials/csroa032.ab",
        "materials/csroa034.ab",
        "materials/csroa036.ab",
        "materials/csroa041.ab",
        "materials/csroa042.ab",
        "materials/csroa052.ab",
        "materials/csroa060.ab",
        "materials/csroa066.ab",
        "materials/csroa070.ab",
        "materials/csroa071.ab",
        "materials/csroa072.ab",
        "materials/csroa073.ab",
        "materials/csroa096.ab",
        "materials/csroa098.ab",
        "materials/csroa099.ab",
        "materials/csroa100.ab",
        "materials/csroa101.ab",
        "materials/csroa102.ab",
        "materials/csroa103.ab",
        "materials/csroa104.ab",
        "materials/csroa105.ab",
        "materials/csroa106.ab",
        "materials/csroa107.ab",
        "materials/csroa108.ab",
        "materials/csroa109.ab",
        "materials/csroa110.ab",
        "materials/csroa111.ab"
    ]
}
```


Means if you are Version 2.0.0, update to 3.0.0 you sould download files listed in "updateList" from server or CDN, and delete files listed in "deleteList" in local. The patch file was generated between versions, serialized from AssetBundleMaster.AssetLoad.LocalVersion.UpdateInfo, json format.

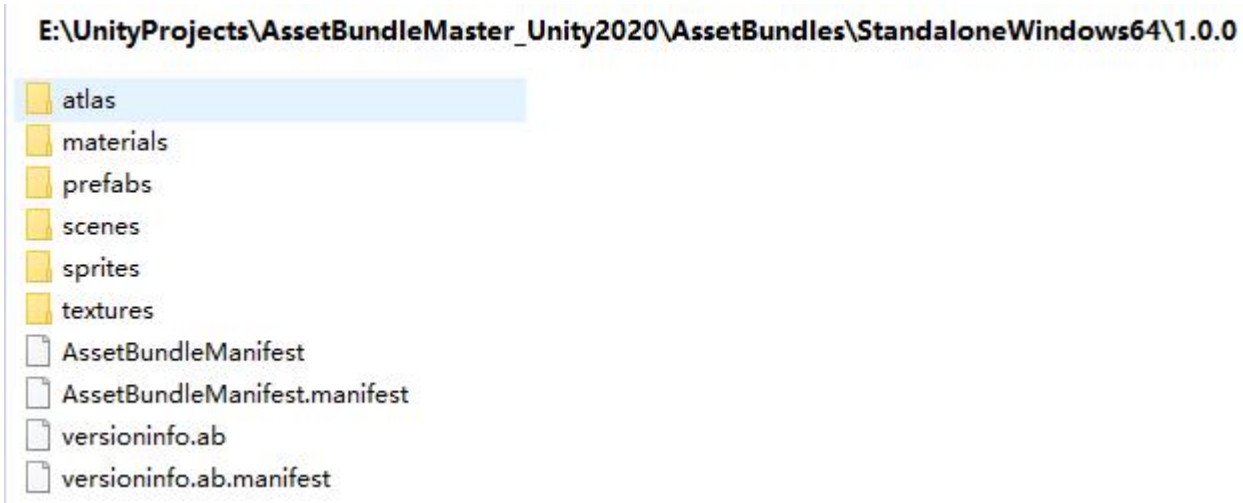
```
public class UpdateInfo : BaseJsonSerialization
{
    [SerializeField]
    public string buildTime;

    public System.Version from;
    public System.Version to;

    [SerializeField]
    public List<string> updateList = new List<string>();
    [SerializeField]
    public List<string> deleteList = new List<string>();

}
```

Notice : we are only generate update-patch files, there is no file update system implemented in AssetBundleMaster, users must implement the file update system your self.(Based on different game requirments, the update logic is very complicated to satisfied)

Notice2 : If you just need a remote assetbundle loader, you just set AssetLoadMode to AssetBundle_Remote, it can load the newest assetbundle from cache or server.



