Table of Contents

整体介绍

这个地方主要是聚焦我在学习游戏开发过程中的一些总结与经验

- Unity拓展编辑器入门指南

Unity里面比较出色我也很喜欢的一个功能就是它易于拓展的编辑器。一般来说拓展编辑器对于游戏运行效率不是有什么大的帮助，但是**有助于开发效率的提高**。

毕竟工欲善其事，必先利其器。

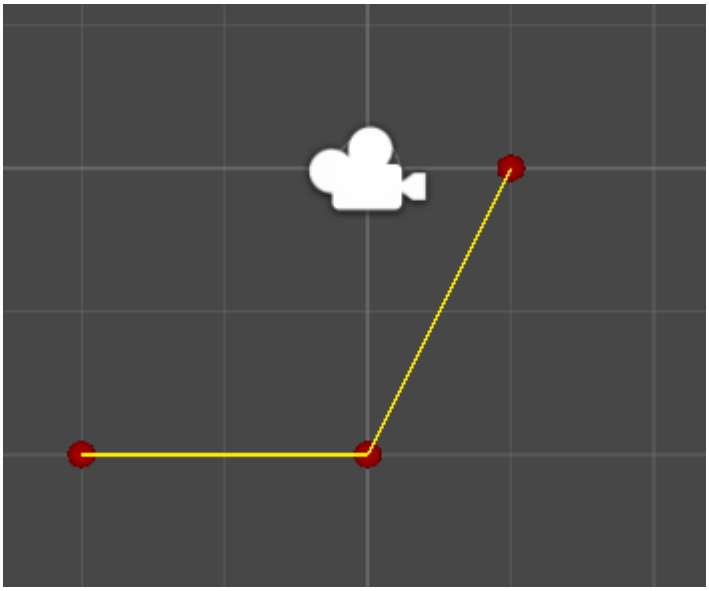
这次介绍一共以下这些拓展编辑器的方法：

- OnDrawGizmos
- OnInspectorGUI
- OnSceneGUI
- MenuItem 与 EditorWindow
- ScriptableWizard
- ScriptObject
- Attributes
- AssetProcess

OnDrawGizmos

OnDrawGizmos是在MonoBehaviour下的一个方法，通过这个方法可以可以绘制出一些Gizmos来使得其一些参数方便在Scene窗口查看。

比如我们有一个沿着路点移动的平台，一般的操作可能是生成一堆新的子物体来确定和设置位置，但其实这样会有点赘余，我们需要的只是一个**Vector2/Vector3**数组。而这个时候我们就可以通过**OnDrawGizmos**方法在编辑器绘制出这些**Vector2/Vector3**的数组点。

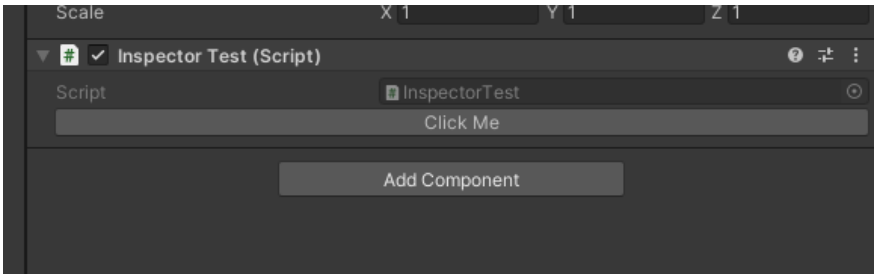


完整代码如下：

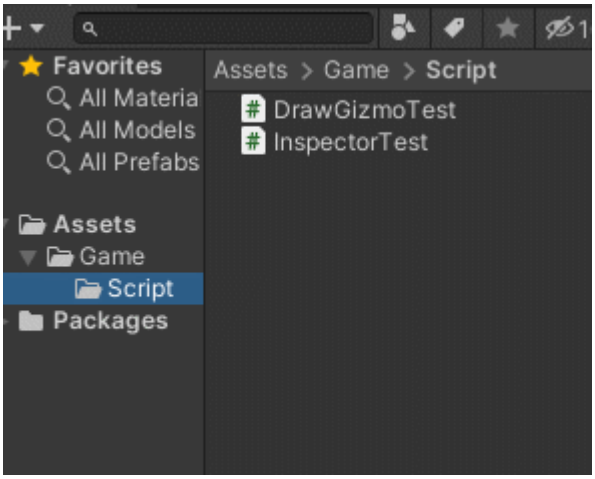
```
public class DrawGizmoTest : MonoBehaviour
{
    public Vector2[] poses;

    private void OnDrawGizmos()
    {
        Color originColor = Gizmos.color;
        Gizmos.color = Color.red;
        if( poses!=null && poses.Length>0 )
        {
            //Draw Sphere
            for (int i = 0; i < poses.Length; i++)
            {
                Gizmos.DrawSphere( poses[i], 0.2f );
            }
            //Draw Line
            Gizmos.color = Color.yellow;
            Vector2 lastPos = Vector2.zero;
            for (int i = 0; i < poses.Length; i++)
            {
                if( i > 0 )
                {
                    Gizmos.DrawLine( lastPos, poses[i] );
                }
                lastPos = poses[i];
            }
        }
        Gizmos.color = originColor;
    }
}
```

OnInspectorGUI

在开发过程中常常需要在编辑器上对某个特定的Component进行一些操作，比如在**Inspector**界面上有一个按钮可以触发一段代码。



这种属于编辑器的，所以一般是在Editor文件夹中新建一个继承自Editor的脚本：



之后编辑继承自**UnityEditor.Editor**,这里注意是必须在类上加入**[CustomEditor(typeof(编辑器脚本绑定的Monobehavior类)]**然后重写它的OnInspectorGUI方法：

```
using UnityEditor;
[CustomEditor(typeof(InspectorTest))]
public class InspectorTestEditor : Editor
{
    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
        if(GUILayout.Button("Click Me"))
        {
            //Logic
            InspectorTest ctr = target as InspectorTest;
        }
    }
}
```

