FORTUNE MINE CASE ASSIGNMENT

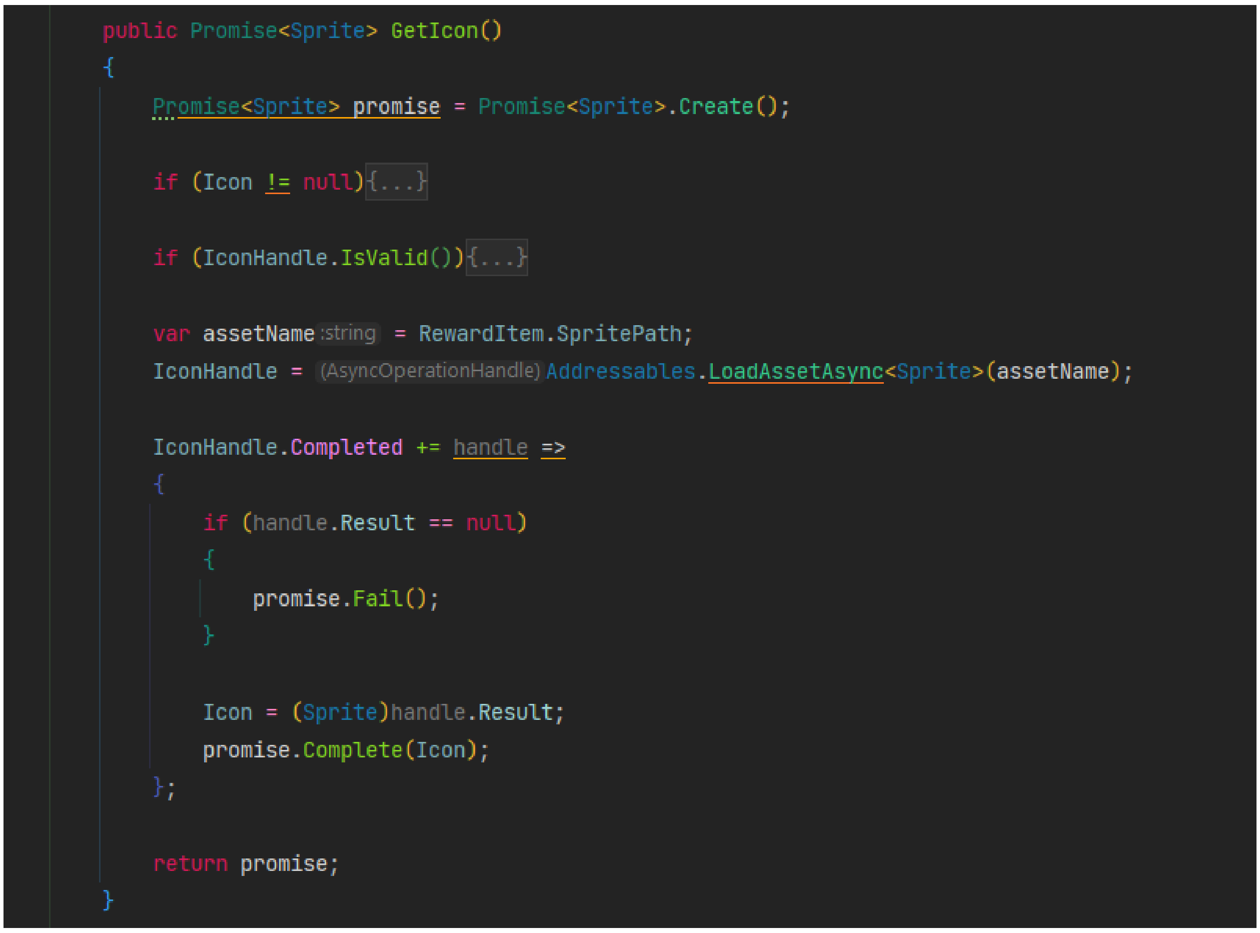
Project Overview

by Şeyma Aybala Pamuk

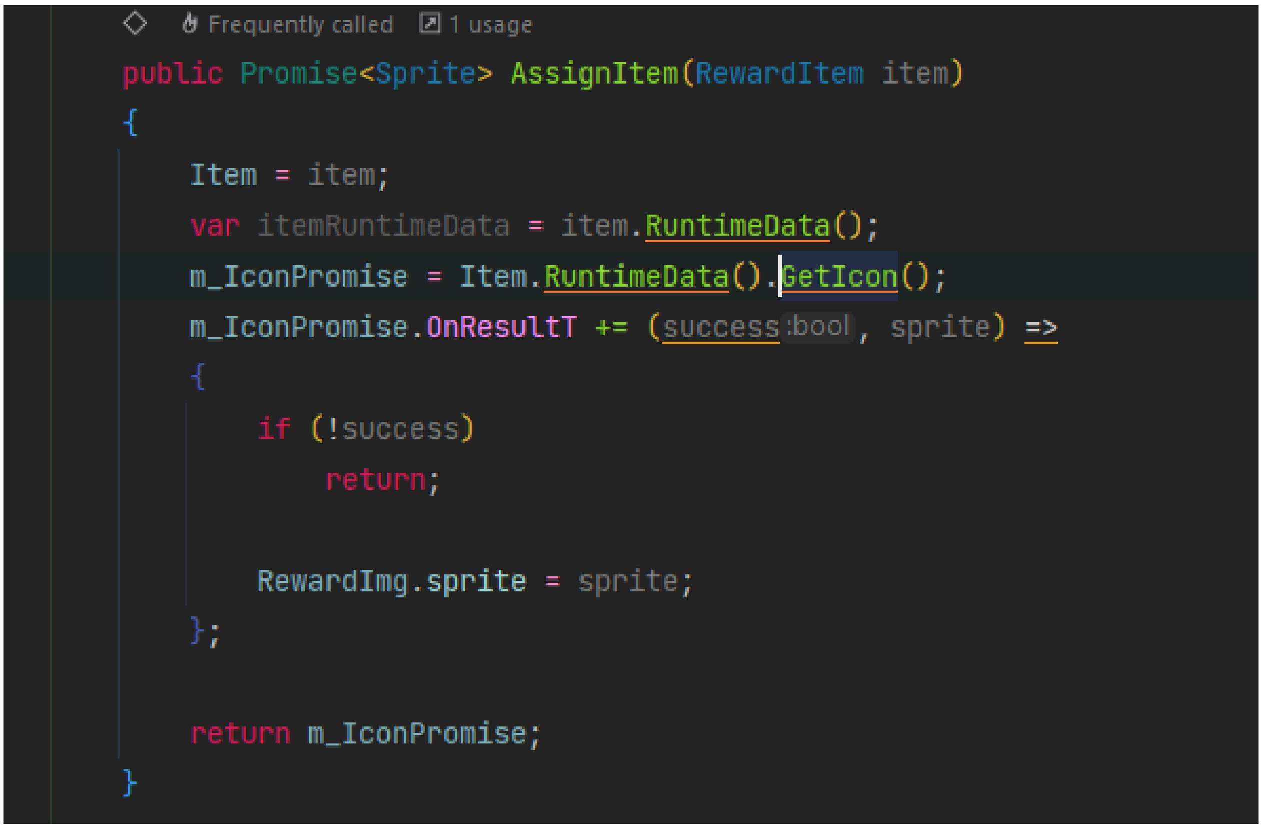
The project is based on 4 major structures: Reward Items, Roulette, Wallet and Scene Management and additionally, Addressables.