Use AssetBundle_Remote load from a server example. what we need to do is copy the built assetbundles to the server directory, for example my built path is :



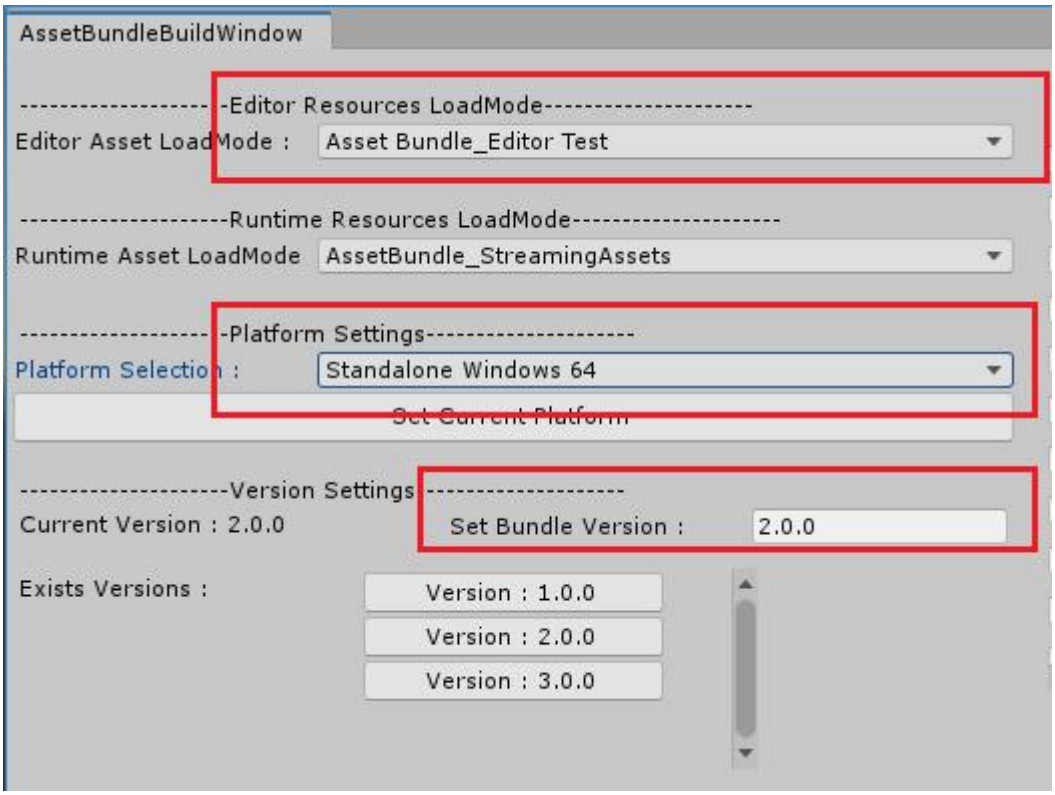
Just Copy To Server Directory (my IIS server path root is E:\LocalServer\WebGL_Raw, port:12354), my Web.config is set non-extention and .ab files to send as bytes :

```
<mimeMap fileExtension="." mimeType="application/octet-stream" />
<mimeMap fileExtension=".assetbundle" mimeType="application/octet-stream" />
<mimeMap fileExtension=".ab" mimeType="application/octet-stream" />
```

so AssetBundles can be downloaded.

.manifest file is no need to copy to server.

If you are using AssetBundle_EditorTest mode, it load AssetBundles from temp folder, you can switch load between built versions easily, just change the [Platform Selection] and [Set Bundle Version] :



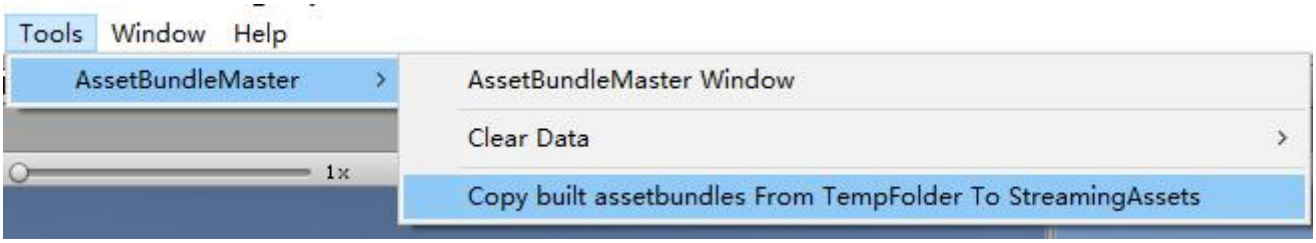
Click The Exists Versions buttons will also change to target version.

Temp Folder is Assets/../AssetBundles/StandaloneWindows64/x.x.x/

PS : when you start run, the asset load root path will log to console, you can check it.

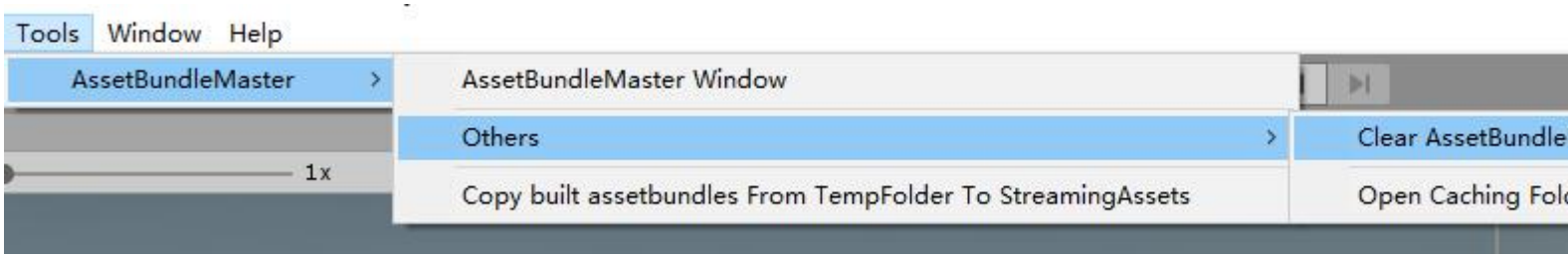
Other functions in the Tools Bar:

1. Copy built AssetBundle from temp folder to StreamingAssetsfolder :



This is the same function as to the AssetBundle build finished tips.

2. Clear AssetBundleMaster serialized datas :



It will clear the generated settings files said before, AssetBundleMaster can not run without these files.

3. Open Caching Folder

Open the cache folder, if you are using AssetBundle_Remote load from a server, you can see cached files in this folder, Unity5 shows nothing even cached(Maybe cached files not in this folder), Unity2017~Unity2020 showing correctly.