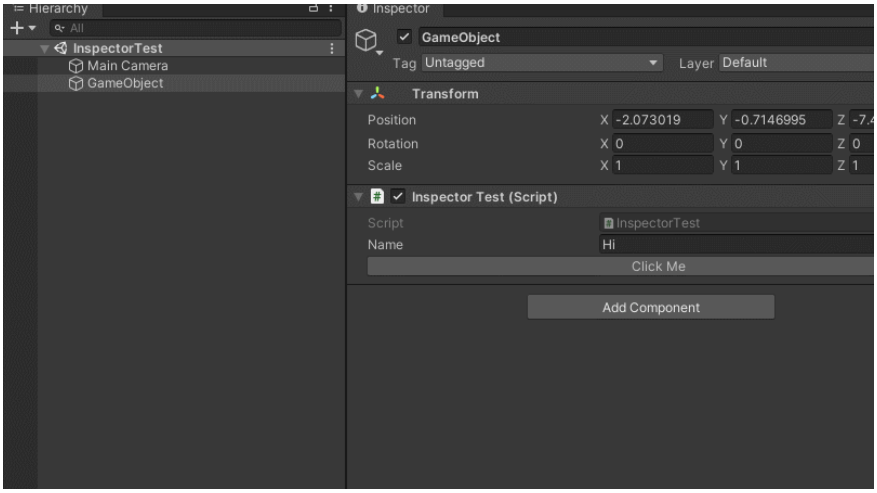
而一般而言在Editor类中操作变量有两种方式，一种是**通过直接访问或者函数调用改动Monobehaviour的变量**，一种是**通过Editor类中的serializedObject来改动对应变量**。

比如我要把Monobehaviour的一个公开的Name改成Codinggamer

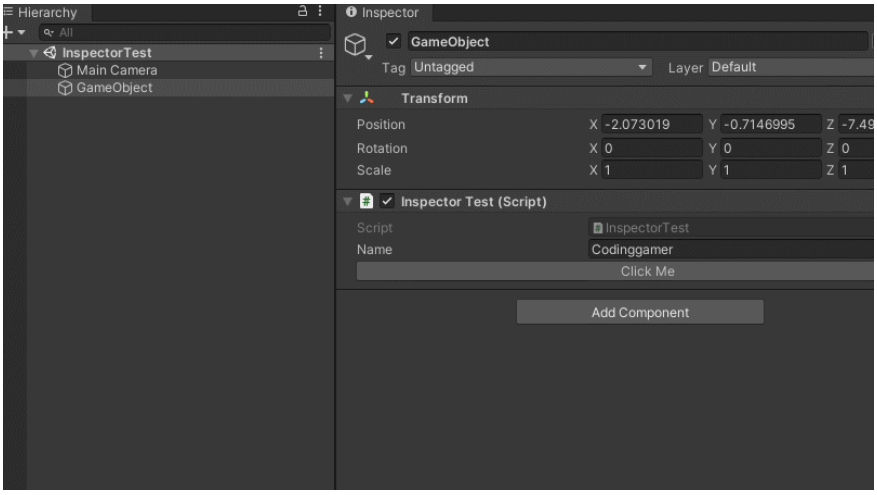
使用**方法一**，在Editor中可以这样写：

```
if(GUILayout.Button("Click Me"))
{
    //Logic
    InspectorTest ctr = target as InspectorTest;
    ctr.Name = "Codinggamer";
}
```

在编辑器中点击会发现Hierarchy界面没有出现一般改动之后会出现的小星星：



一般改动是会出现小星星:



如果这个时候你重新打开场景，会发现改动的值又便回原来的值，也就是**你的改动并没有生效**。

而此时，只需要再调用**EditorUtility.SetDirty(Object)**方法即可。

如果要使用**方法二**，则需要在Editor代码中写：

```
if(GUILayout.Button("Click Me"))
{
    //Logic
    serializedObject.FindProperty("Name").stringValue = "Codingg
    serializedObject.ApplyModifiedProperties();
}
```

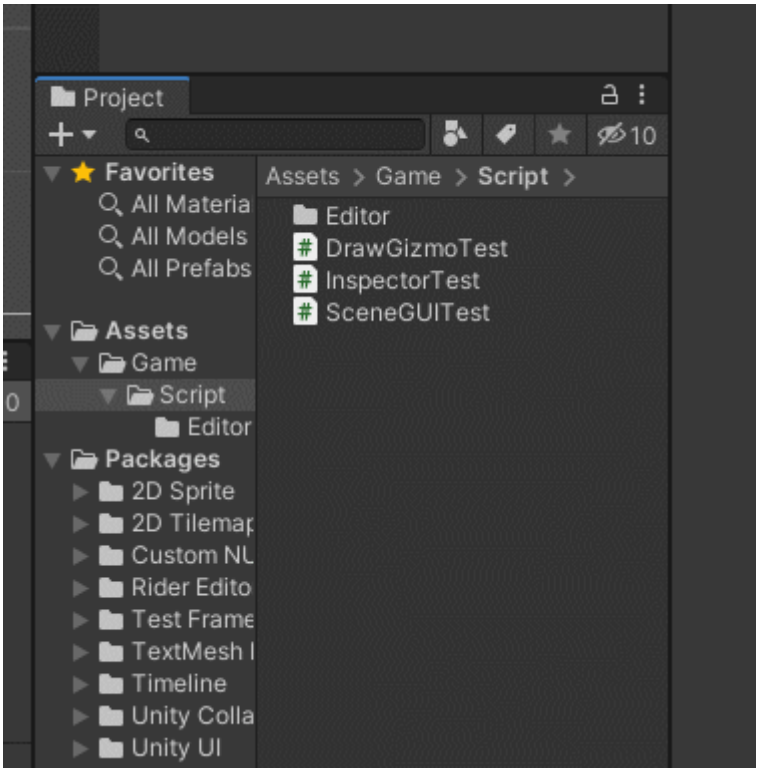
这里不需要调用**EditorUtility.SetDirty(Object)方法**,场景就已经会出现改动之后的小星星，保存重开场景之后也会发现对应值生效。

