MoeFramework框架说明

xrjervis

目　录

Overview

Overview

MoeFramework is a "Manager of Managers" based unity light framework designed for creating and managing a game more easily.

- MoeFramework
  - Core
    - GameManager
    - LevelManager
    - SaveManager
  - Events
    - EventManager
  - Singleton
    - MonoSingleton
    - Singleton
  - UI
    - GUIManager
    - UIBase
    - UIRootHandler
  - Utility
    - Utility

If you have any questions please contact author Rui Xie [xrjervis at gmail.com]. Thank you for your precious advice.

Core

Core

Core folder includes the most important files of this framework. They are designed to complete tasks including initiating, loading levels, managing global variables, etc.

GameManager

The details in GameManager script can be very flexible. It deals with global variables and defines the attributes needed. You may also declare the instance of SaveData and write the specific operations in Load() and Save() methods.

GameManager inherits from MonoSingleton and receives unity lift-cycle.

If save data exists, then Load() method will be executed to retrieve the data from the existing file. But if not, the script will load initial data from .csv file or generate test data from Utility script.

```
if (File.Exists(Application.persistentDataPath + "\\" + _saveDataFileName + ".uml") == false) {
    Utility.TestRelate.GenerateTestData();
    //Or Load data from .csv file
}

Load();
```

It contains const strings to indicate the root path of resources.

```
public const string BACKGROUND_TEXTURE_PATH = "Textures/BackgroundTexture/";
public const string MATERIAL_TEXTURE_PATH = "Textures/MaterialTexture/";
public const string INGREDIENT_TEXTURE_PATH = "Textures/IngredientTexture/";
public const string GIFT_TEXTURE_PATH = "Textures/GiftTexture/";
public const string SNACK_TEXTURE_PATH = "Textures/SnackTexture/";
public const string DRUG_TEXTURE_PATH = "Textures/DrugTexture/";
public const string UI_PREFAB_PATH = "Prefabs/UI/";
public const string CASH_TEXTURE_PATH = "Textures/CashTexture/";
public const string MAP_TEXTURE_PATH = "Textures/MapTexture/";
```

GameManager also contains some other methods like RemoveFromList(), AddToPlayerList(), AddToPlayerAttribute(), etc. which are invoked by other controller scripts outside the MoeFramework. This means that you can only modify the global variables through these methods in GameManager script which can protect the security to some extents.

LevelManager

LevelManager is responsible for the managment of all the levels in this game.

It stores all the scene status by the sceneStatus enum type.

```
public enum SceneStatus{
        Home,
        Sale,
        Purchase
}

public SceneStatus CurrentScene;
```

It contains variables which redirect the name(string type) of each scene such that you can just use these variables to find each scene instead of use several strings. When you need to modify the name of a scene, you do not need to replace every string you have used, which brings convenience.

```
public string homeScene = "Home";
public string saleScene = "Sale";
public string purchaseScene = "Purchase";
```

And it jumps to another scene when you invoke the LoadNext() method. The method can jump to the correct scene according to the switch-case statements. You can also configure your loading strategies, like the ways or the orders to load the next level.

```
public void LoadNext(string button) {
        switch (button) {
                case "MapButton":
                        SceneManager.LoadScene(saleScene);
                        CurrentScene = SceneStatus.Sale;
                        break;
                case "GoToPurchaseButton":
                        SceneManager.LoadScene(purchaseScene);
                        CurrentScene = SceneStatus.Purchase;
                        break;
                case "GoHomeButton":
                        SceneManager.LoadScene(homeScene);
                        CurrentScene = SceneStatus.Home;
                        break;
        }
}
```

SaveManager

SaveManager supports most of the data type in unity with C# including List.
On Windows, the save data file can be found under
C:\Users\YourUserName\AppData\LocalLow\MS\YourAppName

Instruction

First, you need to create the data instance.

```
data = new SaveData(fileName);
```

Then you may add keys with significant names and values.

```
data["Name"] = "Rui Xie";
data["Dude"] = "Siri";
data["Key"] = true;
data["HealthPotions"] = 10;
data["Position"] = new Vector3(20, 3, -5);
data["Rotation"] = new Quaternion(0.1f,0.1f,0.1f,1);
```

Save the data.

```
data.Save();
```

Load the data we just saved.

```
data = SaveData.Load(Application.streamingAssetsPath+"\\"+fileName+".uml");
```