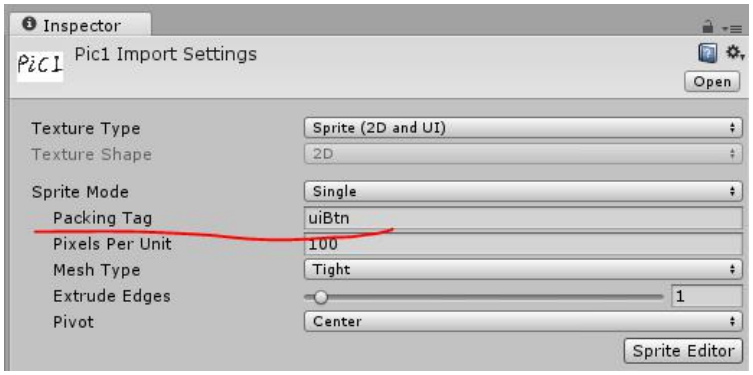
These are all about AssetBundleMaster, it was designed for small or middle-class projects and provided the convenience and correctness for resource control, give you a flexible developing flow chat. Have Fun.

Nowadays the AssetBundleMaster Version 3.3.0 with performance optimized, can support large-class projects as well, based on the non-duplicated asset control core, we think it will be a popular framework design. hope it can help you.

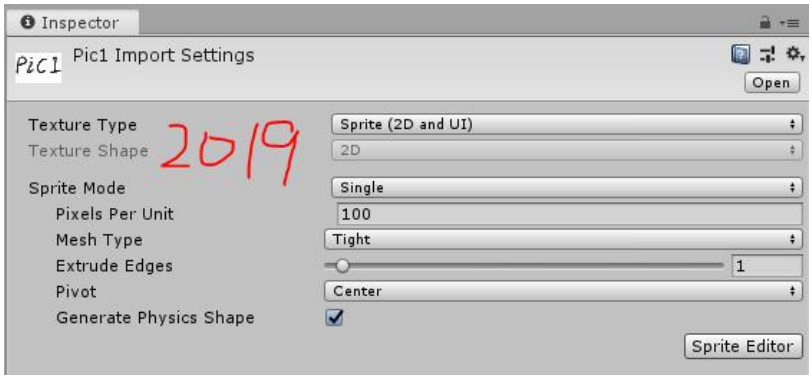
Appendix

Auto asset processing, the Editor code written in AssetBundleBuildWindow.cs, we do process some assets :

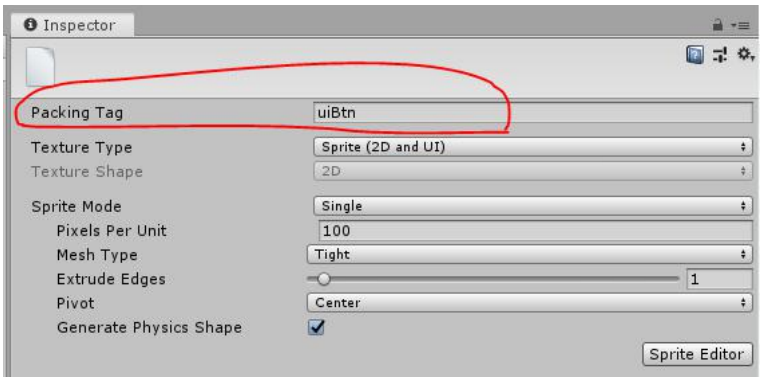
1. SpriteAtlas, you can not create SpriteAtlas yourself in Unity5, it can be created after Unity2017, code in AssetBundleBuildWindow.cs::CreateSpriteAtlasAssets function, it do set all Sprites with same Packing Tag to the same AssetBundle avoid the SpriteAtlas duplicated(works in Unity5). The Packing Tag is the basis for packing AssetBundle, but in Unity2019/Unity2020 the Packing