这两个方法孰优孰劣？

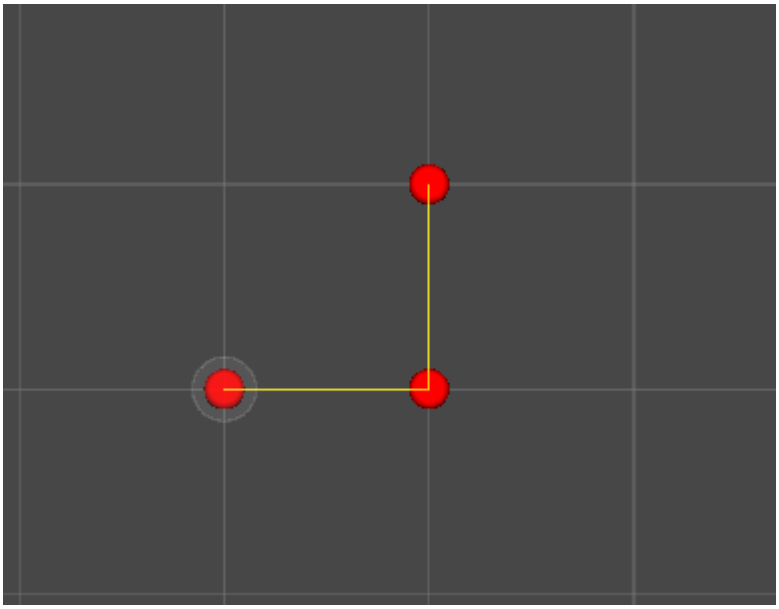
一般来说用第二个方法比较好，但实际上涉及逻辑比较多的时候我是用第一个方法。用第二个方法的好处在于它是**内置了撤销功能**，也就意味着你调用改动之后是可以直接撤销掉的，而第一个方法就不能。

OnSceneGUI

这个方法也是在Editor类中的一个方法，是用来在Scene视图上显示一个UI元素。其创建也是在Editor文件夹下新建一个继承自Editor的脚本:



在OnSceneGUI中可以做出和OnDrawGizmo类似的功能，比如绘制出Vector2数组的路点：



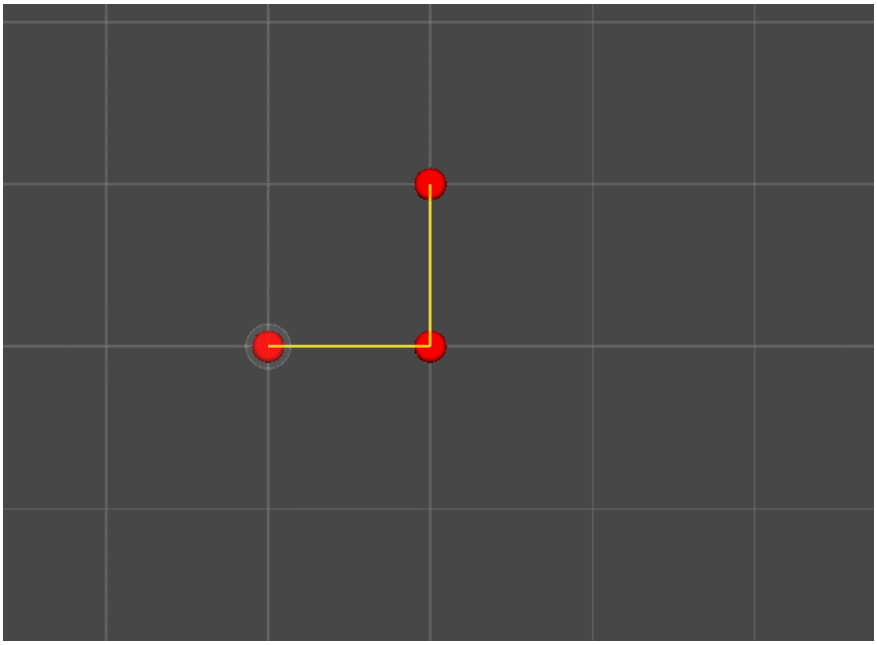
其代码如下：

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(SceneGUITest))]
public class SceneGUITestEditor : Editor
{
    private void OnSceneGUI()
    {
        Draw();
    }