Use data.

```
Debug.Log("Name : "+ data.GetValue<string>("Name"));
Debug.Log("Has health potions : " + data.TryGetValue<int>("HealthPotions", out potions));
Debug.Log("Has buddy : " + data.HasKey("Dude"));
Debug.Log("Buddy's name : " + data.GetValue<string>("Dude"));
Debug.Log("Current position : " + data.GetValue<Vector3>("Position"));
Debug.Log("Has key : " + data.GetValue<bool>("Key"));
Debug.Log("Rotation : " + data.GetValue<Quaternion>("Rotation"));
```

Events

Events

During the development, you may find it is a huge challenge to deal with all the button clicking events, timer trigger events and so forth. Therefore, constructing a message system is necessary, so you do not need to write logic codes inside the OnButtonClick() method or something like that, which keeps the codes clean and easier to maintain.

EventManager

Using C# delegates and UnityEvents, EventManager implements a simple messaging system which will allow items in the project to subscribe to events, and have events trigger actions in our games. This will reduce dependencies and allow easier maintenance of our projects.

Instruction

First, you need to bind an event with a function by assigning a string to the function using EventManager.StartListening(string eventName, Action listener).

```
void OnEnable()
{
    EventManager.StartListening("Test", SomeFunction);
    EventManager.StartListening("Spawn", SomeOtherFunction);
    EventManager.StartListening("Destroy", SomeThirdFunction);
}
```

If you want to stop listening, just invoke EventManager.StopListening(string eventName, Action listener).

```
void OnDisable()
{
    EventManager.StopListening("Test", SomeFunction);
    EventManager.StopListening("Spawn", SomeOtherFunction);
    EventManager.StopListening("Destroy", SomeThirdFunction);
}
```

Then in some other scripts you may define the specific operations in each funtion you need.

```
void SomeFunction()
{
    Debug.Log("Some Function was called!");
}
```

Also, you should use TriggerEvent(string eventName) to invoke an event. The following script below test the event by triggering events every 2 seconds.

```
void Awake()
{
    someListener = new Action(SomeFunction);
    StartCoroutine(invokeTest());
}

IEnumerator invokeTest()
{
```

```
    WaitForSeconds waitTime = new WaitForSeconds(2);
    while (true)
    {
        yield return waitTime;
        EventManager.TriggerEvent("test");
        yield return waitTime;
        EventManager.TriggerEvent("Spawn");
        yield return waitTime;
        EventManager.TriggerEvent("Destroy");
    }
}
```

Singletons

Singletons

As this framwork is based on "Manager of Managers", thus each manager is an implementation of a singleton. Though you can implement by using static class, using singleton is somehow much more common and can adapt most of situations.

Be aware that you need to create an empty GameObject in the initial Unity scene and drag all the singleton managers on it.

Singleton

If you need 10 various manager, you need to copy the same code 10 times causing a huge mess! Then you need to bring in generic type.

```
public abstract class Singleton<T> where T : Singleton<T> {
    protected static T _instance = null;
    protected Singleton() {     }

    public static T Instance() {
        if (_instance == null) {
                ConstructorInfo[] ctors = typeof(T).GetConstructors(BindingFlags.Instance | BindingFlag
s.NonPublic);
            ConstructorInfo ctor = Array.Find(ctors, c => c.GetParameters().Length == 0);
            if (ctor == null)
                throw new Exception("Non-public ctor() not found!");
            _instance = ctor.Invoke(null) as T;
                }

                return _instance;
        }
}
```

The script below shows how Singleton works.

```
public class XXXManager : Singleton<XXXManager> {
    private XXXManager() {
        // to do ...
    }
}

public static void main(string[] args)
{
    XXXManager.Instance().MyMethod();
}
```

MonoSingleton

In MonoSingleton, we need to do the following things:

1. Restrict the number of the instance object.
2. Restrict the number of GameObject.
3. Receive the MonoBehavior life-cycle.
4. Destroy singleton and its corresponding GameObject。

Be aware that the MonoSingleton has already had DontDestroyOnLoad() method, therefore any class inheriting from MonoSingleton do not need to include DontDestroyOnLoad() any more.

```
public abstract class MonoSingleton<T> : MonoBehaviour where T : MonoSingleton<T> {
        protected static T _instance = null;

        public static T Instance() {
                if (_instance == null) {
                        _instance = FindObjectOfType<T>();

                if (FindObjectsOfType<T>().Length > 1) {
                        return _instance;
                }

                if (_instance == null) {
                        string instanceName = "GodManager";
                        GameObject instanceGO = GameObject.Find(instanceName);

                        if (instanceGO == null) {
                                instanceGO = new GameObject(instanceName);
                                _instance = instanceGO.AddComponent<T>();
                                DontDestroyOnLoad(instanceGO);
                        }
                }

                return _instance;
        }

        protected virtual void OnDestroy() {
                _instance = null;
        }
}
```

UI

UI

In order to manage all UI panels based on Unity uGUI system including open, close, show, hide panels dynamically, UI folder contains GUIManager, UIBase and UIRootHandler script which will work together to complete those tasks.

