Unity 案例

一. 2D 角色移动

二. 3D 角色移动

三. 3D 相机控制

四. 输入管理-InputManager

构建输入管理，主要使用事件来传递玩家输入的信息。

```
public class InputManager : SingleComponent<InputManager>
{
    #region 水平输入事件完整声明格式
    /*
    * 事件的拥有者(Event Source) -----> InputManager 类
    * 事件(Event) -----> OnSelectSkill
    * 事件的响应者(Event Subscriber) -----> EquipmentSlotParent
    * 事件处理器(Event Handler) -----> SetSelectSkill
    * 事件的订阅关系(+= Operator) -----> +=
    */
    public delegate void InputHorizontalEventHandler(float _horizontalValue);//为了事件而声明的委托类型最好加上 EventHandler 后缀,这个委托可以指向一个无返回值的方法，
    private InputHorizontalEventHandler inputHorizontalEventHandler;//完整声明格式中最重要的就是这个委托类型的字段，用来存储事件处理器
    public event InputHorizontalEventHandler OnInputHorizontal
    {
        add
        {
            inputHorizontalEventHandler += value;//我们虽然不知道未来传进来的事件处理器是什么，我们可以用上下文关键字 Value 来表示
        }
        remove
        {
            inputHorizontalEventHandler -= value;//移除事件处理器
        }
    }
    #endregion
    //action 委托必定没有返回值,func 委托必定具有一个返回值
    public event Action<float> OnInputVertical;//垂直输入事件的简略声明

    protected override void Awake()
    {
        base.Awake();
```

```
    }

    private void Update()
    {
        #region 水平方向输入
        if (inputHorizontalEventHandler != null)
        {
            inputHorizontalEventHandler(Input.GetAxisRaw("Horizontal"));//只能由事件拥有者在其内部调用内部逻辑调用
        }
        #endregion
        #region 垂直方向输入
        if (OnInputVertical != null)
        {
            OnInputVertical(Input.GetAxisRaw("Vertical"));
        }
        #endregion
    }
}
```

五. 场景异步加载与过渡

六. 自定义 URP 后处理

七. 通用对象池

1. 概述

对象池模式(The Object Pool Pattern)是单例模式的一个变种，它提供了获取一系列相同对象实例的入口。当你需要对象来代表一组可替代资源的时候就变的很有用，每个对象每次可以被一个组件使用。

2. 理解

对象池模式管理一个可代替对象的集合。组件从池中借出对象，用它来完成一些任务并当任务完成时归还该对象。被归还的对象接着满足请求，不管是同一个组件还是其他组件的请求。对象池模式可以管理那些代表的现实资源或者通过重用来分摊昂贵初始化代价的对象。

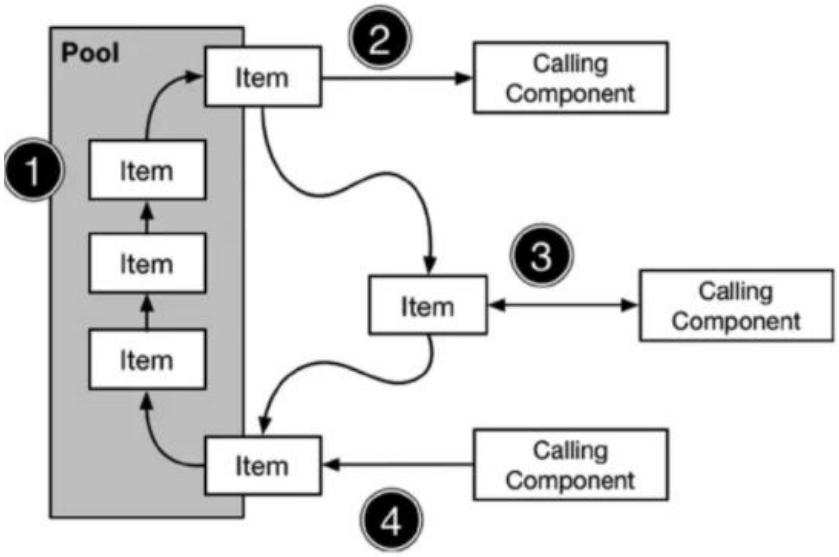

图 1.　　对象池

第一步操作是构建对象池。

第二步操作是借出对象。

第三步操作是组件用借出的对象来完成一些任务。这时候并不需要对象池再做什么，但是这也意味着该对象将被租借一段时间并且不能在被其他组件借出。

第四步操作就是归还，组件归还借出的对象这样可以继续满足其他的租借请求。

3. 优点

复用池中对象，没有分配内存和创建堆中对象的开销， 没有释放内存和销毁堆中对象的开销， 进而减少垃圾收集器的负担， 避免内存抖动；不必重复初始化对象状态， 对于比较耗时的 constructor 和 finalize 来说非常合适。

4. 缺点

① 并发环境中， 多个线程可能同时需要获取池中对象， 进而需要在堆数据结构上进行同步或者因为锁竞争而产生阻塞，这种开销要比创建销毁对象的开销高数百倍。

② 由于池中对象的数量有限， 势必成为一个可伸缩性瓶颈。

③ 很难正确的设定对象池的大小， 如果太小则起不到作用， 如果过大， 则占用内存资源高， 可以起一个线程定期扫描分析， 将池压缩到一个合适的尺寸以节约内存，但为了获得不错的分析结果， 在扫描期间可能需要暂停复用以避免干扰(造成效率低下)， 或者使用非常复杂的算法策略(增加维护难度)。

④ 设计和使用对象池容易出错， 设计上需要注意状态同步， 这是个难点， 使用上可能存在忘记归还(就像 c 语言编程忘记 free 一样)， 重复归还(可能需要做个循环判断一下是否池中存在此对象， 这也是个开销)， 归还后仍旧使用对象(可能造成多个线程并发使用一个对象的情况)等问题。

5. 简单对象池

```
using UnityEngine;
using System.Collections.Generic;
using System;

namespace UniversalPool
{
```

```
[DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
[AddComponentMenu("GameObject/PoolObject")]
public class PoolObject : MonoBehaviour
{
    public string poolName;//对象池名称
    public bool isPooled; //定义对象是在池中等待还是在使用中
}

public enum PoolInflationType//对象池膨胀类型
{
    INCREASE,// 当对象池膨胀时，将一个对象添加到池中。
    DOUBLE// 当对象池膨胀时，将池的大小增加一倍
}

class Pool
{
    private Stack<PoolObject> availableObjStack = new Stack<PoolObject>();//Stack
栈是一个先进后出（LIFO）表
    private Queue<PoolObject> availableObjQueue = new
Queue<PoolObject>();//Queue 队列是先进先出(FIFO)

    //未使用对象的根节点
    private GameObject rootObj;
    private PoolInflationType inflationType;
    private string poolName;
    private int objectsInUse = 0;

    //构造函数
    public Pool(string poolName, GameObject poolObjectPrefab, GameObject
rootPoolObj, int initialCount, PoolInflationType type,bool useStack = true)
    {
        if (poolObjectPrefab == null)
        {
#if UNITY_EDITOR
            Debug.LogError("[ObjPoolManager] 对象池预制体为空 !");
#endif
            return;
        }
        this.poolName = poolName;
        this.inflationType = type;
        this.rootObj = new GameObject(poolName + "Pool");//为对象池命名
        this.rootObj.transform.SetParent(rootPoolObj.transform,
false);//worldPositionStays 如果为 true，则修改父相对位置、缩放和旋转，以使对象保持与
以前相同的世界空间位置、旋转和缩放。
```

```
        GameObject go = GameObject.Instantiate(poolObjectPrefab);
        PoolObject po = go.GetComponent<PoolObject>();
        if (po == null)
        {
            po = go.AddComponent<PoolObject>();//为对象添加 PoolObject 组件
        }
        po.poolName = poolName;//设置对象所属对象池名称
        AddObjectToPool(po, useStack);//确保对象池中至少存在一个对象

        //填充对象池
        FillPool(Mathf.Max(initialCount, 1), useStack);
    }

    //添加到栈对象池,o(1)
    private void AddObjectToPool(PoolObject po,bool useStack = true)
    {
        po.gameObject.SetActive(false);
        po.gameObject.name = poolName;
        if (useStack)
        {
            availableObjStack.Push(po);//入栈
        }
        else
        {
            availableObjQueue.Enqueue(po);//入队列,将对象添加到。
        }
        po.isPooled = true;//对象是在对象池中
        //添加到根节点
        po.gameObject.transform.SetParent(rootObj.transform, false);
    }

    /// <summary>
    ///  填充对象池到指定数量
    /// </summary>
    /// <param name="initialCount">填充的数量</param>
    private void FillPool(int initialCount, bool useStack = true)
    {
        for (int index = 0; index < initialCount; index++)
        {
            PoolObject po = null;
            if (useStack)
            {
                po = GameObject.Instantiate(availableObjStack.Peek());//Peek()返
```

```
回位于栈顶的对象但不将其移除
                        }
                        else
                        {
                            po = GameObject.Instantiate(availableObjQueue.Peek());//Peek()返
回位于队首的对象但不将其移除
                        }
                        AddObjectToPool(po, useStack);
                    }
                }

                //栈中的下一个可用对象，O(1)
                public GameObject NextAvailableObject(bool autoActive,bool useStack = true)
                {
                    PoolObject po = null;
                    if (useStack ? availableObjStack.Count > 1 : availableObjQueue.Count > 1)//
判断当前是否有足够可使用对象
                    {
                        //有就出栈，Pop()删除并返回 Stack 顶部的对象，Dequeue 删除并返
回队首的对象
                        po = useStack ? availableObjStack.Pop() :
availableObjQueue.Dequeue();
                    }
                    else
                    {
                        int increaseSize = 0;
                        switch (inflationType)
                        {
                            case PoolInflationType.INCREASE:
                                increaseSize = 1;
                                break;
                            case PoolInflationType.DOUBLE:
                                {
                                    increaseSize = useStack ? availableObjStack.Count :
availableObjQueue.Count + Mathf.Max(objectsInUse, 0);
                                    break;
                                }
                        }
    #if UNITY_EDITOR
                        Debug.Log(string.Format("Growing pool {0}: {1} populated",
poolName, increaseSize));
    #endif
                        if (increaseSize > 0)
                        {
```

```
                    FillPool(increaseSize, useStack);//填充对象池
                    if (useStack)
                    {
                        po = availableObjStack.Pop();//有就出栈，Pop()删除并返回Stack 顶部的对象
                    }
                    else
                    {
                        po = availableObjQueue.Dequeue();//有就出队列，Dequeue 删除并返回队首的对象
                    }
                }
            }

            GameObject result = null;
            if (po != null)
            {
                objectsInUse++;//已使用对象数增加
                po.isPooled = false;
                result = po.gameObject;
                if (autoActive)
                {
                    result.SetActive(true);
                }
            }

            return result;
        }

        //将对象返回到对象池中，O(1)
        public void ReturnObjectToPool(PoolObject po,bool useStack = true)
        {
            if (poolName.Equals(po.poolName))//Equals(String)确定此实例是否与另一个指定的 String 对象具有相同的值
            {
                objectsInUse--;//已使用对象减少
                /* 我们本可以使用 Stack.Contains(Object)来检查该对象是否在对象池中。
                 * 但是会使此方法的时间复杂度变为 O（n）*/
                if (po.isPooled)
                {
#if UNITY_EDITOR
                    Debug.LogWarning(po.gameObject.name + " 已经在对象池里了。您为什么要再次返回？ 请检查用法。");
```

```
    #endif
                        }
                        else
                        {
                            AddObjectToPool(po, useStack);
                        }
                    }
                    else
                    {
                        Debug.LogError(string.Format("试图将对象添加到不正确的对象池中
{0} {1}", po.poolName, poolName));
                    }
                }
            }
        }
```