Tag do not shows up in the Inspector, but it exists in the serialize file. AssetBundleMaster overwrite the Inspector for it, Packing Tag was added, the script is TextureImporterInspector.cs :



Unity5 has the Packing Tag field

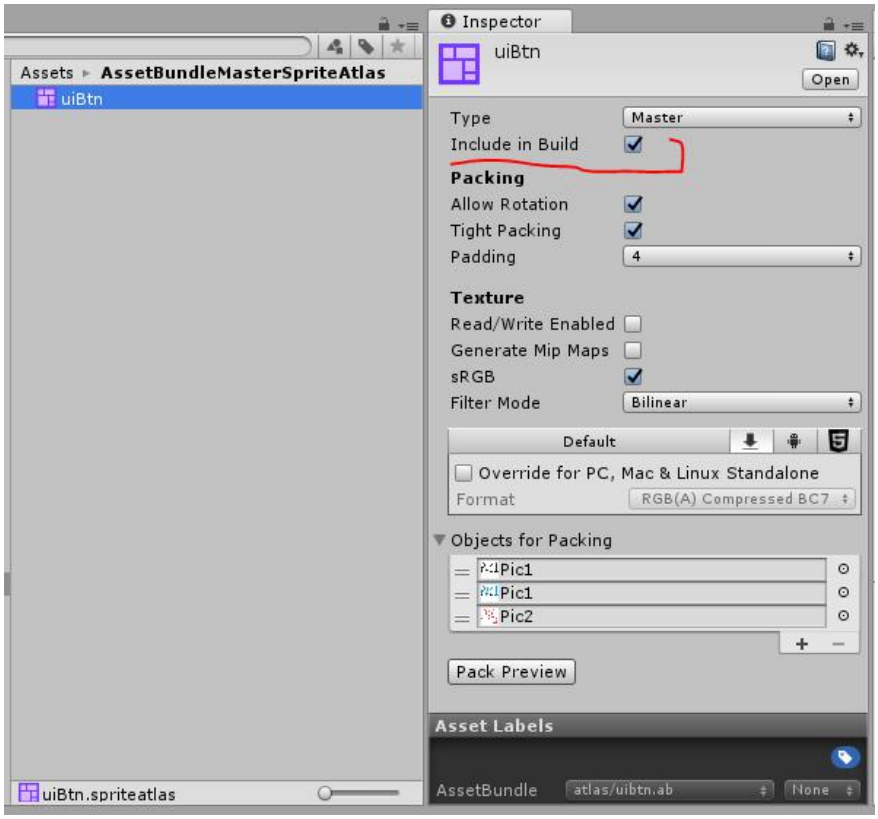


Unity2019 has no Packing Tag



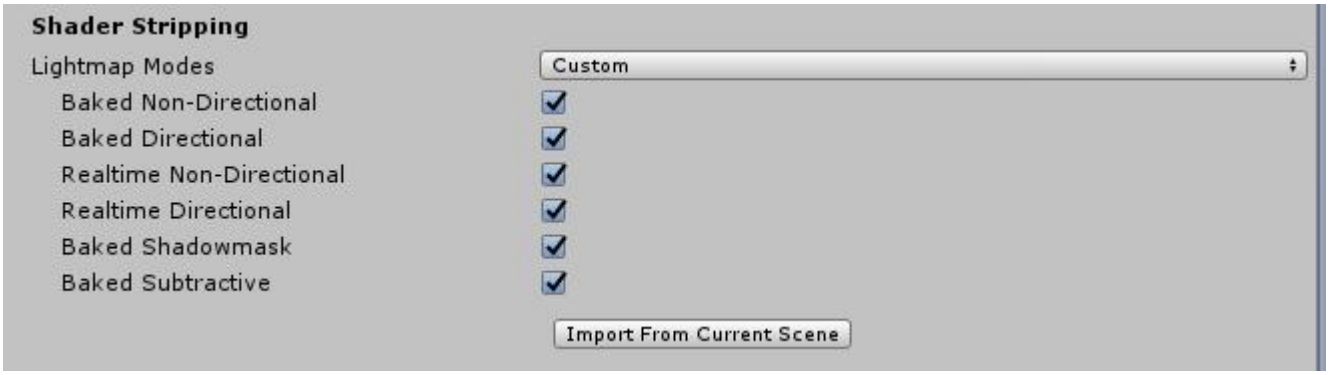
Unity2019 or newer AssetBundleMaster overwrite Texture Inspector

In some cases the [Include in Build] info of SpriteAtlas may lost (for exp AssetBundle update), ResourceLoadManager added a callback to UnityEngine.U2D.SpriteAtlasManager.atlasRequested as a fallback to ensure the Sprites will be load correctly, if you got the issue that can not load Sprite, debug the ResourceLoadManager.RequestAtlas function(Unity2017 or newer).



SpriteAtlas will be created to Assets/AssetBundleMasterSpriteAtlas folder(Unity2017 or newer).

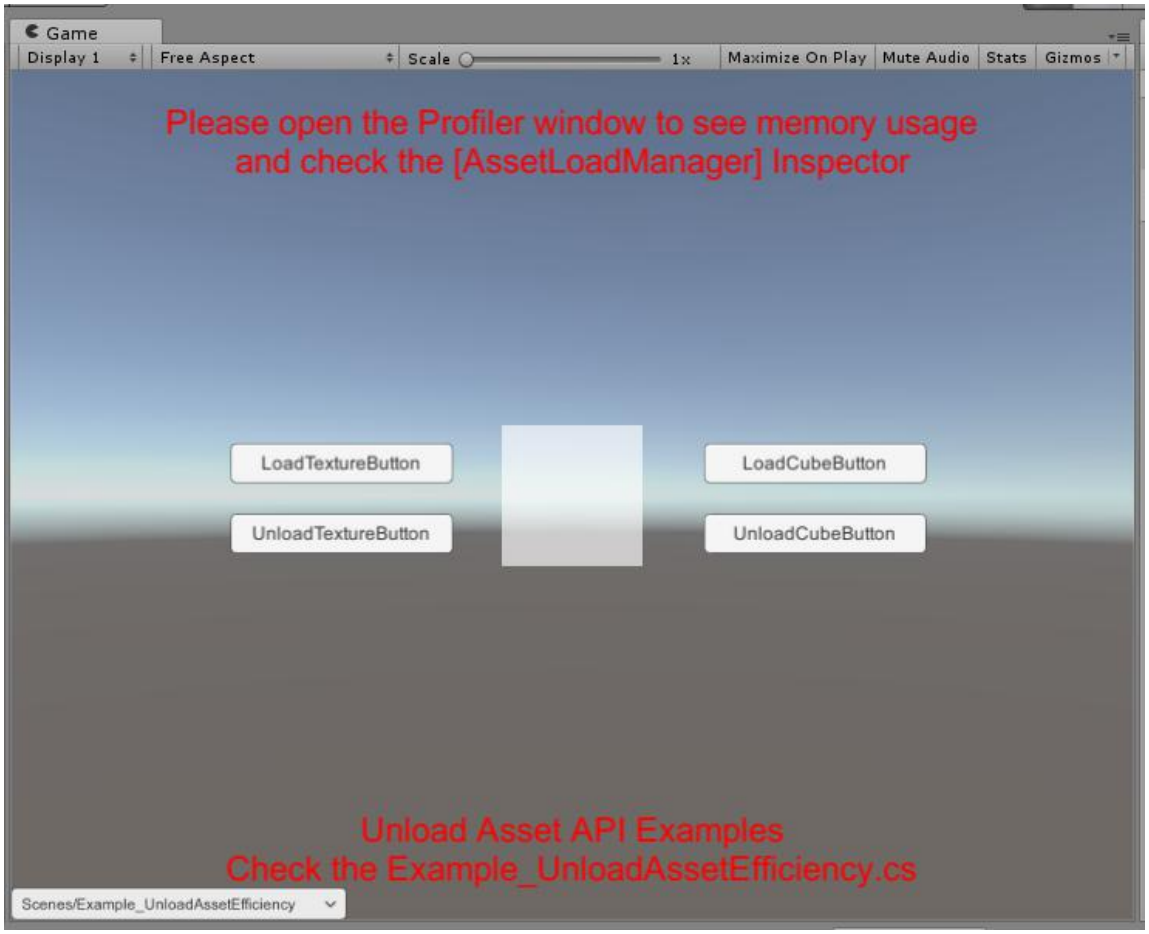