Rules

1. All the UIPrefab needs to be placed under UIprefabs folder.
2. The name of UIPrefab needs to be kept the same as its class.
3. All the root node of UIPrefab is full screen.
4. All button click events must be encapsulated.
5. All node finding methods must be encapsulated.

All UIPrefab must have a corresponding scirpt. For example, a UI_Produce prefab must have a script named UI_Produce.cs.

GUIManager

GUIManager can open, close, show, hide UI panels dynamically at any time. The advantage is that you do not have to get these jobs fixed into the scene. And you may search and modify these panels more conveniently which increase reusability.

Instruction

It contains the panel status to indicate which panel the game currently shows.

```
public enum PanelStatus {
        Loading,
        Home,
        Sale,
        Purchase,
        Bank,
        Dairy,
        Hospital,
        Produce,
        Achievement,
        IngredientStore,
        SnackStore,
        GiftStore,
        MaterialStore,
        Map
}
```

It contains a list to store all the UI panels in the scene. So you can check this list to get a UI panel instance.

```
public List<UIBase> m_UIList = new List<UIBase>();
```

In ShowUI method, we need to check if the UI prefab is null. Then we invoke InstantiateGameObject to show UI panel. Next, we need to remove "(Clone)" string of the panel's name because it is generated by unity automatically. And the ui instance must be added into the m_UIList.

```
public T ShowUI<T>() where T : UIBase {
        if (CheckCanvasRootIsNull())
                return null;

        GameObject loadGo = Utility.AssetRelate.ResourcesLoadCheckNull<GameObject>(GameManager.UI_PREFA
B_PATH + typeof(T));
        if (loadGo == null)
                return null;


        GameObject ui = Utility.GameObjectRelate.InstantiateGameObject(m_CanvasRoot, loadGo);
```

```
        ui.name = ui.name.Replace("(Clone)", "").Trim();
        T t = ui.AddComponent<T>();
        t.Init(t.transform);
        m_UIList.Add(t);
        return t;
}
```

When you need to close a panel, you should remove it from the m_UIList and then destroy it.

```
public void CloseUI(UIBase ui) {
        m_UIList.Remove(ui);
        GameObject.Destroy(ui.gameObject);

}
```

UIBase

All the UI panel need to inherit from UIBase. It has 4 virtual methods which can be overrided by its derived class. They are OnButtonClick(GameObject obj), GetNodeComponent(string name, GameObject obj), OnEnter(), OnExit().

Instruction

First, after you instantiate a ui panel, you need to add its script.

```
T t = ui.AddComponent<T>();
```

Then do not forget to initiate the ui panel. This step will traverse(breadth-first search) all the children of this ui panel and add listener to all buttons in it.

```
t.Init(t.transform);
```

In OnButtonClick(GameObject obj) method you may define the specific operations for some common buttons like BackButton, CancelButton because they always do the same things when you click them.

In GetNodeComponent(string name, GameObject obj) you may get the GameObject type of the children node if you need. For example:

```
public override void GetNodeComponent(string name, GameObject obj)
{
        switch (name)
        {
            case "Background":
                _backgroundImage = obj.GetComponent<Image>();
                _backgroundAnimator = obj.GetComponent<Animator>();
                break;
            case "PageText":
                _pageText = obj.GetComponent<Text>();
                break;
        }
}
```

Also, you may add you own codes in OnEnter() method and OnExit() method.

UIRootHandler

UIRootHandler only complete one task that help to indicate where the main canvas is.

It must be attached to the canvas object in the scene.

```
void Awake() {
        GUIManager.Instance().m_CanvasRoot = this.gameObject;
}
```

Utility

Utility is just like a tool box providing various useful methods for developers. It includes instantiating objects, loading resoureces, check availability and so forth. We annouce these method to static and seperate them by their function using "Struct", which will make the codes easier to read.

How to invoke —— Utility.[Category].[Method]. For example:

```
Utility.GameObjectRelate.ClearChildren(gameObject.transform);
```

Also, under TestRelate struct, you may put all your test data here. Therefore, you can simply invoke GenerateTestData() in GameManager to get all your test data.