6. 对象池管理

```
    using UnityEngine;
    using System.Collections.Generic;

    namespace UniversalPool
    {
        [DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
        [AddComponentMenu("GameObject/PoolResourceManager")]
        public class PoolResourceManager : MonoBehaviour
        {
            //使用数据字典为对象池添加关键字
            private Dictionary<string, Pool> poolDict = new Dictionary<string, Pool>();

            private static PoolResourceManager instance = null;

            public static PoolResourceManager Instance
            {
                get
                {
                    if (instance == null)//不存在实例就创建一个
                    {
                        GameObject go = new GameObject("ResourceManager",
typeof(PoolResourceManager));
                        go.transform.localPosition = new Vector3(0, 0, 0);
                        instance = go.GetComponent<PoolResourceManager>();

                        if (Application.isPlaying)
                        {
                            DontDestroyOnLoad(instance.gameObject);
```

```
                }
                else
                {
                    Debug.LogWarning("[ResourceManager] 您最好在编辑器模式下忽略 ResourceManager");
                }
            }

            return instance;
        }
    }
    #region 初始化对象池
    public void InitPool(string poolName, int size, PoolInflationType type = PoolInflationType.DOUBLE,bool useStack = true)
    {
        if (poolDict.ContainsKey(poolName))//数据字典中是否已有关键字
        {
            return;
        }
        else
        {
            GameObject Prefab = Resources.Load<GameObject>(poolName);//在Resources 文件夹中获取
            if (Prefab == null)
            {
                Debug.LogError("[ResourceManager] Invalide prefab name for pooling :" + poolName);
                return;
            }
            poolDict[poolName] = new Pool(poolName, Prefab, gameObject, size, type, useStack);
        }
    }

    /// <summary>
    /// 初始化对象池
    /// </summary>
    /// <param name="cell">需要初始化的对象</param>
    /// <param name="size">对象数量</param>
    /// <param name="type">膨胀类型</param>
    public void InitPool(GameObject cell, int size, PoolInflationType type = PoolInflationType.DOUBLE, bool useStack = true)
    {
        if (poolDict.ContainsKey(cell.name))
```

```
                {
                    return;
                }
                else
                {
                    poolDict[cell.name] = new Pool(cell.name, cell, gameObject, size, type,
useStack);
                }
            }
            #endregion

            /// <summary>
            ///  获取池中的可用对象,如果池中没有可用的对象则为 null。
            /// </summary>
            /// <param name="poolName">对象池名称</param>
            /// <param name="autoActive">是否自动激活</param>
            /// <param name="autoCreate">自动创建对象数量</param>
            public GameObject GetObjectFromPool(string poolName, bool autoActive = true,
int autoCreate = 0, bool useStack = true)
            {
                GameObject result = null;

                if (!poolDict.ContainsKey(poolName) && autoCreate > 0)
                {
                    InitPool(poolName, autoCreate, PoolInflationType.INCREASE);
                }

                if (poolDict.ContainsKey(poolName))
                {
                    Pool pool = poolDict[poolName];
                    result = pool.NextAvailableObject(autoActive, useStack);
    #if UNITY_EDITOR
                    if (result == null)//对象池中找不到可用对象
                    {
                        Debug.LogWarning("[ResourceManager]:No object available in " +
poolName);
                    }
    #endif
                }
    #if UNITY_EDITOR
                else
                {
                    Debug.LogError("[ResourceManager]:Invalid pool name specified: " +
poolName);
```

```
                }
    #endif
                return result;
            }

            /// <summary>
            ///  将 GameObject 返回对象池
            /// </summary>
            /// <param name="go"></param>
            public void ReturnObjectToPool(GameObject go, bool useStack = true)
            {
                PoolObject po = go.GetComponent<PoolObject>();
                if (po == null)
                {
    #if UNITY_EDITOR
                    Debug.LogWarning("指定的对象不是对象池实例: " + go.name);
    #endif
                }
                else
                {
                    Pool pool = null;
                    //out 关键字是只出不进
                    if (poolDict.TryGetValue(po.poolName, out pool))//TryGetValue 获取与
指定键关联的值
                    {
                        pool.ReturnObjectToPool(po, useStack);
                    }
    #if UNITY_EDITOR
                    else
                    {
                        Debug.LogWarning("没有名称可用的对象池: " + po.poolName);
                    }
    #endif
                }
            }

            /// <summary>
            ///  将对象返回对象池
            /// </summary>
            /// <param name="t"></param>
            public void ReturnTransformToPool(Transform t, bool useStack = true)
            {
                if (t == null)
                {
```

```
    #if UNITY_EDITOR
                    Debug.LogError("[ResourceManager] try to return a null transform to
pool!");
    #endif
                    return;
                }
                ReturnObjectToPool(t.gameObject, useStack);
            }


            //判断该对象是否有对象池
            public bool HasPool(GameObject cell)
            {
                return poolDict.ContainsKey(cell.name);
            }
        }
    }
```

八. Unity 脚部 IK

1. 构建测试场景

首先将所需的模型导入到 Unity 中，然后构建一些用于脚部 IK 的测试物体。

图 2. 构建测试场景

模型需要有 Rigidbody 组件和 Collider 组件，否则，在查看后续的效果时，会非常不方便。

2. 创建 Animator 并设置所需参数

构建完成后就可以创建一个 Animator，我们会用 Animator 去控制动画的播放。将所需的动画拖拽到 Animator 中，我们拖拽进去的第一个动画片段，会被 Animator 设置为默认开始动画。

要修改默认的开始动画，只需要将鼠标移动到 Entry 上，然后点击鼠标右键，找到 Set StateMachine Default State 就可以修改默认的开始动画了。

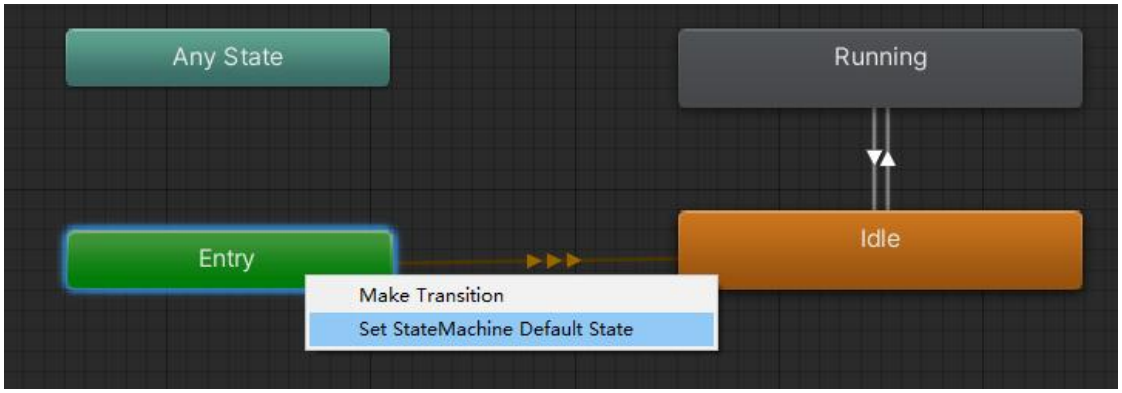

图 3.　　设置状态机的默认状态

要在两个动画片段之间切换，我们则需要在两个动画之间建立一条有向的过渡线。

我们在角色身上添加一个 Animator 组件，并将我们创建的 Animator Controller 赋予它。

过渡线是可以设置条件的，默认情况下是没有任何条件的，一旦设置条件，只有当条件满足时才能从一个动画切换到另一个动画。

为了让动画依据设置的参数去切换，我们需要在 Animator 界面中找到 Paramaters，在里面选择所需的参数类型。

因为这里只是一个简单的测试，就只创建一个 Bool 类型的参数即可。

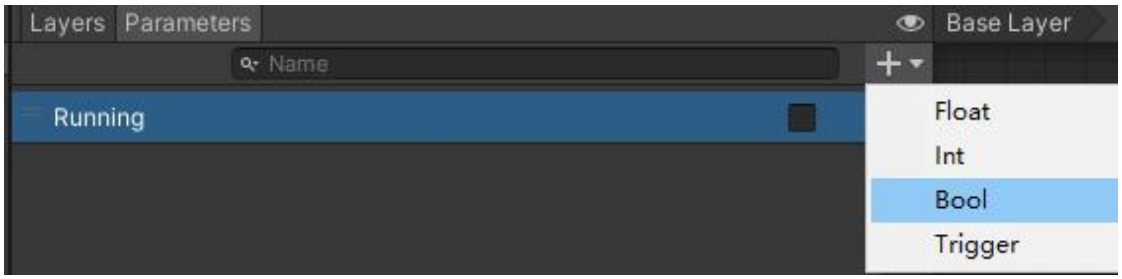

图 4.　　创建动画参数

有了动画参数以后，我们就可以找到过渡线，为其添加动画切换的条件。因为过渡线在默认情况下，是有退出时间的，只有当前动画播放到对应位置时，才能进行动画切换。有了切换条件后，我们就可以关闭它，让其在任何时候都可以进行动画切换，而不需要等待片段播放到特定时间。

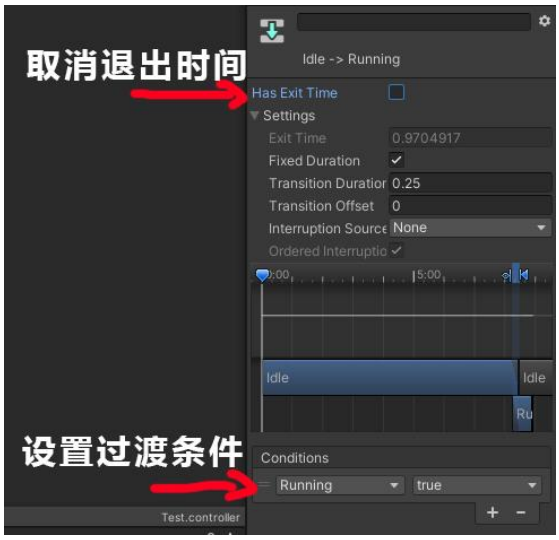

图 5.　　设置过渡条件

3. 创建脚部 IK

1) 设置参数

现在已经可以切换动画了，我们就可以开始创建脚步 IK 的脚本了。

我们在脚本中创建一系列的参数，这些参数主要是获取 IK 的位置、实时脚步位置以及脚所对应的旋转值。要实现 IK，我们需要对角色所处位置进行一个检测，这需要使用到射线检测，为此需要一系列与射线检测有关的参数。

```
private Animator theAnimator;
private Vector3 leftFootIk, rightFootIk;//射线检测需要的 IK 位置
private Vector3 leftFootPosition, rightFootPosition;//IK 位置赋值
private Quaternion leftFootRotation, rightFootRotation;//IK 旋转赋值

[SerializeField] private LayerMask iKLayer;//射线可以检测到的层
[SerializeField] [Range(0, 0.2f)] private float rayHitOffset;//射线检测位置与 IK 位置的偏移
[SerializeField] private float rayCastDistance;//射线检测距离

private RaycastHit hitInfo;
[SerializeField] private bool enableIK = true;//是否启用 IK
[SerializeField] private float iKSphereRadius = 0.05f;
[SerializeField] private float positionSphereRadius = 0.05f;
```

2) 获取组件以及为参数赋值

在Awake()函数中获取组件，并为部分参数赋初值。

```
theAnimator = this.gameObject.GetComponent<Animator>();
leftFootIk = theAnimator.GetIKPosition(AvatarIKGoal.LeftFoot);//获取 IK 位置
rightFootIk = theAnimator.GetIKPosition(AvatarIKGoal.RightFoot);
```

3) 设置脚部位置和旋转

要设置脚部的 IK 位置和旋转值，我们需要将方法写在 OnAnimatorIK 函数中。OnAnimatorIK 函数，会在 Animator 组件更新其内部 IK 系统之前调用。权重就代表 IK 对于动画的影响程度，值为 1 时 IK 将完全掌控动画的相关部分，值为 0 时就是原动画展示出的效果。

```
private void OnAnimatorIK(int layerIndex)
    {
        leftFootIk = theAnimator.GetIKPosition(AvatarIKGoal.LeftFoot);
        rightFootIk = theAnimator.GetIKPosition(AvatarIKGoal.RightFoot);

        if (!enableIK)
        {
            return;
        }

        #region 设置 IK 权重
        theAnimator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, 1);
        theAnimator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, 1);
```

```
            theAnimator.SetIKPositionWeight(AvatarIKGoal.RightFoot, 1);
            theAnimator.SetIKRotationWeight(AvatarIKGoal.RightFoot, 1);
            #endregion
            #region  设置 IK 位置和旋转值
            theAnimator.SetIKPosition(AvatarIKGoal.LeftFoot, leftFootPosition);
            theAnimator.SetIKRotation(AvatarIKGoal.LeftFoot, leftFootRotation);

            theAnimator.SetIKPosition(AvatarIKGoal.RightFoot, rightFootPosition);
            theAnimator.SetIKRotation(AvatarIKGoal.RightFoot, rightFootRotation);
            #endregion
        }
```