    void Draw()
    {
        //Draw a sphere
        SceneGUITest ctr = target as SceneGUITest;
        Color originColor = Handles.color;
        Color circleColor = Color.red;
        Color lineColor = Color.yellow;
        Vector2 lastPos = Vector2.zero;
        for (int i = 0; i < ctr.poses.Length; i++)
        {
            var pos = ctr.poses[i];
            Vector2 targetPos = ctr.transform.position;
            //Draw Circle
            Handles.color = circleColor;
            Handles.SphereHandleCap(  GUIUtility.GetControlID(Fo
            //Draw line
            if(i > 0)
            {
                Handles.color = lineColor;
                Handles.DrawLine( lastPos, pos );
            }
            lastPos = pos;
        }
        Handles.color = originColor;
    }
}
```

OnDrawGizmos与OnSceneGUI的区别

因为OnSceneGUI是在Editor上的方法，而Editor一般都是对应Monobehaviour，这意味它是只能是点击到对应物体才会生成的。而OnDrawGizmos则是可以全局可见。

而如果需要事件处理，比如需要在Scene界面可以直接点击增加或者修改这些路点，就需要在OnSceneGUI上处理事件来进行一些操作。



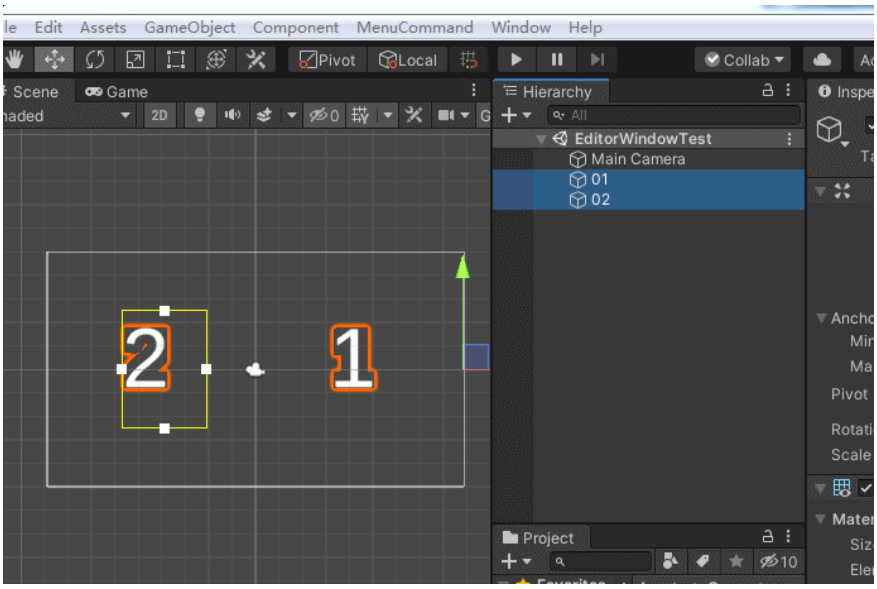
完整的代码如下，这里注意的是原来的poses为了方便改用成了List类型：

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(SceneGUITest))]
public class SceneGUITestEditor : Editor
{
    protected SceneGUITest ctr;

    private void OnEnable()
    {
        ctr = target as SceneGUITest;
    }
    private void OnSceneGUI()
    {
        Event _event = Event.current;

        if( _event.type == EventType.Repaint )
        {
            Draw();
        }
        else if ( _event.type == EventType.Layout )
        {
            HandleUtility.AddDefaultControl( GUIUtility.GetContr
        }
        else
        {
            HandleInput( _event );
            HandleUtility.Repaint();
        }
    }

    void HandleInput( Event guiEvent )
    {
        Ray mouseRay = HandleUtility.GUIPointToWorldRay( guiEven
        Vector2 mousePosition = mouseRay.origin;
        if( guiEvent.type == EventType.MouseDown && guiEvent.but
        {
            ctr.poses.Add( mousePosition );
        }
    }

    void Draw()
    {
        //Draw a sphere
        Color originColor = Handles.color;
        Color circleColor = Color.red;
        Color lineColor = Color.yellow;
        Vector2 lastPos = Vector2.zero;
```

```
        for (int i = 0; i < ctr.poses.Count; i++)
        {
            var pos = ctr.poses[i];
            Vector2 targetPos = ctr.transform.position;
            //Draw Circle
            Handles.color = circleColor;
            Vector2 finalPos = targetPos + new Vector2( pos.x, p

            Handles.SphereHandleCap(  GUIUtility.GetControlID(Fo
            //Draw line
            if(i > 0)
            {
                Handles.color = lineColor;
                Handles.DrawLine( lastPos, pos );
            }
            lastPos = pos;
        }
        Handles.color = originColor;
    }
}
```

MenuItem与EditorWindow

MenuItem可以说是用得最多的了，它的作用是编辑器上菜单项，一般用于一些快捷操作，比如交换两个物体位置：



由于是涉及编辑器的代码，所以依然可以放在Editor文件夹下面，具体代码如下：

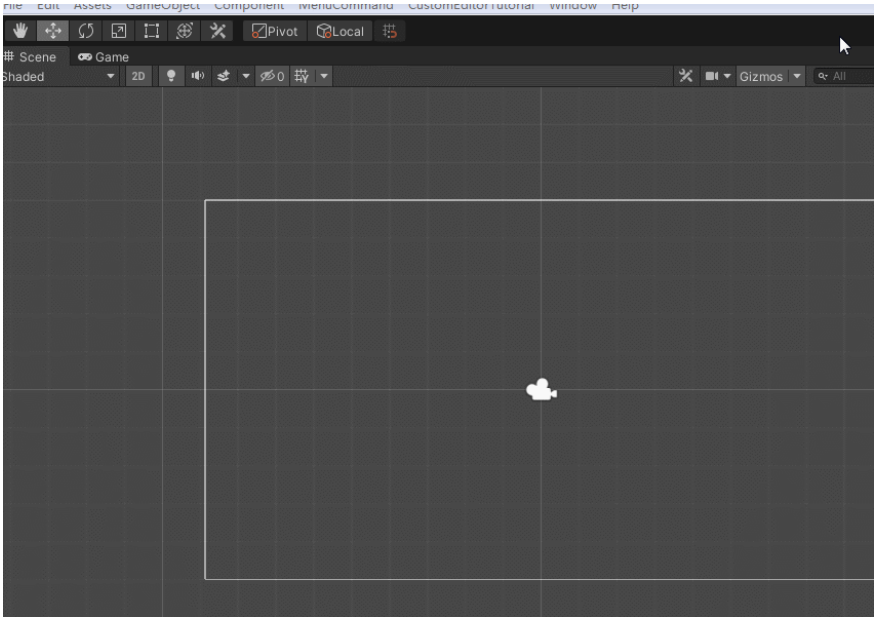
```
using UnityEditor;
using UnityEngine;

public class MenuCommand
{
    [MenuItem("MenuCommand/SwapGameObject")]
    protected static void SwapGameObject()
    {
        //只有两个物体才能交换
        if( Selection.gameObjects.Length == 2 )
        {
            Vector3 tmpPos = Selection.gameObjects[0].transform.
            Selection.gameObjects[0].transform.position = Select
            Selection.gameObjects[1].transform.position = tmpPos
            //处理两个以上的场景物体可以使用MarkSceneDirty
            UnityEditor.SceneManagement.EditorSceneManager.MarkS
        }
    }
}
```

EditorWindow

EditorWindow在Unity引擎中的应用也算是比较多，比如Animation、TileMap和Animitor窗口应该就是用到了EditorWindow。创建方法仍然是在Editor文件夹中创建一个继承自EditorWindow的脚本。EditorWindow有一个GetWindow的方法，调用之后如果当前没有这个窗口会返回新的，如果有就返回当前窗口，之后调用Show即可展示这个窗口。可以使用MenuItem来显示这个EditorWindow，重写OnGUI方法即可以写Editor的UI：

```
using UnityEngine;
using UnityEditor;

namespace EditorTutorial
{
    public class EditorWindowTest : EditorWindow
    {

        [MenuItem("CustomEditorTutorial/WindowTest")]
        private static void ShowWindow()
        {
            var window = GetWindow();
            window.titleContent = new GUIContent("WindowTest");
            window.Show();
        }

        private void OnGUI()
        {
            if(GUILayout.Button("Click Me"))
            {
                //Logic
            }
        }
    }
}
```