2. EditorConfigSettings, some of the editor settings will be set, you can take a look codes at AssetBundleBuildWindow.cs::EditorConfigSettings function, it did set the UnityEditor.EditorSettings.spritePackerMode for create SpriteAtlas, and change GraphicsSettings > Lightmap Modes to Custom mode, for preventing the LightMap info being stripped while build AssetBundle :



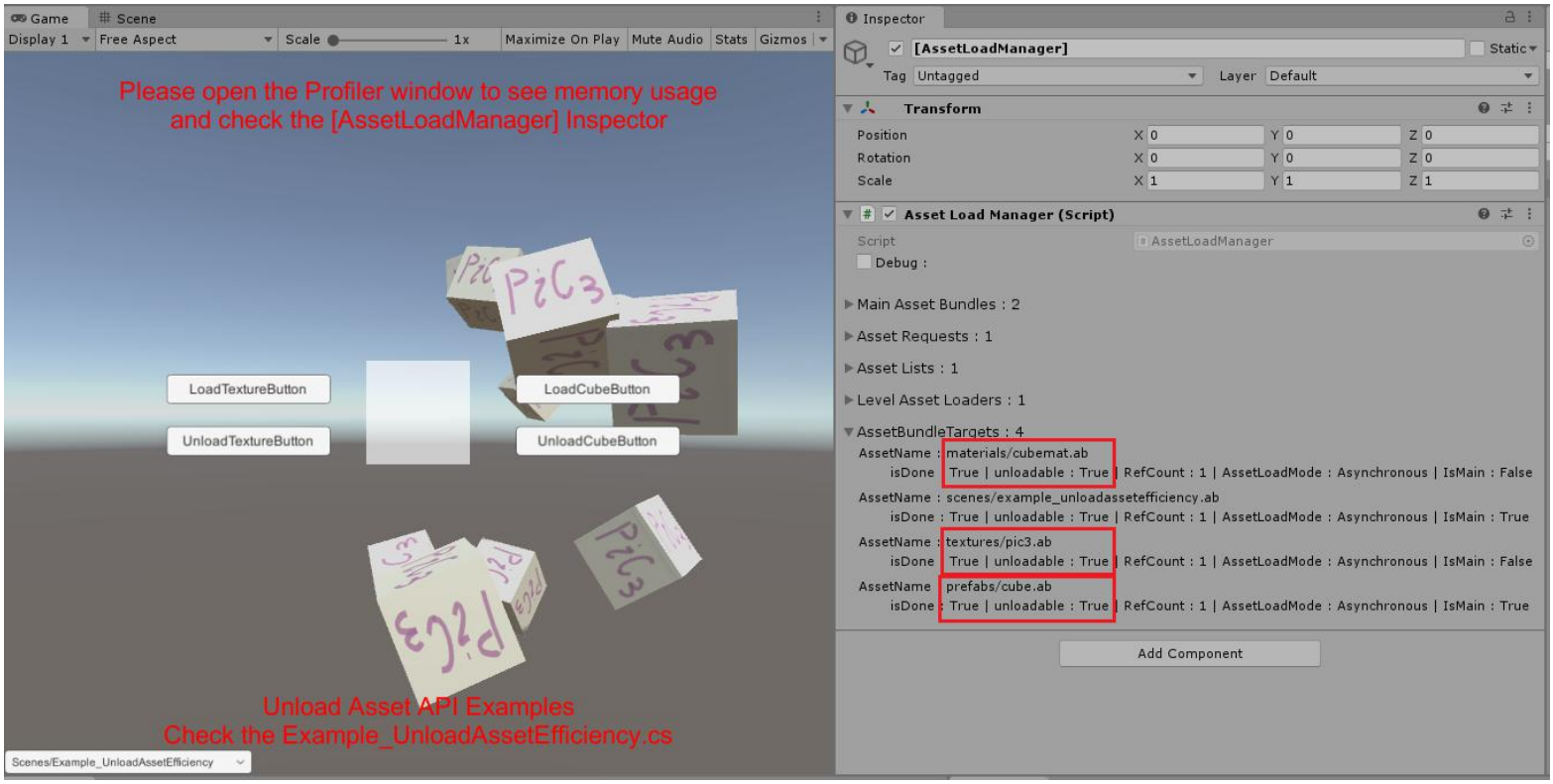
If you got some issues unexpect, check the code AssetBundleBuildWindow.cs::EditorConfigSettings();

3. Auto Resource Control

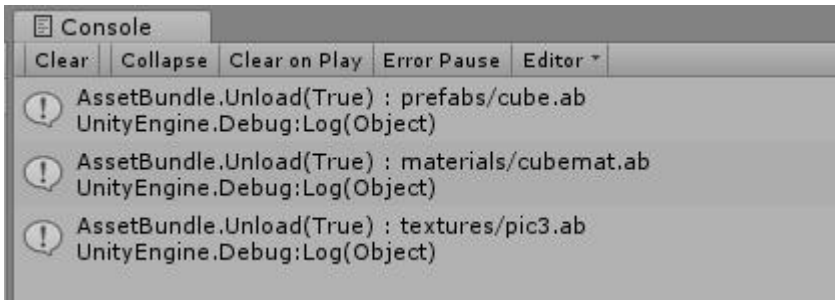
The logic of auto resource control in AssetBundle mode, It's far more complicated, but the core is to keep any asset will not duplicated in the memory, we use weakreference to listening asset is alive or not, if it is alive, we don't unload the assetbundle, on the contrary it is simple. we can see Demo : Example_UnloadAssetEfficiency. please select any AssetBundle mode and build AssetBundles, and select this scene to load assets :



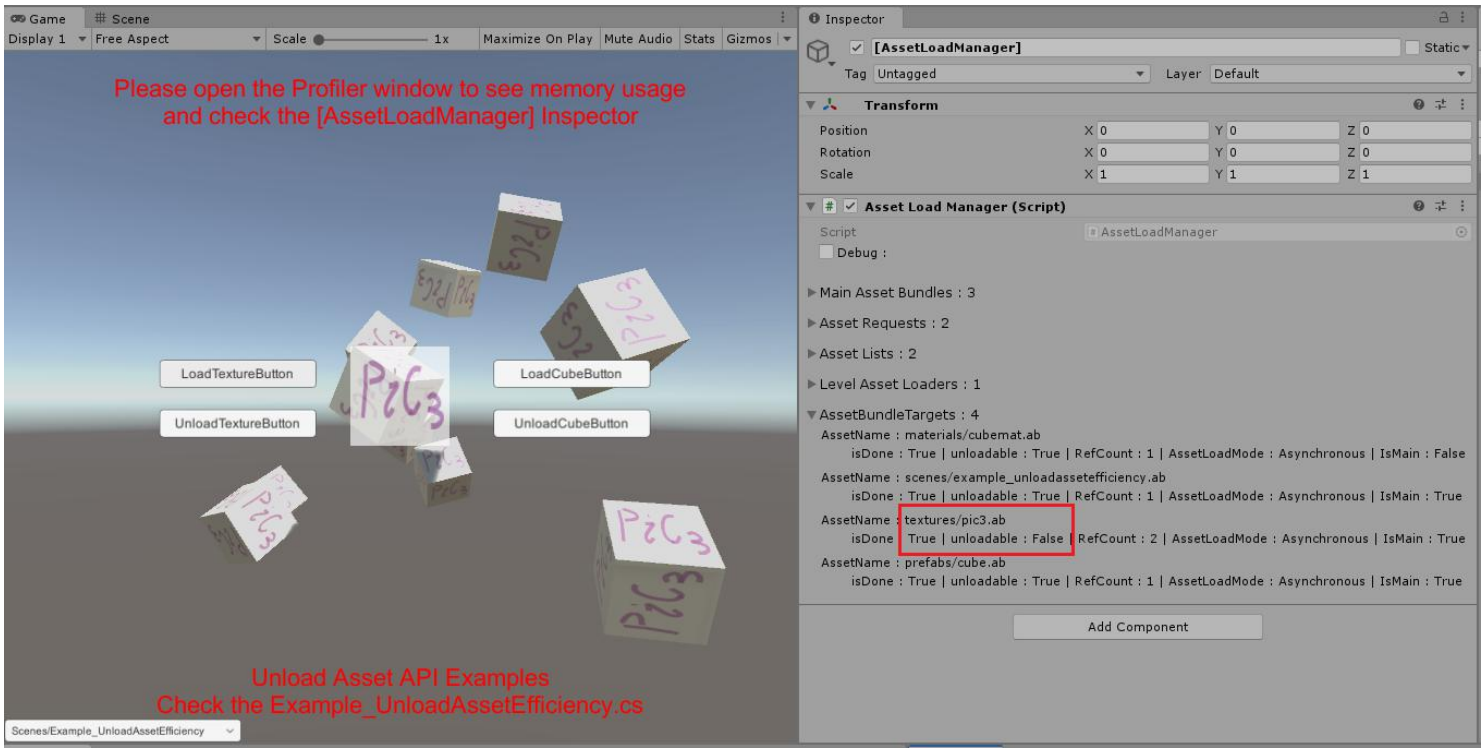
Click LoadCubeButton to load Cubes, we can see AssetBundles are auto controled:



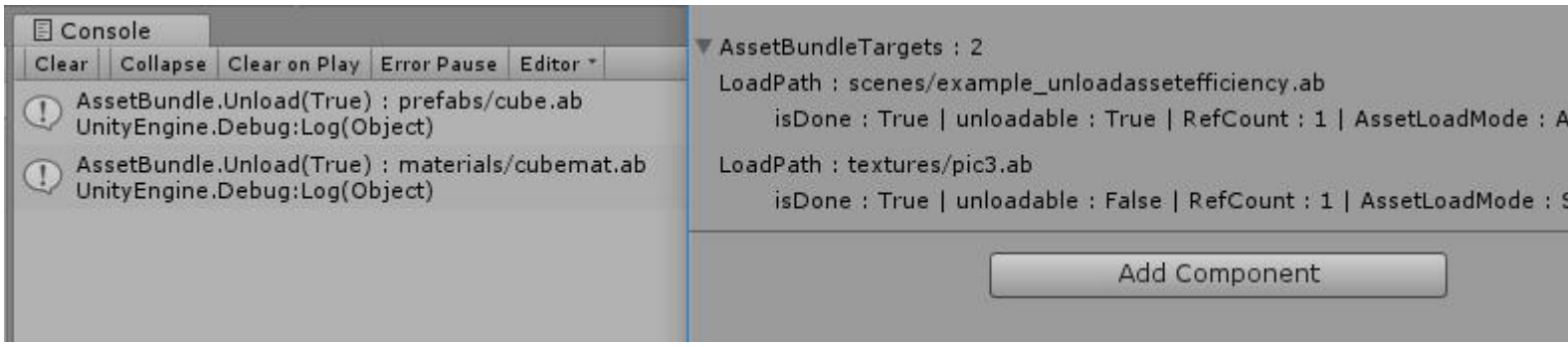