在这里我们先将所有的权重设置为 1，后面会再对其进行更改。

4) 获得位置和旋转值

在上面的 OnAnimatorIK 函数中，我们已经设置了动画的 IK 位置和旋转值。目前，我们的这些参数值都还是默认值，我们需要在游戏运行过程中去获取。

我们知道，角色脚的位置和角度，与其所站的平台有关。在平地上，角色的脚就是和地面平行的；在坡面上，脚就应该和坡面平行。要获得这些数据，我们就需要使用射线检测去实现了，因为射线检测与 Unity 的物理系统相关，因此这部分内容需要写在 FixedUpdate 函数中。

为了方便查看，使用 Debug.DrawLine 去绘制线段，这也可以写在 OnDrawGizmos 函数中，只不过需要进行一些微小的改动。

```
Debug.DrawLine(leftFootIk + (Vector3.up * 0.5f), leftFootIk + Vector3.down * rayCastDistance, Color.blue, Time.deltaTime);
Debug.DrawLine(rightFootIk + (Vector3.up * 0.5f), rightFootIk + Vector3.down * rayCastDistance, Color.blue, Time.deltaTime);
```

当射线检测到指定层物体时，我们就返回碰撞点的各种信息，脚步的旋转值就与返回的法线信息有关，即下面的 hit.normal。

```
            if (Physics.Raycast(leftFootIk + (Vector3.up * 0.5f), Vector3.down, out RaycastHit hit, rayCastDistance + 1, iKLayer))
            {
                Debug.DrawRay(hit.point, hit.normal, Color.red, Time.deltaTime);

                leftFootPosition = hit.point + Vector3.up * rayHitOffset;// 如果让脚的位置等于碰撞点，那么可能会出现穿模的情况。
                leftFootRotation = Quaternion.FromToRotation(Vector3.up, hit.normal) * transform.rotation;
            }

            if (Physics.Raycast(rightFootIk + (Vector3.up * 0.5f), Vector3.down, out RaycastHit hit_01, rayCastDistance + 1, iKLayer))
            {
                Debug.DrawRay(hit_01.point, hit_01.normal, Color.red, Time.deltaTime);
```

```
                rightFootPosition = hit_01.point + Vector3.up * rayHitOffset;
                rightFootRotation = Quaternion.FromToRotation(Vector3.up, hit_01.normal) *
transform.rotation;
            }
```

上面的代码中，我们将脚的位置进行了一定的抬升，这是因为，如果让脚的位置等于碰撞点，那么可能会出现穿模的情况。

5) 绘制 IK 位置和脚的位置

为了方便大家理解，我们在 OnDrawGizmos 函数中将 IK 位置和脚的位置绘制出来，IK 位置用绿色表示，脚的位置用青色表示。

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(leftFootIk, iKSphereRadius);
        Gizmos.DrawSphere(rightFootIk, iKSphereRadius);
        Gizmos.color = Color.cyan;
        Gizmos.DrawSphere(leftFootPosition, positionSphereRadius);
        Gizmos.DrawSphere(rightFootPosition, positionSphereRadius);
    }
```

当我们完成上述内容后，我们还需要回到 Unity 中，为场景中的测试对象设置一个 Layer。

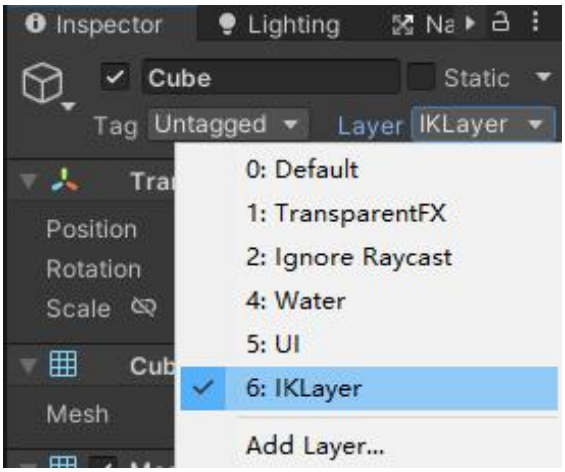

图 6.　　设置对象的 Layer

当射线触碰到对象时，就会判断是否和设定的检测层一致，如果一致就返回碰撞点的各种信息。

4. 设置动画 IK 权重

当我们完成上面的部分，就可以将脚本挂载到角色身上，以测试其实际效果。因为现在的权重是 1，当我们执行待机动画时，几乎没有问题，毕竟此时角色的脚是一直处于地面的。

一旦我们将动画切换为跑动，在原本的动画中，角色的脚是需要抬起来的。因为我们设置的 IK 权重为 1，导致脚的动画将被 IK 完全影响。

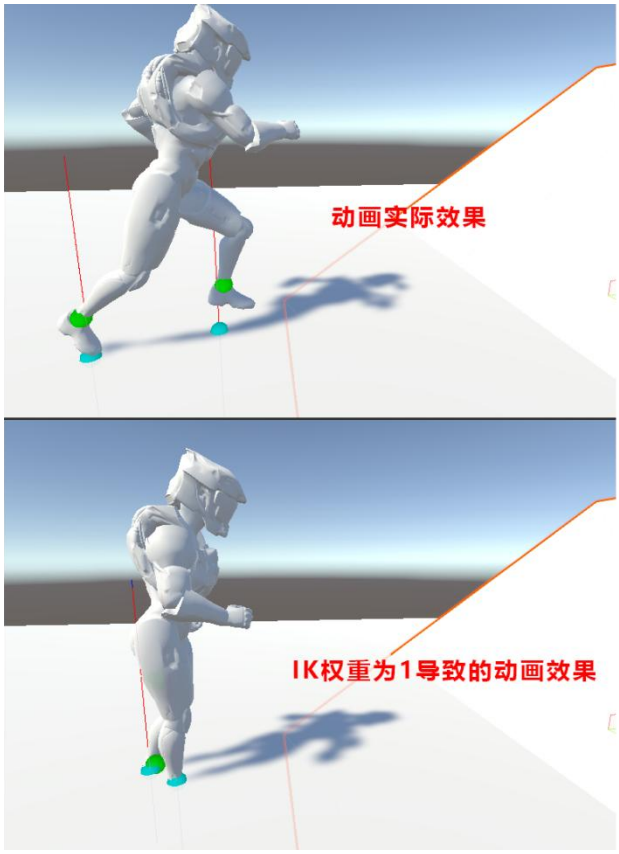

图 7.　　权重对于动画的影响

为了能够解决这种不良的影响，我们可以创建一条动画曲线，为其添加一系列关键帧，用这些关键帧去控制 IK 的值。

在 Animator 中我们可以创建一个，和动画曲线名称一致的参数，它会在动画播放时自动获取对应曲线的值。然后，在代码中，我们可以获取这个参数的值，来实时修改权重。

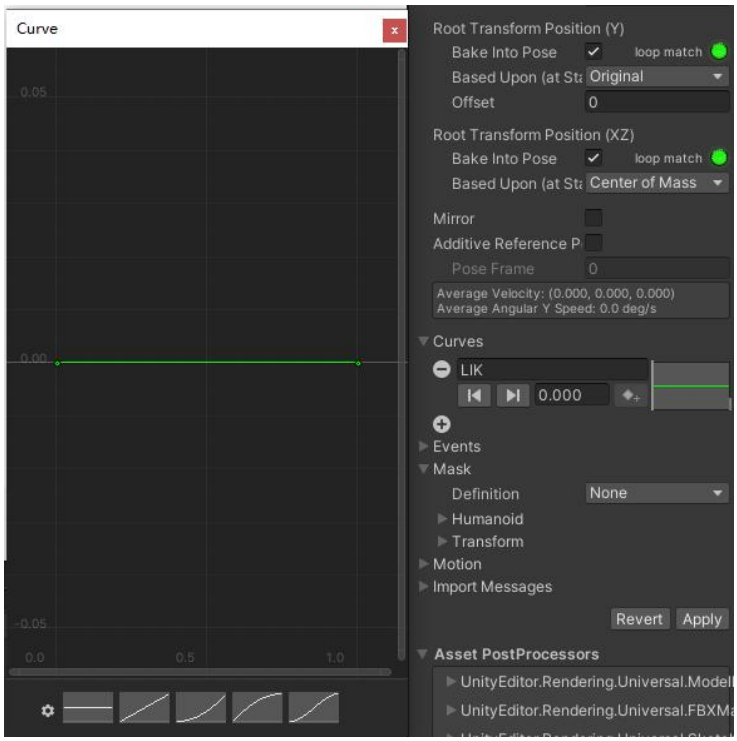

图 8.　　设置动画曲线

当需要脚完全和地面贴合时，其 IK 的权重就应该为 1，当脚需要离开地面时，其 IK 的权重就应该为 0。

```
theAnimator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, theAnimator.GetFloat("LIK"));
theAnimator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, theAnimator.GetFloat("LIK"));

theAnimator.SetIKPositionWeight(AvatarIKGoal.RightFoot, theAnimator.GetFloat("RIK"));
theAnimator.SetIKRotationWeight(AvatarIKGoal.RightFoot, theAnimator.GetFloat("RIK"));
```

修改完成后再次运行，现在就不会再出现 IK 对动画影响过大的情况了。

5. 可能出现的问题

没有 IK 效果出现，请检查是否勾选了对应 Layer 的 IK Pass 选项。只有勾选了 IK Pass 的 laye 才会调用 OnAnimatorIK 方法，每个勾选了 IK Pass 的 layer 调用一次。

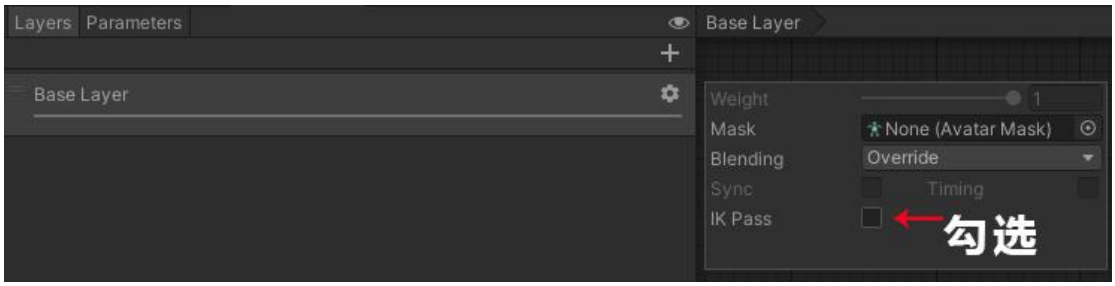

图 9.　　检查 IK Pass 是否勾选

九. 射线检测应用

1. 添加 Layer 层

在使用射线检测之前，我们需要先为被检测的游戏对象设置一个 Layer。Unity 本身为我们预先设置了一系列的 Layer，但实际开发过程中我们是一定会添加许多自己的 Layer 的。

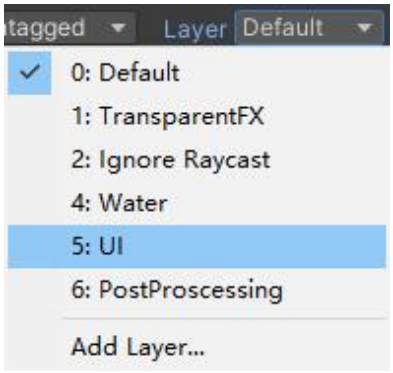

图 10.　　Unity 默认的 Layer

首先我们需要明白的一点是，Layer 最多只能有 32 层，且开始的下标是 0，结束的下标是 31。这是为什么呢？

答案其实很简单，Unity 中是用 int32 来表示 32 个 Layer 层。int32 表示的二进制数一共只有 32 位。因此，我们所设定的 Layer 层，最多也就只能是 32 层。

我们可以为其新增三个 Layer 层用作后面的测试。我这里将它们命名为，Enemy、Friend、Stranger。

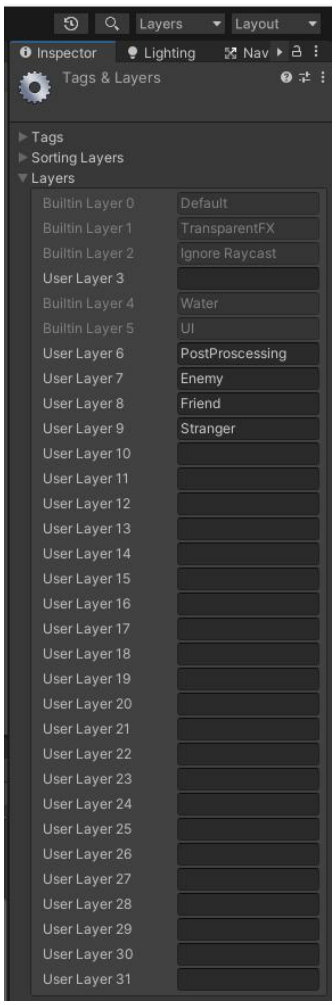

图 11.　新增 Layer 层

2. Layer 和 LayerMask

现在我们已经新增了一些 Layer，我们就可以创建一个 C#脚本，并将其挂载到对象上去进行测试。

首先，我们直接在脚本中打印一下脚本对象的 Layer，查看一下它在代码中实际是何值。

PS：