之后点击编辑器的Menu就会有这个EditorWindow出来:



EditorWindow的UI的写法跟OnInspectorGUI的写法差不多，基本是**GUILayout**和**EditorGUILayout**这两个类。

EditorWindow与OnInspectorGUI的差别

最主要的差别是EditorWindow可以停靠的在边栏上，不会因为你点击一个物体就重新生成。而OnInspectorGUI的Editor类在你每次切换点击时候都会调用OnEnable方法。

EditorWindow如何绘制Scene界面UI

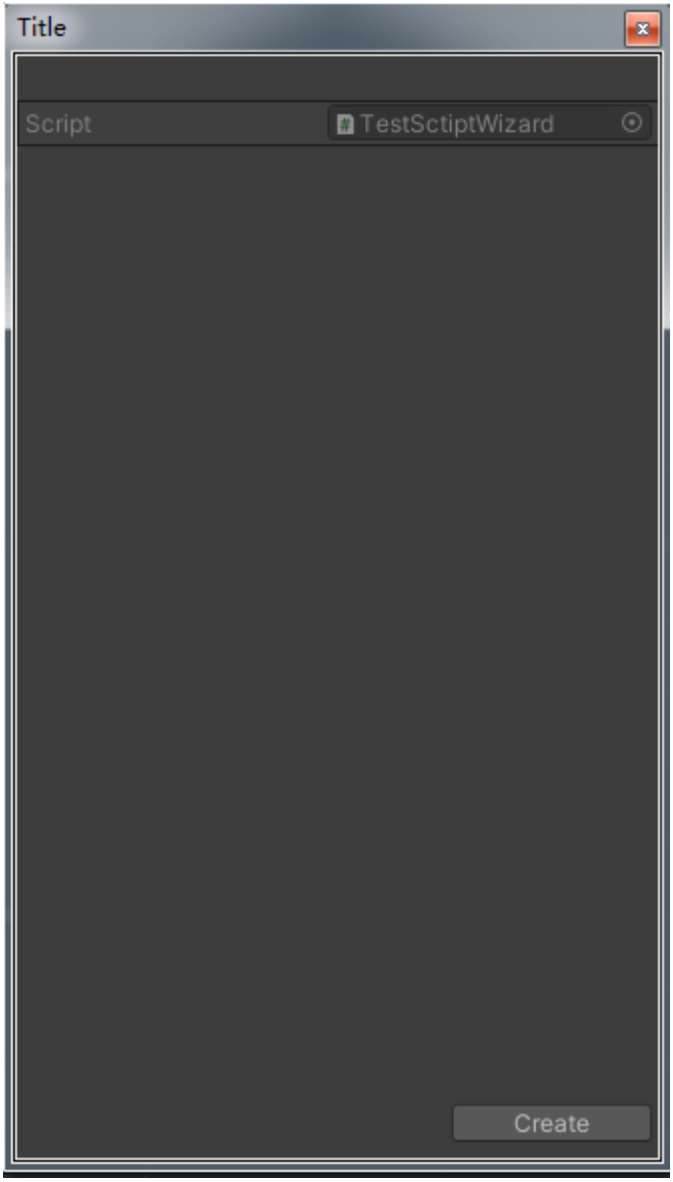
在EditorWindow中如果需要对Scene绘制一些UI，这个时候使用Editor那种OnSceneGUI是无效的，这个时候则需要在Focus或者OnEnable时候加入SceneView的事件回调中,并且在OnDestroy时候去除该回调：

```
private void OnFocus()
{
    //在2019版本是这个回调
    SceneView.duringSceneGui -= OnSceneGUI;
    SceneView.duringSceneGui += OnSceneGUI;

    //以前版本回调
    // SceneView.onSceneGUIDelegate -= OnSceneGUI
    // SceneView.onSceneGUIDelegate += OnSceneGUI
}

private void OnDestroy()
{
    SceneView.duringSceneGui -= OnSceneGUI;
}
private void OnSceneGUI( SceneView view )
{
}
```

ScriptWizard

Unity引擎的中的BuildSetting窗口(Ctr+Shift+B弹出的窗口)就是使用了SciptWizard，一般来开发过程中作为比较简单的生成器和初始化类型的功能来使用，比如美术给我一个序列帧，我需要直接生成一个带SpriteRenderer的GameObject，而且它还有自带序列帧的Animator。

默认的显示样式:



其创建过程还是在Editor文件夹下创建一个继承自ScriptWizard的脚本，调用ScriptWizard.DisplayWizard方法即可生成并显示这个窗口，点击右下角的Create会调用OnWizardCreate方法：

```
public class TestSctiptWizard: ScriptableWizard
{

    [MenuItem("CustomEditorTutorial/TestSctiptWizard")]
    private static void MenuEntryCall()
    {
        DisplayWizard("Title");
    }

    private void OnWizardCreate()
    {

    }
}
```