Reward Items & Addressables

I developed a class called **RewardItem**, which inherits from ScriptableObject, to encapsulate data about reward icons. This design was influenced by my previous projects. RewardItem contains essential information such as name, icon sprite reference, material reference, addressable path, and an ID. Another class, **RewardItemCreator**, located within the Resources folder, is responsible for creating RewardItem objects from sprite atlases found under the designated path, utilising Addressable references.
For optimization, I put the reward icons into a **Sprite Atlas** which is **Addressable**. Normally, with sprite atlases, Unity would load all of the atlas at the same, which might not be a problem for this case because I assumed that we are using reward sets and the rewards in one set will always be used together. However, by using sub-asset referencing as explained in this article, we can still get the best of both worlds by both using sprite atlases and being able to only load the part of the atlas which is needed, which is what I have implemented in this project. I also hold the string for the sprite atlassed icons path so that it can be calculated only once and we don't have to do string computations on runtime.
I also created materials for these RewardItem objects for the particle animation. Again, instead of changing a constant materials texture on the runtime, which results in creating a bunch of instances, I chose to create a material for each of them and assign it as Addressable, which is actually done by hand right now. This is the only part of the RewardItem creation which is not automated but could be.
So to sum it up, inside a RewardItem scriptable, the asset references of its corresponding icon sprite and material are held. These data objects are held inside **RewardItemCollections** which just hold a list of objects. When being used on the runtime, there is a wrapper class called **RewardItemRuntimeData** which is Disposable. These are basically created (or called if already created) by **RewardItemManager** when needed and disposed at the appropriate times. RewardItemRuntimeData is responsible for getting the Addressable data of a RewardItem. Here is one example of where I used a third-party implementation of **Promises** used in Javascript. For example, when the icon of the RewardItem is needed, **GetIcon** method inside the RewardItemRuntimeData is called. This method returns a promise which can take a sprite inside. The caller of the GetIcon method adds the action wished to the OnResultT event of the promise returned by method. Promises are used for any asynchronous operation.



```
public Promise<Sprite> GetIcon()
{
    Promise<Sprite> promise = Promise<Sprite>.Create();

    if (Icon != null){...}

    if (IconHandle.IsValid()){...}

    var assetName :string = RewardItem.SpritePath;
    IconHandle = (AsyncOperationHandle) Addressables.LoadAssetAsync<Sprite>(assetName);

    IconHandle.Completed += handle =>
    {
        if (handle.Result == null)
        {
            promise.Fail();
        }

        Icon = (Sprite)handle.Result;
        promise.Complete(Icon);
    };

    return promise;
}
```



```
    Frequently called   1 usage
public Promise<Sprite> AssignItem(RewardItem item)
{
    Item = item;
    var itemRuntimeData = item.RuntimeData();
    m_IconPromise = Item.RuntimeData().GetIcon();
    m_IconPromise.OnResultT += (success :bool, sprite) =>
    {
        if (!success)
            return;

        RewardImg.sprite = sprite;
    };

    return m_IconPromise;
}
```

Roulette Management

The roulette management is pretty straightforward. **RouletteController** holds the **RouletteSlot** objects in the scene and also the RewardItemCollection for which set to use, if there were more sets being used, this would be assigned somewhere else but right now it is just referenced from the editor.
RouletteController assigns one RewardItem to each of the slots in the beginning of the scene. This could also be constant but I chose to have them randomly.
Here I can touch on the Event system used in this project; it is a 3rd party asset and it uses pooling to manage the events for extra optimization. To keep modularity as much as possible, almost every communication between classes is made through these events. Also for modularity and for flexibility, I refrained from using hard-coded values and instead used another scriptable object called **GeneralSettings** to hold them. For example GeneralSettings holds the duration for the animation of particles moving towards the wallet icon so that the RouletteSlot which was selected can play its selected animation after the particle animation is complete.
Objects related to roulette management are all inside the **Barbeque** scene.

Wallet Management

Wallet class holds a dictionary with RewardItem and int pair.
WalletUIController listens to a WalletEvent in the context of "Earn" which is done by an enum. On every earn event, WalletUIController activates the particle animation where reward icons move from the roulette to the wallet icon. When the wallet icon is clicked, a quick popup shows all the rewards earned.
For popups, which are also used when a reward is used, I created a base abstract class called **PopupController** which implements **IPopup** interface. Right now, having both an abstract class and an interface might be redundant but for future uses where popups might be more specialised, this structure can be useful.
Objects related to wallet management are all inside the **Shared** scene.

Scene Management

The project consists of 3 scenes: Shared, Initial, Barbeque. Shared scenes purpose is to hold objects which should be persistent through all scenes. Initial scene is just a start scene and the Barbeque scene is our game scene. The **SceneTransitionManager** enables two types of scene management. Firstly, by using **SceneController**s, we can activate and deactivate additive scenes so for smaller or frequently used scenes, like Initial scene, the approach is to load them additionally once in the beginning of the session and activate/deactivate them as needed. For more heavy scenes which should not be accessed very frequently, like a mini-game like the Barbeque scene, the approach is to load them on use and unload them and their related data on completion.