```
private void Start()
    {
        Debug.Log(this.gameObject.layer);
    }
```

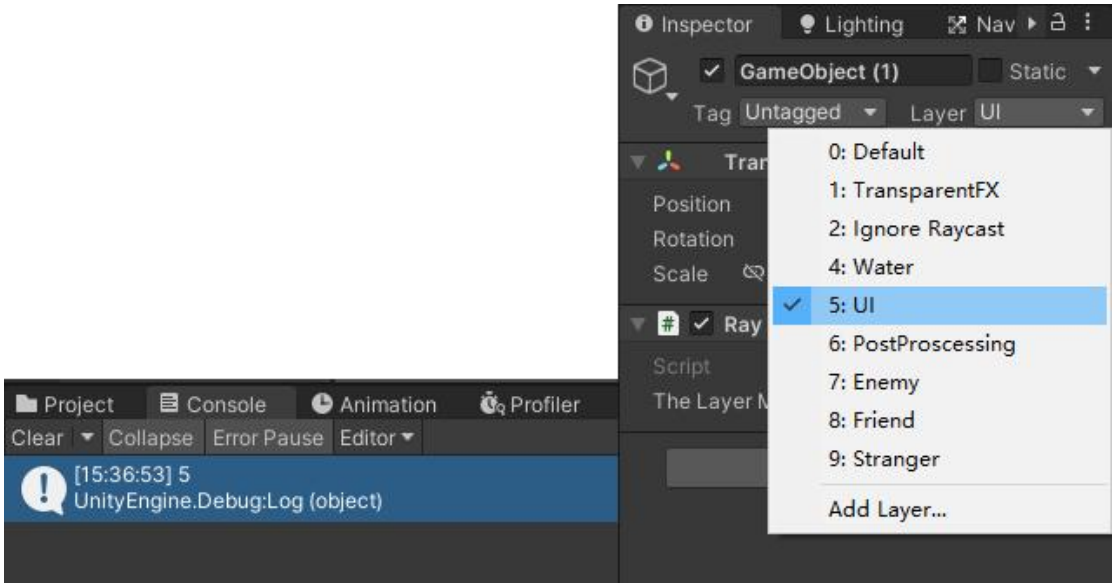

图 12.　打印出的当前脚本对象的 Layer

我们可以看到，打印的结果就是我们所设定的层数。再在代码中公开一个 LayerMask 变量，将这个 LayerMask 也打印出来，看看它的值和 Layer 有什么区别。

PS：

```
public LayerMask theLayerMask;
private void Start()
{
    Debug.Log("this is layer value "+this.gameObject.layer);
    Debug.Log("this is layerMask value " + theLayerMask.value);
}
```

我们先保持 LayerMask 层和对象的 Layer 层一致，也就是 LayerMask 只勾选一个层，然后我们来查看一下它们的值有什么不同。

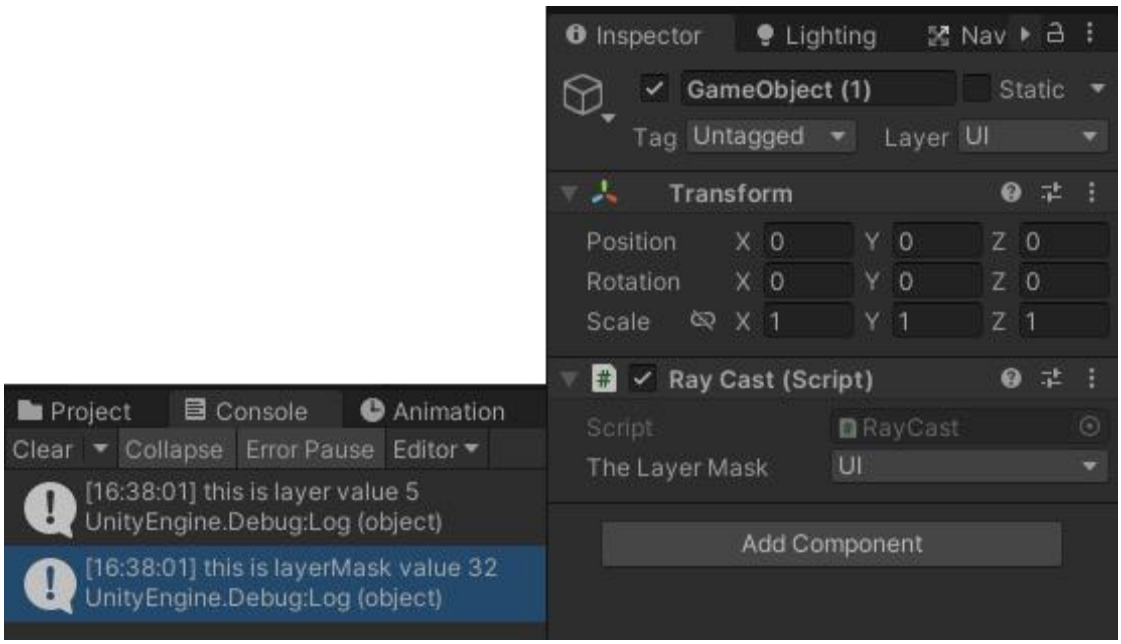

图 13.　保持 LayerMask 层和 Layer 层一致

从上图我们可以得知，我们的 Layer 输出的值是 5，而 LayerMask 输出的值则是 32。如

果，我们将 LayerMask 勾选多个层级，那么它的输出值又会有什么变化呢？

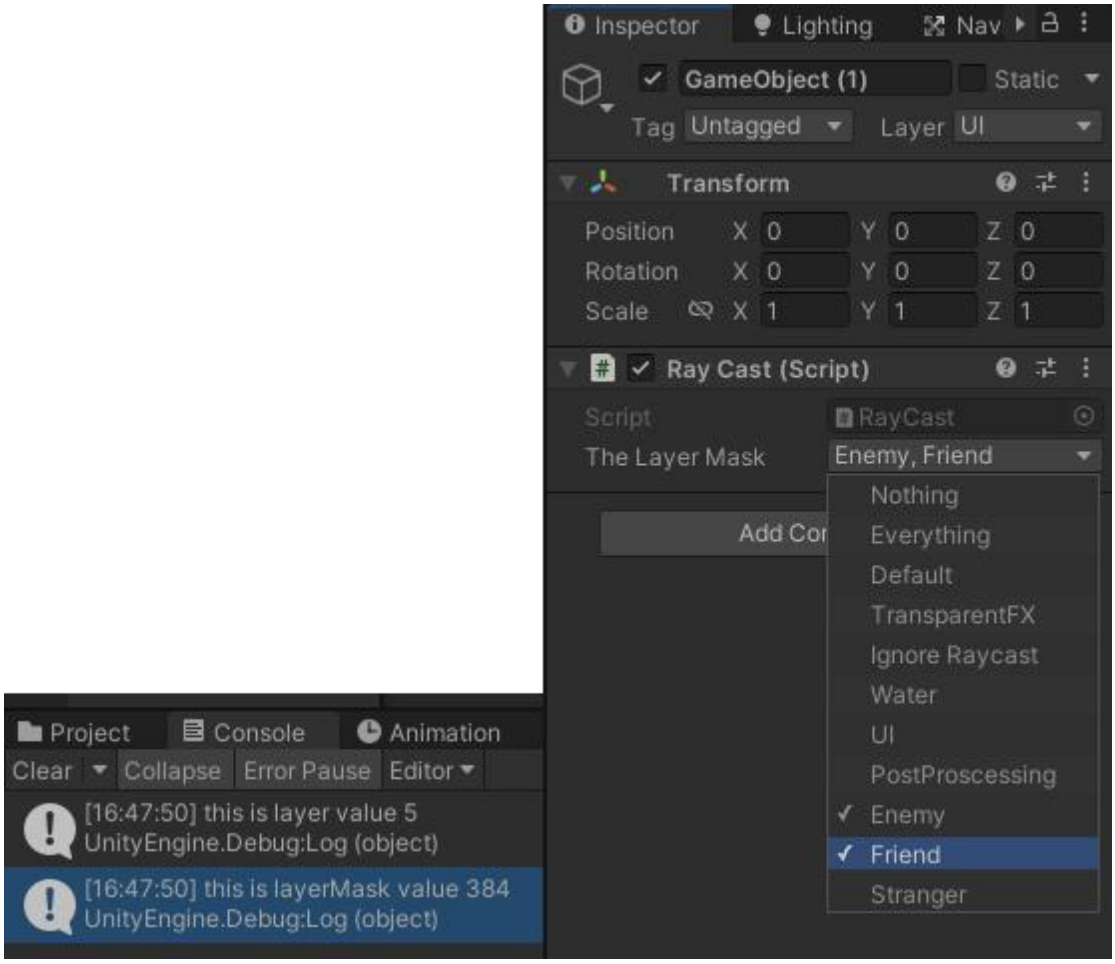

图 14.　LayerMask 勾选多个层级

如上图所示，我们勾选了第七层 Enemy 和第八层 Friend，我们的输出结果就是 384。那么这个值到底是怎么得到的呢？

前面已经说过了，Unity 中的 Layer 最多只能有 32 层，也即 32 位二进制。而我们在 LayerMask 中所勾选的层级，就表示其对应的二进制位为 1，而未勾选的部分则对应 0。之前我们只勾选了第五层 UI，它的 LayerMask 其实对应的就是 0010 0000 这个二进制数。我们可以对数值 1 左移 5 位来表示这个二进制数，即 1 << 5 = 0010 0000 = 32。

那么，LayerMask 勾选第七层和第八层时，所对应的二进制数就是 0001 1000 0000。我们可以将左移运算符和按位或运算符结合，来表示这个二进制数，即 1 << 7 | 1 << 8 = 0001 1000 0000 = 384。

PS：

```
public LayerMask theLayerMask;
    private void Start()
    {
        Debug.Log("this is layer value "+this.gameObject.layer);
        Debug.Log("this is layerMask value " + theLayerMask.value);
        Debug.Log("this is 5 Layer value " + (1 << 5));
        Debug.Log("this is 7 and 8 layerMask value " + (1 << 7 | 1 << 8));
    }
```

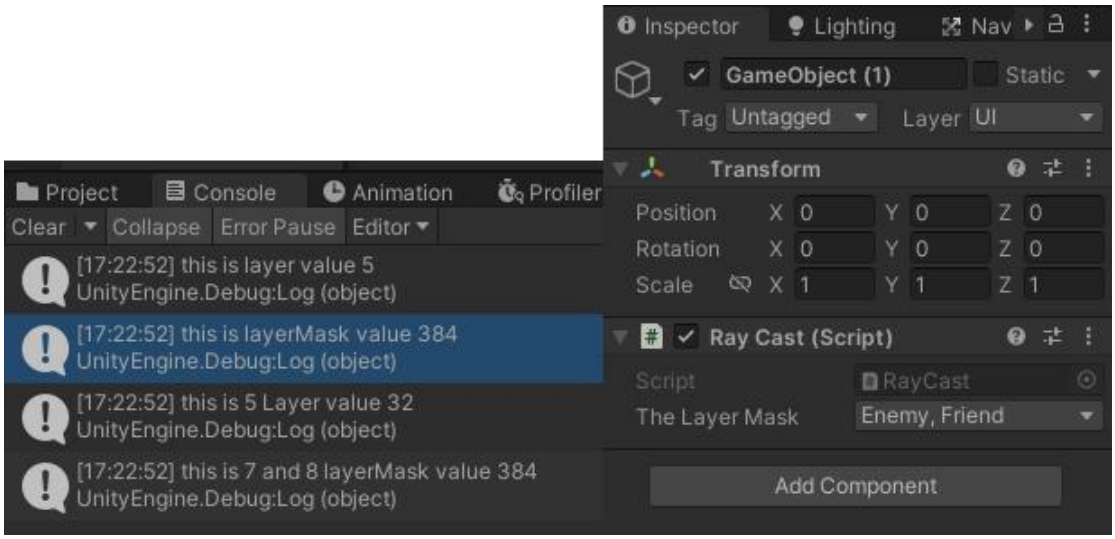

图 15.　打印出计算机计算的二进制结果

3. 使用单体射线检测

现在我们清楚了 Layer 和 LayerMask 的值，以及如何在计算机中表示它们，下面就可以使用射线检测 RayCast 了。

我们首先声明一个 LayeMask 类型的私有变量，然后使用 SerializeField 字段去强制序列化，使该私有变量在 Inspector 面板可见。要对 3D 对象使用射线检测则是使用 Physics 中的 RayCast；对 2D 对象使用射线检测则是使用 Physics2D 中的 RayCast。

最简单的射线检测只有两个参数，一个起始点坐标和射线的方向。这里需要特别说明一下，transform.forward、transform.right 以及 transform.up。

图 16.　transform.up、forward、right

从上图可以知道，transform.forward 对应的是 Unity 中的蓝色轴，也就是 Unity 中的 Z 轴；transform. right 对应的是 Unity 中的红色轴，也就是 Unity 中的 X 轴；transform.up 对应的是 Unity 中的绿色轴，也就是 Unity 中的 Y 轴。因为我这里的 2D 项目不会使用到 Z 轴，因此射线的方向就只能是 right 或者 up。

PS：