ScriptWizard与EditorWindow的区别

在ScriptWizard中如果你声明一个Public的变量，会发现在窗口可以直接显示，但是在EditorWindow则是不能显示。

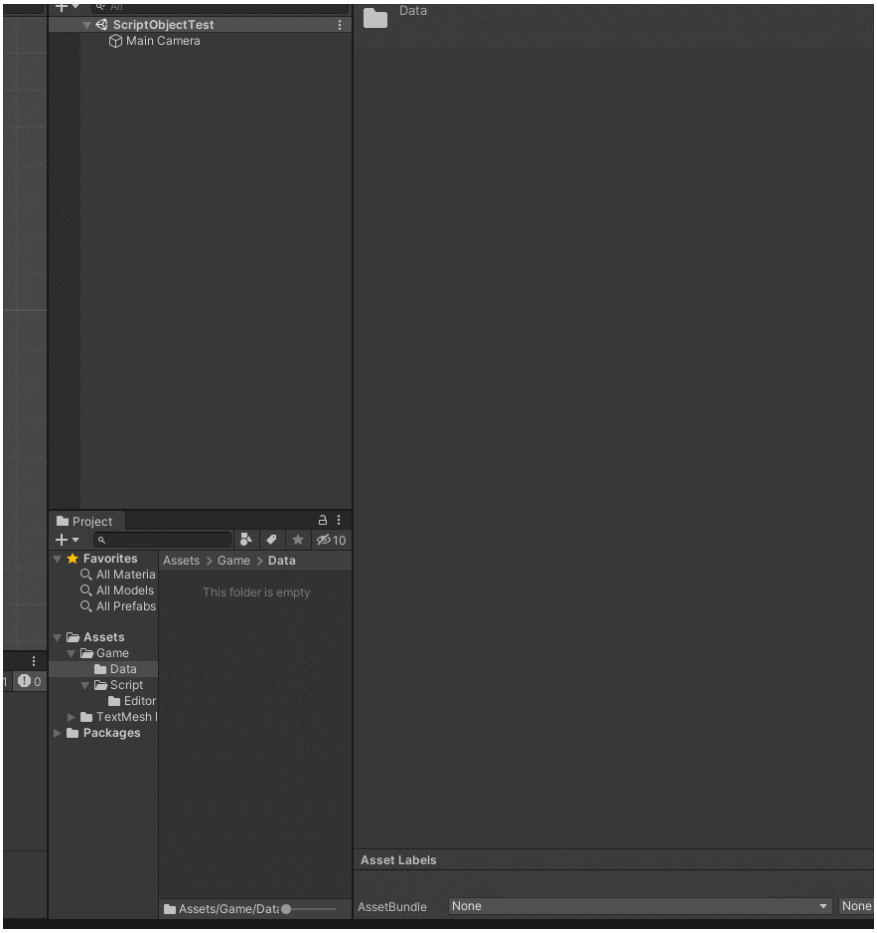
ScriptObject

对于游戏中一些数据和配置可以考虑用ScriptObject来保存，虽然XML之流也可以，但是ScriptObject相对比较简单而且可以保存UnityObject比如Sprite、Material这些。甚至你会发现上面说的几个类都是继承自SctriptObject。 因为其不再是只适用编辑器，所以不必放在Editor文件夹下 。

与SctiptWizard类似，也是声明Public可以在窗口上直接看到，自定义绘制GUI也是在OnGUI方法里面：

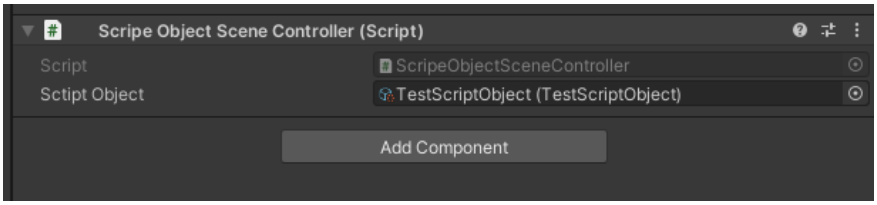
```
[CreateAssetMenu(fileName = "TestScriptObject", menuName = "Cust
public class TestScriptObject : ScriptableObject
{
    public string Name;
}
```

使用CreateAssetMenu的Attribute的作用是使得其可以在Project窗口中右键生成：



ScriptObject与System.Serializable的差别

初学者可能会对这两个比较困扰（我一开始就比较困扰），一开始我把SctiptObject拖拽到Monobehaviour上面发现其不会显示出SctiptObject的属性



然后我在SctiptObject上面加上[System.Serializable]，也是没用：

```
[CreateAssetMenu(fileName = "TestScriptObject", menuName = "Cust
[System.Serializable]
public class TestScriptObject : ScriptableObject
{
    public string Name;
}
```

所以是在ScriptObject上面使用[System.Serializable]是不可取的，[System.Serializable]适合于普普通通的Class，比如：

```
[System.Serializable]
public class Data
{
    string Name;
}
```

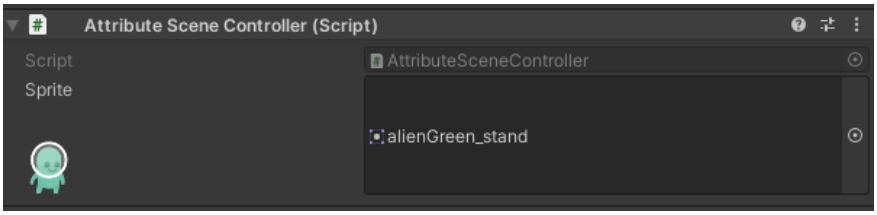
ScriptObject上面调用编辑器修改需要调用EditorUtility.SetDirty，不可调用EditorSceneManager.MarkSceneDirty

因为MarkSceneDirty顾名思义是标记场景为已修改，但是编辑ScriptObject并不属于场景内数据，所以如果修改只可调用EditorUtility.SetDirty，不然会造成数据改动未生效。

Attributes

Attributes是C#的一个功能，它可以让声明信息与代码相关联，其与C#的反射联系很紧密。在Unity中诸如[System.Serializable],[Header],[Range]都是其的应用。一般来说他它功能也可以通过Editor来实现，但是可以绘制对应的属性来说会更好复用。

拓展Attribute相对来说稍微复杂一点，它涉及两个类：**PropertyAttribute**和**PropertyDrawer**，前者是定义它行为，后者主要是其在编辑器的显示效果。一般来说Attribute是放在Runtime，而Drawer则是放在Editor文件夹下。这里的例子是加入[Preview]的Attribute，使得我们拖拽Sprite或者GameObject可以显示预览图：



使用时候的代码如下：

```
public class AttributeSceneController : MonoBehaviour
{
    [Preview]
    public Sprite sprite;
}
```

我们现在Runtime层的文件夹加入继承自PropertyAttribute的PreviewAttribute脚本：

```
public class Preview : PropertyAttribute
{
    public Preview()
    {

    }
}
```