Click UnloadCubeButton, The Unload log shows AssetBundle.Unload(true); was called:



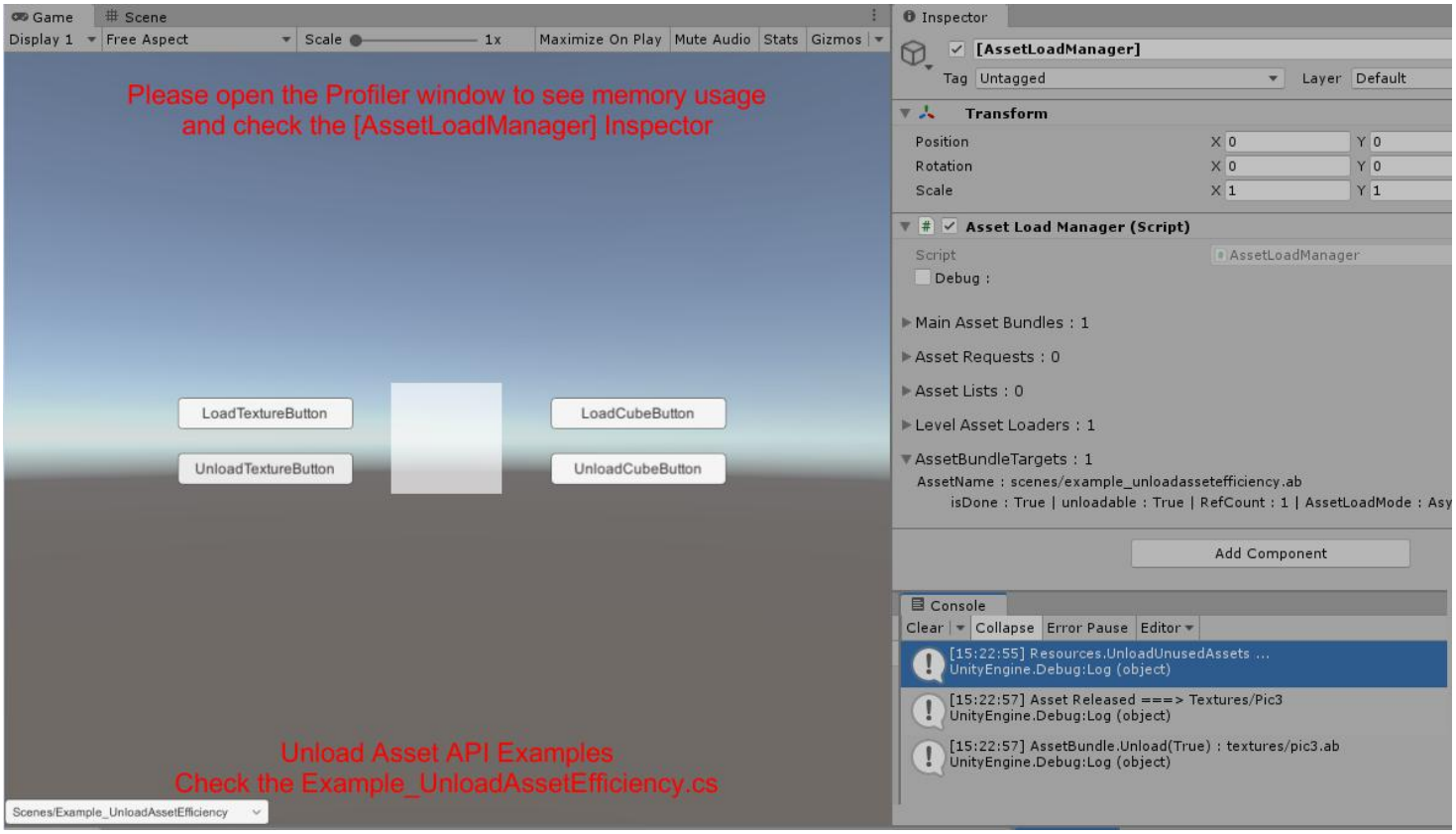
This time we do load both assets, Cube and Pic3, Click LoadCubeButton, LoadTextureButton:



We can see textures/pic3.ab was set to unloadable : False, it means Pic3 asset was set to not auto control asset, Click UnloadCubeButton, you can see the unload log:



Only 2 assets was unloaded, the AssetBundle textures/pic3.ab was kept, because it loaded by ResourceLoadManager and was referenced by it, Click UnloadTextureButton:

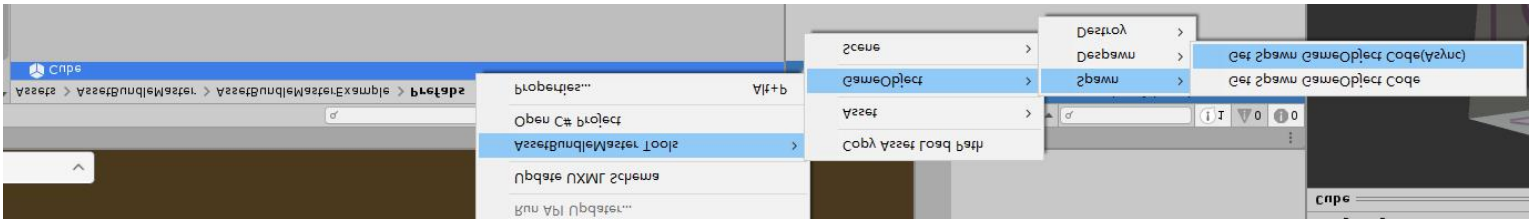


The unload process is start from Resources.UnloadUnusedAssets(); and get a weak reference check after that, if the resource was released, the AssetBundle will be unload later. This logic can make sure your assets will never be load duplicated.

That's the basic unload logic.

Tools :

We added some tools for fast code generate. You can right click in Project window, see AssetBundleMasterTools selection.



For Example right click on Cube.prefab file, Select the [Get Spawn GameObject Code(Async)], it will do copy the follow code to your clip board :