```
[SerializeField] //强制序列化，使 Private 在 Inspector 面板可见
    private LayerMask checkLayer;
    private void Update()
    {
        //3D 的射线检测
        Physics.Raycast(this.transform.position, this.transform. right);
        //2D 的射线检测
        Physics2D.Raycast(this.transform.position, this.transform. right);
    }
```

但，我们的射线检测的参数并不是只有起始坐标和方向这两个参数。我们还可以设置射线的最大距离，射线可以检测的 Layer 层以及射线反馈的信息等。为了方便调整射线检测，

我们需要将这些变量声明出来。

声明 Float 类型的变量去控制射线可以检测的最大距离，声明 RaycastHit2D 类型的参数去接收 2D 射线检测返回的信息，声明 RaycastHit 类型的参数去接收 3D 射线检测返回的信息。在 3D 射线检测中作为参数传入时，需在 RaycastHit 变量前加上 Out 关键字，2D 的射线检测返回结果就是 RaycastHit2D 类型的，直接赋值即可。

PS：

```
[SerializeField] //强制序列化，使 Private 在 Inspe 面板可见
    private LayerMask checkLayer;
    [SerializeField]
    private float maxDistance = 10;
    private RaycastHit hitInfo;
    private RaycastHit2D hitInfo2D;
    private void Update()
    {
        //3D 的射线检测
        Ray theRay = new Ray(this.transform.position, this.transform.right);
        Physics.Raycast(theRay, out hitInfo, maxDistance, checkLayer);
        //2D 的射线检测
        hitInfo2D = Physics2D.Raycast(this.transform.position, this.transform.right,
maxDistance, checkLayer);
    }
```

4. 旋转游戏对象

前面我们已经在 Update 函数中执行了射线检测，接下来就是编写一个方法去旋转我们的脚本对象。因为这里只需旋转一个轴即可，就直接使用欧拉角；如果需要避免万向节死锁，则使用四元数来旋转脚本对象。

为了便于控制旋转的速度，我们需要一个 Float 类型的参数去控制，这里将额外使用一个 Range 字段，去控制该变量的数值范围。

PS：

```
[SerializeField, Range(1, 10)] private float rotateSpeed = 5;
private void Update()
    {
        Vector3 temporary = new Vector3(0, 0, this.transform.eulerAngles.z -
Time.deltaTime * 10 * rotateSpeed);
        this.transform.eulerAngles = temporary;
        //2D 的射线检测
        hitInfo2D = Physics2D.Raycast(this.transform.position, this.transform.right,
maxDistance, checkLayer);
    }
```

5. 可视化射线

为了便于观察射线检测，可以使用 OnDrawGizmos 或 OnDrawGizmosSelected 函数来辅助。OnDrawGizmos 是始终绘制提示信息，而 OnDrawGizmosSelected 是只在脚本对象被选中时才开始绘制。

图 17.　Gizmos 方法区别

为了便于区分，我们可以为绘制出的辅助射线设置一个我们想要的颜色，然后使用 Gizmos.DrawLine 去绘制一条射线。Gizmos.DrawLine 的第一个参数是起始位置，第二个参数则是射线的结束位置。

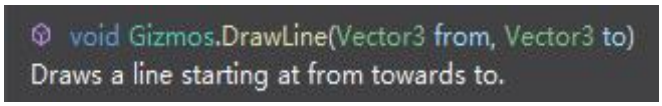

图 18.　Gizms.DrawLine 方法

PS：

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(this.transform.position, this.transform.right);
    }
```

我们回到 Unity 中，打开我们的 Gizmos。（根据版本的不同，Gizoms 的图标以及开启位置会有所不同，请自行操作）

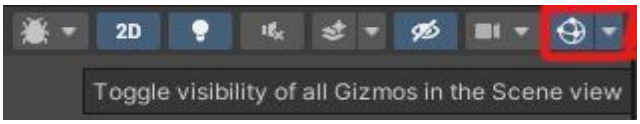

图 19.　打开 Gizmos

一旦打开了 Gizmos，我们就可以在 Unity 中看到一系列的辅助信息了。我们可以看到绘制出的蓝色射线似乎不正确，射线的终点位置并没有跟随着我们进行移动。这是什么原因？

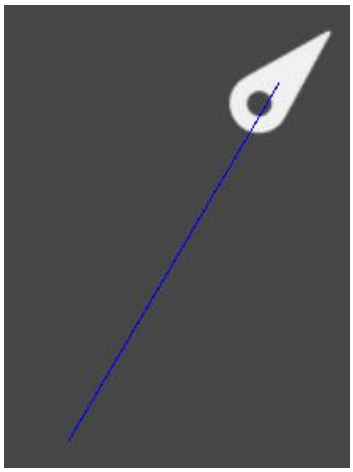

图 20.　绘制的射线位置似乎不太准确

其实，我们只需要仔细查看一下 transform.right 的解释就清楚了。transform.right、forward、up 对应的是世界空间下的，并不是局部空间的，因此我们的目标点将始终指向世界空间中

的那一点。

图 21.　transform.up、forward、right

要让辅助射线跟随对象移动，我们可以将对象的当前位置和 transform.right 相加。

PS：

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(this.transform.position, this.transform.position +
this.transform.right);
    }
```

在前面我们已经说了，游戏对象的 Layer 输出的它的层数，而 LayerMask 输出的是一个二进制数，且 LayerMask 是可以包含多个 Layer 层。那么，我们射线检测到的对象，就可以使用左移和按位与去判断 Layer 层，从而对不同 Layer 对象执行不同的行为。

PS：

```
private void OnDrawGizmos()
    {
        Color temporary = new Color(0,0,0);
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Friend")) != 0)
        {
            temporary = Color.green;
        }
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Enemy")) != 0)
        {
            temporary = Color.red;
        }
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Stranger")) != 0)
        {
            temporary = Color.blue;
        }
        Gizmos.color = temporary;
        Gizmos.DrawLine(this.transform.position, this.transform.position +
this.transform.right * (hitInfo2D.collider == null ? maxDistance : hitInfo2D.distance));
    }
```

6. 添加用于检测的对象

现在我们已经有了射线检测，也完善了不同 Layer 层所对应的行为，接下来我们就可以在 Unity 中添加用于检测的对象了。因为射线检测与 Unity 的物理系统相关，因此我们必须

为被检测的对象添加上 Collider 组件，否则射线检测无法检测到对象。

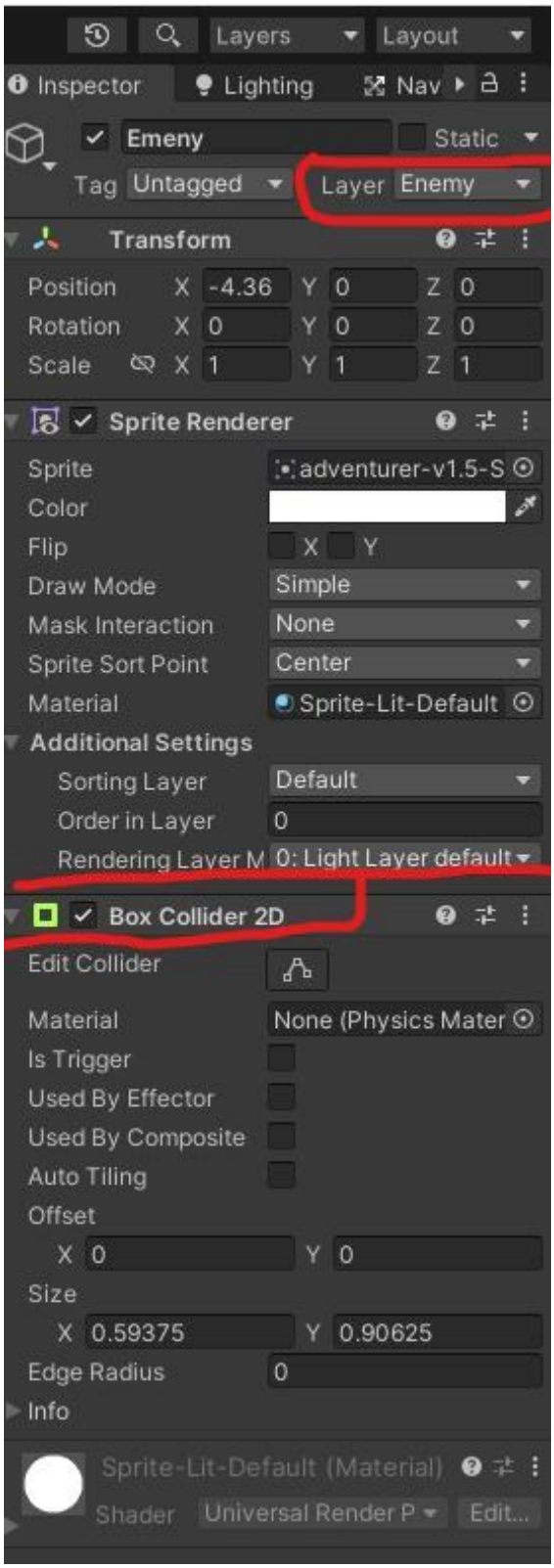

图 22.　射线检测的对象，需要包含 Collider 组件

7. 使用多对象的射线检测

前面讲了单对象的射线检测，它只会返回检测到的第一个对象的信息，如果在射线的检测方向上还有其他可被检测的对象，这些对象将无法被检测到。为此，我们就可以使用多对

象的射线检测，即 Physics2D.RaycastAll。

RaycastAll 和 RacyCast 功能上没大的区别，只是 RaycastAll 返回的是，射线方向上所有可被检测对象的信息；而 RacyCast 返回的是，射线方向上第一个被检测对象的信息，也就是说只要检测到一个对象，就不再继续向后检测了。

1) 射线检测的性能消耗

名称	速度
Raycast	1ms
RaycastNonAlloc	2 ms
RaycastAllTest	2 ms
Linecast	1 ms
BoxCastTest	17 ms
BoxCastNonAllocTest	18 ms
BoxCastAllTest	20 ms
CheckBoxTest	1 ms
OverlapBoxTest	62 ms
OverlapBoxNonAllocTest	24 ms
CapsuleCastTest	162 ms
CapsuleCastNonAllocTest	227 ms
CapsuleCastAllTest	1 ms
CheckCapsuleTest	84 ms
OverlapCapsuleTest	30 ms
OverlapCapsuleNonAllocTest	96 ms
SphereCastTest	171 ms
SphereCastNonAllocTest	226 ms
SphereCastAllTest	22o6 ms
CheckSphereTest	1 ms
OverlapSphereTest	33 ms
OverlapSphereNonAllocTest	16 ms

Ray 和 Line 的开销非常小，撇开这个可以发现，性能消耗的顺序是这样的。

同方法下不同模型 GC 开销：Box < Sphere < Capsule

同模型下不同方法 GC 开销：CheckXXX < OverlapXXX < XXXCast

十. Text Mesh Pro 效果

1. 打字机效果

一种最简单的想法就是，将一个字符串拆分成若干个字符，然后依次传入给 TextMeshPro 组件。

PS：