然后在Editor文件夹下加入继承自PropertyDrawer的PreviewDrawer脚本：

```
using UnityEngine;
using UnityEditor;
namespace EditorTutorial
{
    [CustomPropertyDrawer(typeof(Preview))]
    public class PreviewDrawer: PropertyDrawer
    {
        //调整整体高度
        public override float GetPropertyHeight( SerializedPrope
        {
            return base.GetPropertyHeight( property, label ) + 6
        }
        public override void OnGUI(Rect position, SerializedProp
        {
            EditorGUI.BeginProperty(position, label, property);
            EditorGUI.PropertyField(position, property, label);

            // Preview
            Texture2D previewTexture = GetAssetPreview(property)
            if( previewTexture != null )
            {
                Rect previewRect = new Rect()
                {
                    x = position.x + GetIndentLength( position )
                    y = position.y + EditorGUIUtility.singleLine
                    width = position.width,
                    height = 64
                };
                GUI.Label( previewRect, previewTexture );
            }
            EditorGUI.EndProperty();
        }

        public static float GetIndentLength(Rect sourceRect)
        {
            Rect indentRect = EditorGUI.IndentedRect(sourceRect)
            float indentLength = indentRect.x - sourceRect.x;

            return indentLength;
        }

        Texture2D GetAssetPreview( SerializedProperty property )
        {
            if (property.propertyType == SerializedPropertyType.
            {
                if (property.objectReferenceValue != null)
                {
                Texture2D previewTexture = AssetPreview.GetAsset
```

```
                return previewTexture;
                }
            return null;
            }
            return null;
        }
    }
}
```

这里是对属性的一些绘制，实际开发过程中我们常常需要一些可交互的UI，比如在方法上面加一个[Button]然后在编辑器暴露出一个按钮出来，具体的例子可以参考NaughtyAttribute

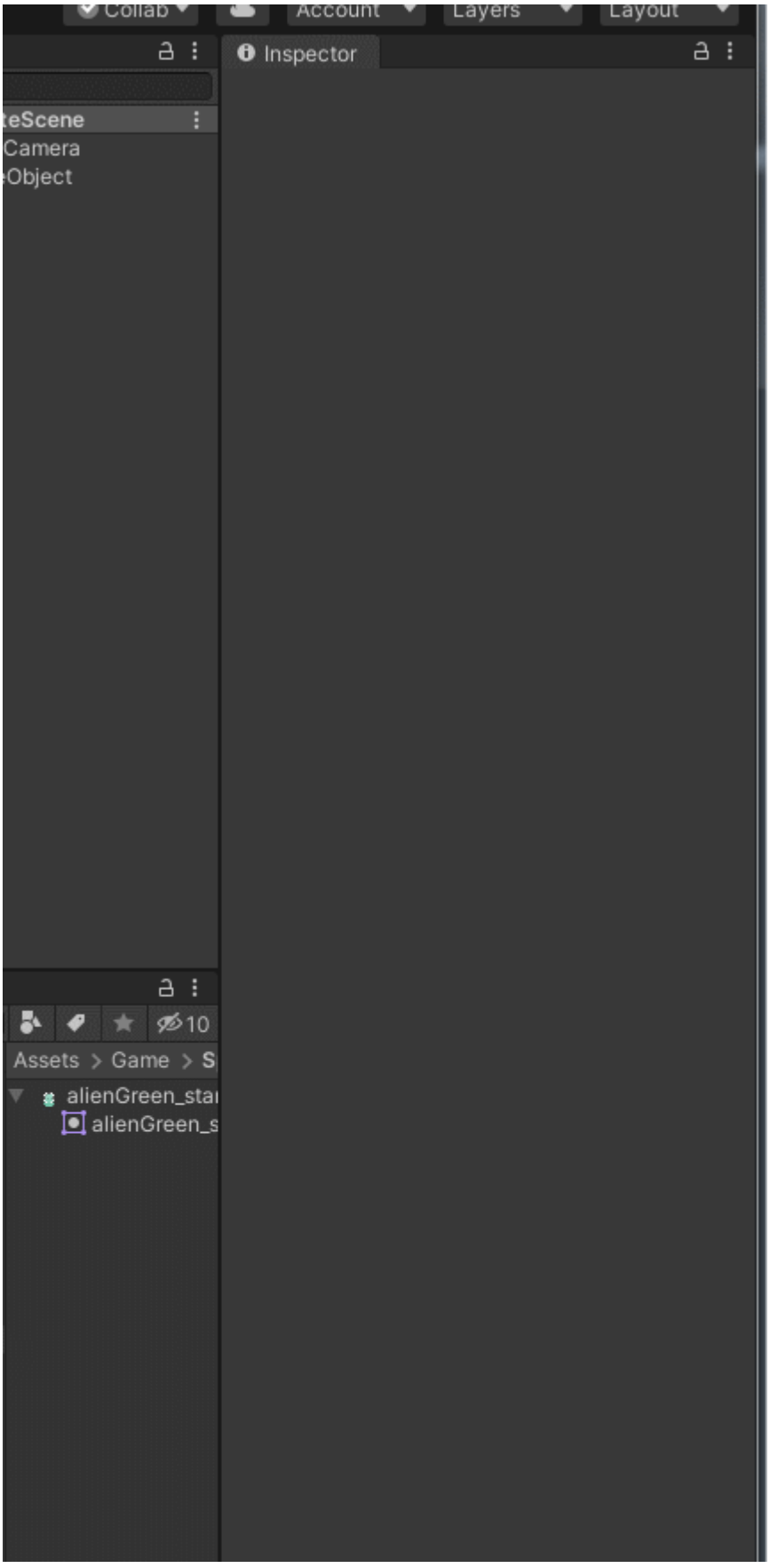
AssetPostprocessor

在开发过程中常常会遇到资源导入问题，比如我制作像素游戏图片要求是FilterMode为Point,图片不需要压缩，PixelsPerUnit为16，如果每次复制到一个图片到项目再修改会很麻烦。这里一个解决方案是可以用MenuItem来处理，但还需要多点几下，而使用AssetPostprocessor则可以自动处理完成。