```
AssetBundleMaster.ResourceLoad.PrefabLoadManager.Instance.SpawnAsync("Prefabs/Cube.prefab", (Cube_go)=>{ });
```

you can Ctrl + V to paste it to you script. if you have no asset selected or incorrect selected(not under the Asset Root folder), it will copy the code without loadpath filled :

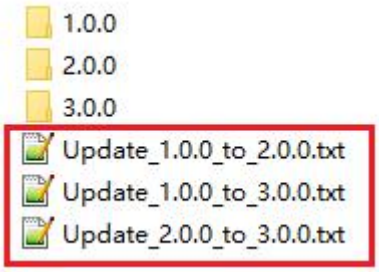
```
AssetBundleMaster.ResourceLoad.PrefabLoadManager.Instance.SpawnAsync("", (Cube_go)=>{ });
```

Yes, quite simple. less coding.

Update support :

1. We are not to implement any file update system. but as we can see the update-patch file can be generated by 6. Build Update File (CheckBox) selected, no matter which version of AssetBundles you build or rebuild, all update patch files will be refresh again, so your update system can use it.

Assets/../AssetBundles/[Platform]/



How to get current assetbundle version is to get VersionInfo from LocalVersion :

```
var version = AssetBundleMaster.AssetLoad.LocalVersion.Instance.versionInfo.BundleVerison;
Debug.Log(version);
```

It is a string, you can send it to server to quary a update patch file, and run your update system if needed.

2. If you select AssetBundle_PersistantDataPath mode, the AssetBundleMaster will try to load assetbundle from Application.persistentDataPath first, if there is no target file exists, AssetBundleMaster will try to load from Application.streamingAssetsPath. It means you can publish a basic version app such as [1.0.0] or [2.0.0] with assetbundles in StreamingAssets folder, and you can download new version files to Application.persistentDataPath, the runtime will load from this path if new version file exists. Most platforms can Read/Write to persistentDataPath, your project is updatable in this mode(but AssetBundleMaster not implement any file update system, you must make it your self).

3. If you select AssetBundle_Remote mode, witch means you load assetbundle from remote address, you don't need to make any file update system, it will load newest assetbundle from remote or local cached(remote asset will cache to local).