```
contentTetx.text = "";
foreach (char letter in textList[textLineIndex].ToCharArray())  //ToCharArray 将字符串转换为字符数组
{
    contentTetx.text += letter;
}
```

但这种方式需要产生多个 Substring（子串），给后续 GC 带来压力，且每次对 Text 赋值

就需要重新计算文字的网格位置。

1) 设置可见的文字数量

我们将完整字符串显示的时候就已经计算得到了最终的网格，那么就可以考虑利用这份网格数据，将需要显示的网格数据依次输出来实现字符的依次显示。

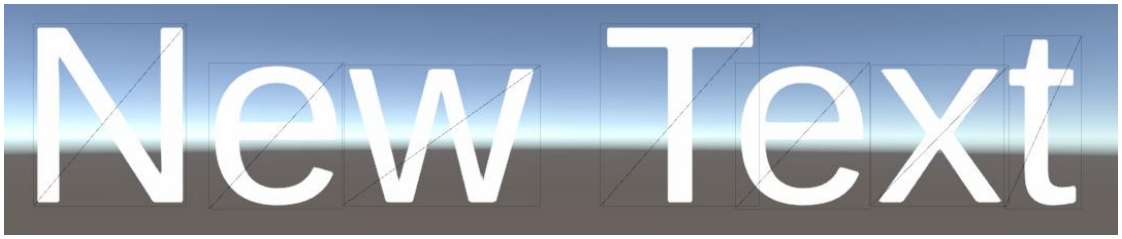

图 23. 每一个字符都有其对应的网格信息

要在代码中使用 Text Mesh Pro，需要引用一个命名空间，即 using TMPro。在 TextMeshPro 中，要显示的文字和网格信息都存储在 textInfo 这个变量中，通过 TMP_TextInfo 中的 characterCount 获取到文本对象中的总可见字符数，然后再设置文本组件中的最大可见字符数(m_TextComponent.maxVisibleCharacters)来设置当前显示的字符数量。

PS：

```
        [SerializeField] private TMP_Text m_TextComponent;
        [SerializeField] private bool hasTextChanged;
        [SerializeField] private bool repeatEffect = false;
[Header("这里是打字机效果的相关设置"), Space]
        [SerializeField, Range(0, 1), Tooltip("这里控制每个字符之间要间隔多久，数值越低打印速度越快")]
        private float typeWriterSpeed = 1.0f;
        [SerializeField, Range(0, 5), Tooltip("控制打印完成后要间隔多少秒才能再次开始打印，使用该数据需要开启 Repeat")]
        private float completeDelay = 1.0f;
        [SerializeField] private bool typeWriterComplete = false;

private IEnumerator TypeWriter(TMP_TextInfo textInfo)
        {
            //通过 TMP_TextInfo 中的 characterCount 获取文本对象中的总可见字符数
            int totalVisibleCharacters = textInfo.characterCount;
            //当前已经显示出的字符数
            int currentVisibleCount = 0;

            while (!typeWriterComplete)
            {
                if (hasTextChanged)
                {
                    //当文本对象发生改变时新的获取文本对象中的可见字符数
                    totalVisibleCharacters = textInfo.characterCount;
                    hasTextChanged = false;
                }
```

```
                if (currentVisibleCount > totalVisibleCharacters)
                {
                    //所有的字符都显示出来后，我们延迟一段真实时间
                    typeWriterComplete = !repeatEffect;
                    if (repeatEffect)
                        currentVisibleCount = 0;
                    else
                        currentVisibleCount = totalVisibleCharacters;

                    yield return new WaitForSecondsRealtime(completeDelay);
                }
                //当前允许显示的最大字符数
                m_TextComponent.maxVisibleCharacters = currentVisibleCount;
                //增加待显示的字符数
                currentVisibleCount += 1;
                yield return new WaitForSecondsRealtime(typeWriterSpeed);
            }
            yield return null;
        }
```

a. 优点

① 不用重新生成多个子字符串，也不会触发 TextMesh 对文字网格重新计算，减少多个 string 的 GC 和网格计算的消耗。

② 可以轻松处理富文本，因为富文本携带的信息已经被转化为 characterInfo 中的数据存储好了。

b. 缺点

① 对于 fontStyle 设置了下划线，删除线等效果暂时无法在动效中复现（目前暂未找到下划线和删除线的网格生成方式）

② GeometrySorting 设置为 Reverse 时，动效显示将出问题。（GeometrySorting 的值影响网格数据的顶点排序，Reverse 为反向保存的，即顶点数组从最后一个字符开始保存顶点坐标。目前还未研究清楚这种方式有何作用，暂时没想到合适的处理方式）

2) 修改网格数据实现

我们将完整字符串显示的时候就已经计算得到了最终的网格，那么就可以考虑利用这份网格数据，将需要显示的网格数据依次输出给 renderer 来实现字符的依次显示。

在 Text Mesh Pro 里的每个字符，都是绘制在由 4 个顶点构成的面片上的，我们只要想办法控制绘制出的顶点的数量，就能控制显示出的文字长度。

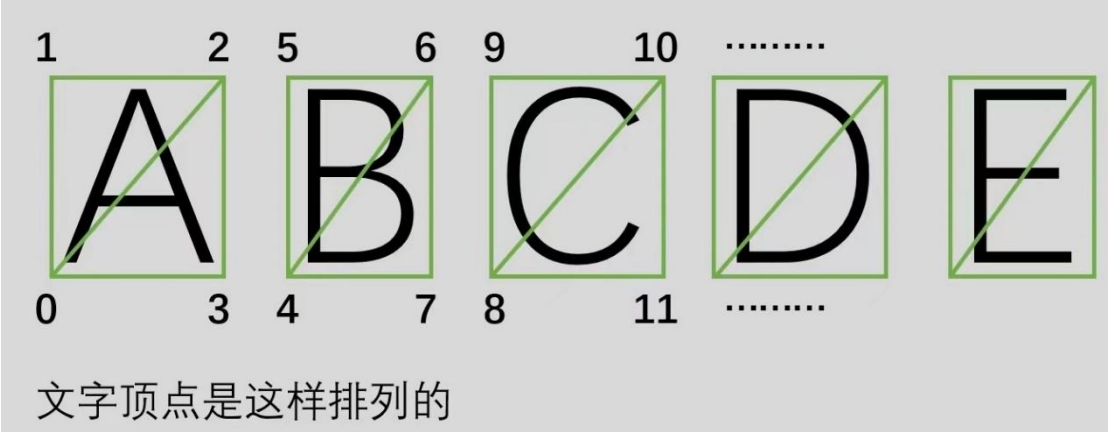

图 24.　Text Mesh Pro 字符的顶点构成

在 TextMeshPro 中，要显示的文字和网格信息都存储在 textInfo 这个变量中，变量的类型为 TMP_TextInfo。TMP_TextInfo 类型中，meshInfo 缓存了文字网格数据，可以修改该变量中的内容，之后调用 UpdateVertexData 方法，即可将网格数据传递给相应的 Renderer 中，所以要做的就是在合适的时机修改 meshInfo 中的数值。

PS：

```
IEnumerator TypeWriterEffect(TMPro.TMP_Text textComponent, float speed)
        {
            if (textComponent == null)
            {
                Debug.LogError("Input component is null!!!");
                yield break;
            }
            if (speed <= 0)
            {
                Debug.LogError("Input speed should be larger than 0!!!");
                speed = 1;
            }

            TMPro.TMP_TextInfo textInfo = textComponent.textInfo;

            // 开始播放打印机效果，首先清空所有文字网格
            for (int i = 0; i < textInfo.materialCount; ++i)
            {
                textInfo.meshInfo[i].Clear();
            }

textComponent.UpdateVertexData(TMPro.TMP_VertexDataUpdateFlags.Vertices);

            float timer = 0;
            float duration = 1 / speed;
            for (int i = 0; i < textInfo.characterCount; ++i)
            {
```

```
                while (timer < duration)
                {
                    yield return null;
                    timer += Time.deltaTime;
                }

                timer -= duration;

                TMPro.TMP_CharacterInfo characterInfo = textInfo.characterInfo[i];

                int materialIndex = characterInfo.materialReferenceIndex;
                int verticeIndex = characterInfo.vertexIndex;
                if (characterInfo.elementType == TMPro.TMP_TextElementType.Sprite)
                {
                    verticeIndex = characterInfo.spriteIndex;
                }
                if (characterInfo.isVisible)
                {
                    //顶点是顺时针构建的，构建的第一个顶点是左下角的那个
                    textInfo.meshInfo[materialIndex].vertices[0 + verticeIndex] =
characterInfo.vertex_BL.position;
                    textInfo.meshInfo[materialIndex].vertices[1 + verticeIndex] =
characterInfo.vertex_TL.position;
                    textInfo.meshInfo[materialIndex].vertices[2 + verticeIndex] =
characterInfo.vertex_TR.position;
                    textInfo.meshInfo[materialIndex].vertices[3 + verticeIndex] =
characterInfo.vertex_BR.position;

textComponent.UpdateVertexData(TMPro.TMP_VertexDataUpdateFlags.Vertices);
                }
            }
        }
```

每次 TextMesh 属性发生改变的时候，都会重新计算显示文字的顶点位置，每个字符的显示信息通过结构体 TMP_CharacterInfo 来保存。因此在清除掉所有网格数据后，再定时从该结构体中获取网格数据重新构建网格，来得到每个字符依次显示的动画效果。

2. 效果主体

在前面我们已经实现了打字机效果，下面我们就可以接着去实现其他的效果了。为了方便后续效果的展现，我们创建一个主体，在这个主体里去执行这些效果。

首先我们需要新增两个变量，一个用于存储 Text Mesh Pro 中的总字符数，另一个用于存储 Text Mesh Pro 的原始顶点数据，后续的很多效果都需要使用这个原始的顶点数据。

PS：

```
private int characterCount = 0;
private Vector3[] originalVertices;
```

因为我们在 Awake 函数中尝试获得了 Text Mesh Pro 组件，所以在 Start 函数中我们就可

以使用 Text Mesh Pro 组件相关的东西了。

PS：

```
private void Start()
    {
        m_TextComponent.ForceMeshUpdate();
        characterCount = m_TextComponent.textInfo.characterCount;
        originalVertices = m_TextComponent.textInfo.meshInfo[0].vertices;
    }
```

注意！在访问之前，一定要保证我们是获取到了组件的，否则将会出现错误。如果在 Start 函数阶段没有获取到 Text Mesh Pro 组件，那么就直接为当前脚本对象添加一个组件。

PS：

```
private void Start()
    {
        if (m_TextComponent == null)
        {
            gameObject.AddComponent<TMP_Text>();
            gameObject.TryGetComponent<TMP_Text>(out m_TextComponent);
        }
        m_TextComponent.ForceMeshUpdate();
        characterCount = m_TextComponent.textInfo.characterCount;
        originalVertices = m_TextComponent.textInfo.meshInfo[0].vertices;
    }
```

我们也可以为这个脚本添加一些属性，比如 RequireComponent 就会要求该脚本对象必须具有一个某种类型的组件，而 DisallowMultipleComponent 则会保证一个对象只能拥有一个该脚本。

在主体内执行一个 While 循环，如果我们不主动跳出，那么这些效果将会一直执行下去。在循环的开头需要强制更新一下网格的数据，然后使用两个临时变量去存储文字的信息并拷贝当前的顶点数据。

PS：

```
                    m_TextComponent.ForceMeshUpdate();
                    //textInfo 中存储了要显示的文字和网格信息
                    TMP_TextInfo textInfo = m_TextComponent.textInfo;
                    //拷贝网格的顶点数据，后面的部分效果将使用这份被拷贝的数据做插值
                    TMP_MeshInfo[] textInfoCopy = textInfo.CopyMeshInfoVertexData();

                    characterCount = textInfo.characterCount;
```

现在有了存储显示文字和网格信息的 textInfo，我们就可以遍历所有的文字，对每一个文字的数据进行一系列修改。因为每一个效果所使用到的数据会有所不同，如果我们在效果的方法中获取所需的数据，那么就很有可能造成一些不必要的性能消耗。为此，我们将效果所需的参数尽可能的写在主体内。

PS：

```
//TMP_CharacterInfo 包含有关单个文本元素（字符或子画面）的信息的结构。
                        TMP_CharacterInfo characterInfo = textInfo.characterInfo[i];
```

```
                    // 获取当前角色使用的材质索引。
                    int materialIndex =
textInfo.characterInfo[i].materialReferenceIndex;