在Editor文件夹下新建一个继承自AssetPostprocessor的TexturePipeLine：

```
public class TexturePipeLine : AssetPostprocessor
{
    private void OnPreprocessTexture()
    {
        TextureImporter importer = assetImporter as TextureImpor
        if( importer.filterMode == FilterMode.Point ) return;

        importer.spriteImportMode = SpriteImportMode.Single;

        importer.spritePixelsPerUnit = 16;
        importer.filterMode = FilterMode.Point;
        importer.maxTextureSize = 2048;
        importer.textureCompression = TextureImporterCompression

        TextureImporterSettings settings = new TextureImporterSe
        importer.ReadTextureSettings( settings );
        settings.ApplyTextureType( TextureImporterType.Sprite );
        importer.SetTextureSettings( settings ) ;
    }
}
```

之后再导入一个像素图片发现就已经全部设置好了：



可以想象的一些使用场景是可以根据XML和SpriteSheet来实现自动生成动画或者自动切图，解析PSD或ASE自动导入PNG。

其他一些值得注意的地方

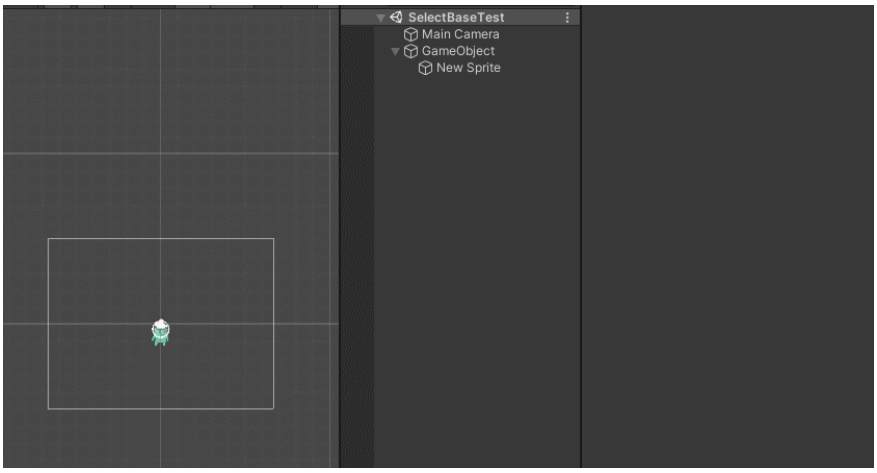
Undo

在之前说过在Editor里面直接改动原来的Monobehaviour脚本是变量是无法撤销的，但是使用 serializedObject来修改则可以撤销。这里可以自己写一个Undo来记录使其可以撤销，代码如下：

```
if(GUILayout.Button("Click Me"))
{
    InspectorTest ctr = target as InspectorTest;
    //记录使其可以撤销
    Undo.RecordObject( ctr ,"Change Name" );
    ctr.Name = "Codinggamer";
    EditorUtility.SetDirty( ctr );
}
```

SelectionBase

当你的类中使用[SelectionBase]的Attribute时候，如果你点击其子节点下的物体，其仍然只会聚焦这个父节点。



不在Editor文件夹里面写编辑器代码

有的时候我们Monobehaviour本身很短小，拓展InspectorGUI的代码也很短小，没有必要在Editor上创建一个新的脚本，可以直接使用 #UNITY_EDITOR 的宏来创建一个拓展编辑器，比如之前的拓展InspectorGUI可以这样写：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

namespace EditorTutorial
{
    public class InspectorTest : MonoBehaviour
    {
        public string Name = "hello";
    }

    #if UNITY_EDITOR
    [CustomEditor(typeof(InspectorTest))]
    public class InspectorTestEditor : Editor
    {
        public override void OnInspectorGUI()
        {
            base.OnInspectorGUI();
            if(GUILayout.Button("Click Me"))
            {

                InspectorTest ctr = target as InspectorTest;
                //记录使其可以撤销
                Undo.RecordObject( ctr ,"Change Name" );
                ctr.Name = "Codinggamer";
                EditorUtility.SetDirty( ctr );
            }
        }
    }
    #endif
}
```

EditorWindow和Editor保存数据

这里需要使用EditorPrefs来保存和读取数据，在需要保存的数据上面加上[System.SerializeField]的Attribute，然后在OnEnable和OnDisable时候可以保存或者度序列化json：

```
[SerializeField]
public string Name = "Hi";

private void OnEnable()
{
    var data = EditorPrefs.GetString( "WINDOW_KEY", JsonUtility.
    JsonUtility.FromJsonOverwrite( data, this );
}

private void OnDisable()
{
    var data = JsonUtility.ToJson( this, false  );
    EditorPrefs.SetString("WINDOW_KEY", data);
}
```

感觉这种方法也可以在运行时序列化脚本保存到本地。

在代码中检索对应MonoBehaviour的Editor类

在EditorWindow的使用过程中，有的时候可能需要调用到对应拓展MonoBehaviour的Editor代码，这个时候可以使用Editor.CreateEditor方法来创建这个Editor。

在编辑器代码中生成 SerializedObject

上面说过，在编辑器代码中一般比较多使用SerializedObject，像Editor类中就内置了serializedObject。实际上所有继承自SctiptObject或者Monobehaviour的脚本都可以生成SerializedObject。其生成方式很简单只需要new的时候传入你需要序列化的组件即可：

```
SerializedObject serialized = new SerializedObject(this);
```

后记

这篇文章中并没有涉及比较多的API的使用，更多的是想展现出**可以用拓展编辑器来做什么以及当你想做一些拓展时候需要从哪里入手**。比如如果你想给美术人员做一个快捷生成角色的工具，就可以使用ScriptWizard。如果你需要让美术人员和设计师更加方便地调整人物属性，则可以考虑使用Editor。如果你需要给关卡设计师制作一个关卡编辑器，那可以考虑使用EditorWindow。

写得比较多，以上这个就是我在使用Unity拓展编辑器的总结与遇到的一些问题的经验。

示例Github仓库： EditorTutorial

推荐资源

Extending Unity with Editor Scripting

unity editor extension manual(日文，看样章不错，可只看代码)

(Youtube) Sebastian Lague的Extending the Editor系列

NaughtyAttribute