                    // 获取此文本元素使用的第一个顶点的索引。
                    int vertexIndex = textInfo.characterInfo[i].vertexIndex;
                    //获取文本元素（字符或精灵）使用的网格的顶点颜色。
                    Color32[] vertexColor =
textInfo.meshInfo[characterInfo.materialReferenceIndex].colors32;
                    //meshInfo 中包含文本对象的顶点属性（几何图形）的结构。
                    Vector3[] vertices = textInfo.meshInfo[materialIndex].vertices;
                    Vector3[] copyVertices = textInfoCopy[materialIndex].vertices;
```

接着，我们就可以根据设定好的枚举，去执行不同的效果了。但是，在执行效果前，我们可以排除掉一些情况。我们的这些效果其实只需要影响可见的文字就行，对于那些不可见的文字以及空格，我们就直接跳过即可。

PS：

```
if (!characterInfo.isVisible)
                    {
                        //这里是为了避免字符为空格时，UnFold 效果不继续应用了
                        effectEnable[i + 1] = true;
                        continue;
                    }
                    else
                    {
                        //依据选择的类型去执行不同的方法
                        switch (effectType)
                        {
                            case TMPEffectType.Typewriter:
                                yield return StartCoroutine(TypeWriter(textInfo));
                                break;
                            default:
                                break;
                        }
                    }
```

最后，我们就需要遍历所有的网格信息，将我们的修改应用上去。要更新 Text Mesh Pro 的网格数据，我们需要使用 UpdateGeometry 方法；更新顶点数据使用 UpdateVertexData 方法。

PS：

```
for (int i = 0; i < textInfo.meshInfo.Length; i++)
                {
                    //meshInfo 中缓存了每个文字的网格数据
                    TMP_MeshInfo meshInfo = textInfo.meshInfo[i];
                    meshInfo.mesh.vertices = meshInfo.vertices;
                    //更新几何网格数据
```

```
                m_TextComponent.UpdateGeometry(meshInfo.mesh, i);
            }
            //更新顶点数据

m_TextComponent.UpdateVertexData(TMP_VertexDataUpdateFlags.Colors32);
```

在运行前，需要在开启打字机效果协程的前面加上 yield return;

1) 当前主体代码

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public enum EffectType
{
    typewriter = 0,
}

[RequireComponent(typeof(TextMeshProUGUI)),DisallowMultipleComponent]
public class Test : MonoBehaviour
{
    public TMP_Text m_TextComponent;
    [Range(0,1)]public float speed = 1;
    [SerializeField]
    private EffectType effectType = EffectType.typewriter;
    private int characterCount = 0;//总字符数
    private Vector3[] originalVertices;//原始的顶点数据

    private void Awake()
    {
        gameObject.TryGetComponent<TMP_Text>(out m_TextComponent);
    }

    private void Start()
    {
        if (m_TextComponent == null)
        {
            gameObject.AddComponent<TMP_Text>();
            gameObject.TryGetComponent<TMP_Text>(out m_TextComponent);
        }
        m_TextComponent.ForceMeshUpdate();
        characterCount = m_TextComponent.textInfo.characterCount;
        originalVertices = m_TextComponent.textInfo.meshInfo[0].vertices;
        StartCoroutine(MainBody());
```

```
    }
    private IEnumerator MainBody()
    {
        while (true)
        {
            m_TextComponent.ForceMeshUpdate();
            //textInfo 中存储了要显示的文字和网格信息
            TMP_TextInfo textInfo = m_TextComponent.textInfo;
            //拷贝网格的顶点数据，后面的部分效果将使用这份被拷贝的数据做插值
            TMP_MeshInfo[] textInfoCopy = textInfo.CopyMeshInfoVertexData();

            characterCount = textInfo.characterCount;
            for (int i = 0; i < characterCount; i++)
            {
                //TMP_CharacterInfo 包含有关单个文本元素（字符或子画面）的信息
的结构。
                TMP_CharacterInfo characterInfo = textInfo.characterInfo[i];
                // 获取当前角色使用的材质索引。
                int materialIndex = textInfo.characterInfo[i].materialReferenceIndex;

                // 获取此文本元素使用的第一个顶点的索引。
                int vertexIndex = textInfo.characterInfo[i].vertexIndex;
                //获取文本元素（字符或精灵）使用的网格的顶点颜色。
                Color32[] vertexColor =
textInfo.meshInfo[characterInfo.materialReferenceIndex].colors32;
                //meshInfo 中包含文本对象的顶点属性（几何图形）的结构。
                Vector3[] vertices = textInfo.meshInfo[materialIndex].vertices;
                Vector3[] copyVertices = textInfoCopy[materialIndex].vertices;

                //跳过不可见的字符，因此当字符不可见时，将不执行任何操作。
                if (!characterInfo.isVisible)
                {
                    continue;
                }
                else
                {
                    switch (effectType)
                    {
                        case EffectType.typewriter:
                            yield return StartCoroutine(TypeWriter());
                            break;
                        default:
                            break;
                    }
```

```
                }
            }

            for (int i = 0; i < textInfo.meshInfo.Length; i++)
            {
                //meshInfo 中缓存了每个文字的网格数据
                TMP_MeshInfo meshInfo = textInfo.meshInfo[i];
                meshInfo.mesh.vertices = meshInfo.vertices;
                //更新几何网格数据
                m_TextComponent.UpdateGeometry(meshInfo.mesh, i);
            }
            //更新顶点数据

m_TextComponent.UpdateVertexData(TMP_VertexDataUpdateFlags.Colors32);
            yield return null;
        }
    }

    private IEnumerator TypeWriter()
    {
        m_TextComponent.ForceMeshUpdate();
        TMP_TextInfo textInfo = m_TextComponent.textInfo;
        int total = textInfo.characterCount;
        bool complete = false;
        int current = 0;
        while (!complete)
        {
            if (current > total)
            {
                current = total;
                yield return new WaitForSecondsRealtime(1);
                complete = true;
            }
            m_TextComponent.maxVisibleCharacters = current;
            current += 1;
            yield return new WaitForSecondsRealtime(speed);
        }
        yield return null;
    }
}
```

3. Text Mesh Pro 浮动效果

在前面我们已经完成了主体部分的编写，现在我们就可以继续编写下一个效果了。在编写下一个效果之前，我们需要在效果枚举内，添加上效果所对应的枚举元素。添加完成以后，我们就可以在主体内使用这个枚举元素了。

为了方便在外部控制浮动的频率和范围，我们需要添加两个 Float 类型的变量去控制。

PS：

```
[SerializeField , Range(1, 10)] private float frequence = 1.0f;
[SerializeField, Range(1, 10), Tooltip("控制浮动效果的移动范围")]
        private float floatRange = 1.0f;
```

因为访问修饰符使用的是私有类型 Private，在 Inspector 面板中是不会被显示出来的，因此我们需要使用 SerializeField 将其强制序列化。为了限定变量的范围，我们可以使用属性 Range，它的第一个参数是最小值，第二个参数是最大值。

有时候我们忘记变量有什么作用，我们就可以属性 Tooltip，来为这个变量添加一个提示信息。一旦变量带有 Tooltip，那么在 Inspector 面板中，鼠标移动到变量上一段时间，就会显示出设定好的提示信息。

因为字体的浮动效果是通过顶点动画实现的，因此我们需要知晓是哪一个文字，以及是哪些顶点将被修改，也即主体里的 characterInfo 和 vertices。

因为 Text Mesh Pro 中的字体是使用面片进行显示的，每个面片只有四个顶点，因此循环的次数为 4 次。我们只需要先存储文字当前的顶点信息，然后用修改后的顶点信息与原顶点相加，就可以将文字的位置进行移动了。

为了保证浮动效果的周期性，需要使用三角函数去修改顶点。

PS：

```
private void Floating(int vertexIndex, ref Vector3[] vertices)
        {
            //因为 Text Mesh Pro 中的字体是使用面片进行显示的，每个面片只有四个顶点，因此循环次数为 4 次
            for (int i = 0; i < 4; i++)
            {
                //从顶点数组中取出单个顶点，并对其进行计算
                Vector3 originalValue = vertices[vertexIndex + i];
                float verticesYPosition = (Mathf.Sin(Time.time * frequence * Mathf.PI + originalValue.x)) * floatRange;
                //执行顶点动画
                vertices[vertexIndex + i] = originalValue +    new Vector3(0, verticesYPosition, 0);
            }
        }
```

4. Text Mesh Pro 自定义斜体字

Text Mesh Pro 的默认斜体只有一个倾斜角度，如果我们想要修改字体的倾斜角度，就需要去修改 Text Mesh Pro 字体的顶点位置。要让 Text Mesh Pro 的字体发生倾斜，我们就需要使用到错切变换了。错切变换是一种数学变换，最简单的理解就是保持四边形的两个顶点不动，然后拖动另外的两个顶点，这个过程就是错切的过程。在错切变换中，坐标保持不变的轴被称为依赖轴，余下的坐标轴被称为方向轴。

因为这里是针对 2D 的 Text Mesh Pro，因此后面就只讨论二维错切变换。二维错切变换可以分为水平和垂直，水平错切变换矩阵就是保持 Y 坐标不变，仅改变 X 坐标的的过程，这个变换的等式可以表示为$\begin{cases} x' = x + y\tan\alpha \\ y' = y \end{cases}$，矩阵表示为$\begin{pmatrix} 1 & \tan\alpha \\ 0 & 1 \end{pmatrix}$；垂直错切变换矩阵就

是保持 X 坐标不变，仅改变 Y 坐标的的过程，这个变换的等式可以表示为$\begin{cases} x' = x \\ y' = x\tan\alpha + y \end{cases}$，矩阵表示为$\begin{pmatrix} 1 & 0 \\ \tan\alpha & 1 \end{pmatrix}$。

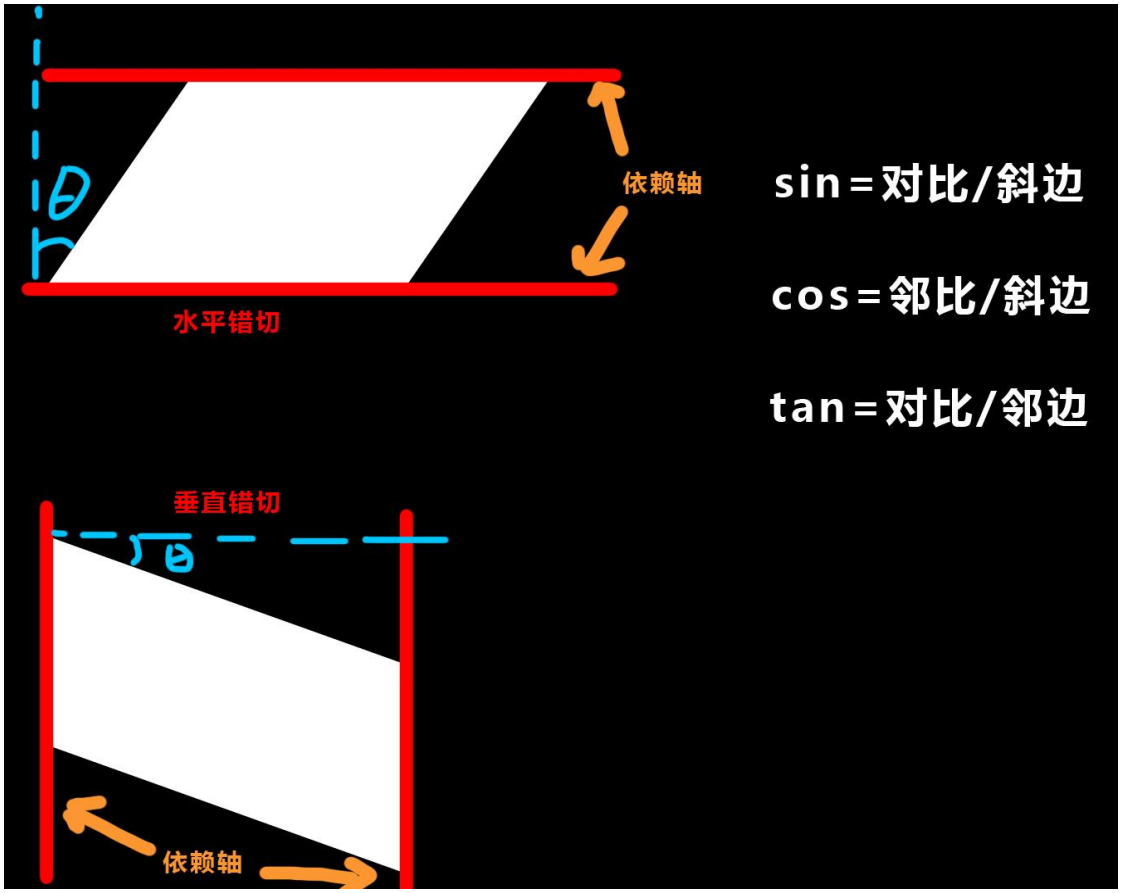

图 25.　错切变换

我们在脚本中的效果枚举中加入斜体字效果所对应的枚举，即 Italic，接着我们再添加上该效果所需的参数。在这里我只设置四个参数，两个 Float 类型的参数分别控制水平错切的角度和垂直错切的角度，布尔类型的变量用于反转错切的方向，三维向量的数组存储最初的顶点样式。

PS：

```
[Header("斜体效果的相关设置"), Space]
        [SerializeField,Range(0,60),Tooltip("这里使用的是角度为单位，不是弧度为单位")]
        private float slopeXAngle = 0.0f;//这里使用的是弧度
        [SerializeField, Range(0, 45), Tooltip("这里使用的是角度为单位，不是弧度为单位")]
        private float slopeYAngle = 0.0f;//这里使用的是弧度
        [SerializeField,Tooltip("勾选该选项，将使字体向另一个方向偏转")]
        private bool reverseDirection = false;
        private Vector3[] originalVertices;
```

当添加完成效果所需的参数后，就可以创建对应的方法了。斜体效果的方法需要两个形参，即所有的顶点以及当前文字的起始顶点索引。我们需要在方法体中去修改当前字符所对

应的 4 个顶点位置，然后将修改后的顶点传递回去，因此循环次数为 4 次。

因为我们前面的参数是角度值，而数学库中的 Tan 方法使用的是弧度，因此在这里使用时需要将其转换为弧度，也即需要乘上 PI 除以 180。

PS：

```
private void Italic(Vector3[] vertices, int vertexIndex)
    {
        //因为 Text Mesh Pro 中的字体是使用面片进行显示的，每个面片只有四个顶点，因此循环次数为 4 次
        for (int j = 0; j < 4; j++)
        {
            //将角度转为弧度，执行错切变换
            float xAngle = slopeXAngle * (Mathf.PI / 180);
            float yAngle = slopeYAngle * (Mathf.PI / 180);
            /*水平方向进行变换（角度是与 Y 轴的夹角） 垂直方向进行变换（角度是与 X 轴的夹角）
            xyz    1    0 0                    xyz    1   tan 0
                   tan 1 0                          0   1    0
                   0    0 1                         0   0    1
            */

            vertices[vertexIndex + j] = originalVertices[vertexIndex + j] +
(reverseDirection ? -1 : 1)
                * new Vector3(originalVertices[vertexIndex + j].y * Mathf.Tan(xAngle),
                originalVertices[vertexIndex + j].x * Mathf.Tan(yAngle), 0);
        }
    }
```

5. 字体生长效果

从这里开始，我们就可以开始创建 Text Mesh Pro 的顶点动画了。首先，我们需要新增一个 float 类型的数组变量，这个变量将存储每一个字符所对应的过渡值。

PS：

```
private float[] lerpValue;
```

因为我们的这个效果实际上是一个动画，因此它自然也就需要一个过渡的结束时间。我们继续新增一个 float 类型的变量，这个变量将用于控制过渡动画的结束时间。

为了方便在外部修改这个参数的值，我们为这个参数添加上滑动条属性，来方便我们在 Inspector 面板中的参数调整。这里再说明一下，Range 属性的第一个参数是最小值，第二个参数是最大值。

PS：

```
[SerializeField, Range(0, 2),Tooltip("设置展开动画的完成时间")]
        private float overallLerpTime = 1f;
```

接下来，我们就可以分析一下这个字体生长动画了。首先，Text Mesh Pro 的每个字体都具有四个顶点，所以在最初的时候需要将 Text Mesh Pro 的每一个顶点位置进行重新设置。如图所示，我们将顶点 1 的初始位置设置到顶点 0 的位置，将顶点 2 的初始位置设置到顶点 3 的位置。

然后再使用前面定义的变量结合 lerp 函数，去实现这个顶点动画的效果。

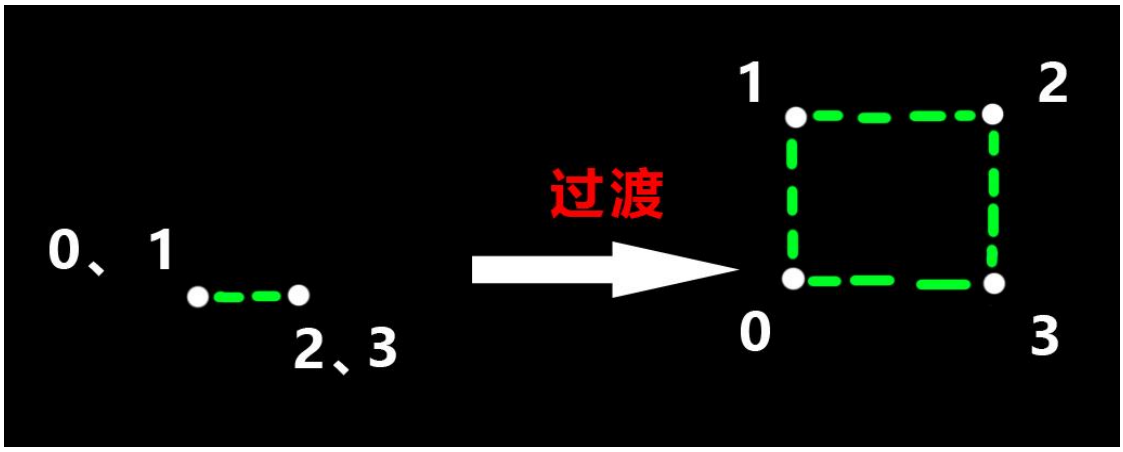

图 26.　字体生长

接下来我们就可以创建一个名为 Grow 的方法，依据前面的分析我们为这个方法赋予对应的形参。首先我们需要知道，执行效果字符的起始顶点索引，因此需要一个 vertexIndex。然后，我们还需要目标顶点的信息和当前字符的过渡值，以便于我们使用 lerp 函数，因此需要网格的 vertices、字符的 lerpValue 和拷贝出的 copyVertices。

```
//拷贝网格的顶点数据，后面的部分效果将使用这份被拷贝的数据做插值
TMP_MeshInfo[] textInfoCopy = textInfo.CopyMeshInfoVertexData();

characterCount = textInfo.characterCount;
for (int i = 0; i < characterCount; i++)
{
    //TMP_CharacterInfo包含有关单个文本元素（字符或子画面）的信息的结构。
    TMP_CharacterInfo characterInfo = textInfo.characterInfo[i];
    // 获取当前字符使用的材质索引。
    int materialIndex = characterInfo.materialReferenceIndex;

    // 获取此文本元素使用的第一个顶点的索引。
    int vertexIndex = characterInfo.vertexIndex;
    //获取文本元素（字符或精灵）使用的网格的顶点颜色。
    Color32[] vertexColor = textInfo.meshInfo[characterInfo.materialReferenceIndex].colors32;
    //meshInfo中包含文本对象的顶点属性（几何图形）的结构。
    Vector3[] vertices = textInfo.meshInfo[materialIndex].vertices;
    Vector3[] copyVertices = textInfoCopy[materialIndex].vertices;
```

图 27.　对引用类型使用深拷贝

注意！这里的 copyVertices，是我们前面对整个网格信息进行的深拷贝。这里一定不能将原本的 vertices 赋值给 copyVertices，不然就会因为值类型和引用类型的差别从而导致错误的结果。

PS：

```
#region 生长
        /// <summary>
        /// 文本内容执行生长动画效果，也就是顶点 1 和 2 最开始位于顶点 0 和 4 的位置
        /// </summary>
        /// <param name="index">顶点索引</param>
        /// <param name="vertices">存储原始顶点的数组</param>
        /// <param name="copyVertices">存储备份顶点的数组</param>
        /// <param name="lerpValue">线性插值的值</param>
        private void Grow(int index, Vector3[] vertices, Vector3[] copyVertices, float
```

```
lerpValue)
        {
            //如果传入的是左下角的顶点 0，那么就是左下角和左上角顶点之间的中点
            Vector3 a = copyVertices[index];
            Vector3 a2 = copyVertices[index + 3];
            //lerp 不允许返回的值超过 a 和 b 规定的范围，当 lerp 为 0 时返回值为 a，当 lerp 为 1 时返回值为 b。
            vertices[index] = Vector3.Lerp(a, copyVertices[index], lerpValue);
            vertices[index + 3] = Vector3.Lerp(a2, copyVertices[index + 3], lerpValue);
            vertices[index + 1] = Vector3.Lerp(a, copyVertices[index + 1], lerpValue);
            vertices[index + 2] = Vector3.Lerp(a2, copyVertices[index + 2], lerpValue);
        }
        #endregion
```

我们的逻辑代码也非常的简单，只需取得对应的目标点位置，然后使用 lerp 函数即可。

完成逻辑代码后，记得在前面的枚举中将对应的类型给添加进去，这样才方便我们在 switch 中使用。与此同时，因为我们需要使用到字符的 lerpValue，而这个变量我们还没有对其进行一个初始化。因此，需要在 Start 函数中，将其进行初始化然后再在 Switch 中对其进行运用。

6. 使用模板方法进行边角缩放

前面我们已经实现了字体的生长动画，现在我们就可以编写下一个效果了，即边角缩放动画。为了节省时间，我们直接将前面的代码复制过来，做一些简单的修改就可以得到边角缩放的动画。

首先我们需要新增一个枚举，这个枚举用于设置边角缩放动画从字符的哪个顶点开始，同时我们也需要将这个新效果也添加到前面的效果枚举中。

```
6 个引用
public enum EffectType
{
    typewriter = 0,
    Floating = 1,
    Italic = 2,
    Grow = 3,
    CornerZoom = 4,
}

0 个引用
public enum CornerZoomType
{
    BottomLeft = 0,
    TopLeft = 1,
    TopRight = 2,
    BottomRight = 3,
}
```

图 28.　添加边角缩放控制枚举

紧接着，我们就可以在类中使用新增的枚举去创建一个变量了。完成后就可以在方法的形参中添加两个新的 int 型形参，一个用于传递当前的字符索引，另一个就是前面设置的边角缩放枚举。

同时，我们还需要将控制过渡的形参删除，因为我们将在这个方法体内设置对应字符的过渡值。

现在我们所需的形参都已经设置好了，下面就可以删除掉那些无用的逻辑了。删除完成

后，我们就在方法体内写入我们需要的逻辑代码。我们首先需要设置过渡值，这就直接使用前面主体中的代码即可；然后我们再设置字符的边角缩放顶点，这需要使用到拷贝出的顶点数组。

PS：

```
private void CornerZoom(int characterIndex,int vertexIndex,int cornerIndex, Vector3[] vertices, Vector3[] copyVertices)
        {
            lerpValue[characterIndex] += 1 / overallLerpTime * Time.deltaTime;
            Vector3 corner = copyVertices[cornerIndex];
            //lerp 不允许返回的值超过 a 和 b 规定的范围，当 lerp 为 0 时返回值为 a，当 lerp 为 1 时返回值为 b。
            vertices[vertexIndex] = Vector3.Lerp(corner, copyVertices[vertexIndex], lerpValue[characterIndex]);
            vertices[vertexIndex + 3] = Vector3.Lerp(corner, copyVertices[vertexIndex + 3], lerpValue[characterIndex]);
            vertices[vertexIndex + 1] = Vector3.Lerp(corner, copyVertices[vertexIndex + 1], lerpValue[characterIndex]);
            vertices[vertexIndex + 2] = Vector3.Lerp(corner, copyVertices[vertexIndex + 2], lerpValue[characterIndex]);
        }
    }
```

现在，这个方法体就已经编写完成了。但是有一个问题，现在我们的代码效果都是针对整个字符串的顶点进行应用的，后面我们将让这些效果逐字符进行应用，难道我们就需要再单独编写一个该效果的方法用于逐字符应用吗？

答案肯定是不用的，我们这个时候就需要使用到模板方法了。模板方法就像填空题一样，该方法已经有了大量的内容，但是留有部分的空白区域。那么这部分空白的部分是什么呢？答案就是函数，我们使用委托来让函数能够作为参数进行传递。

因为我们的所使用的函数是没有返回值的，因此这个委托就可以直接使用 Action 委托；如果是有返回值的函数，则使用 Func 委托。当然，我们也可以使用自定义委托，只是使用 Action 委托和 Func 委托要方便一点。

注意！使用 Action 委托和 Func 委托，我们需要引用命名空间 System。并且，我们传递的方法，其参数列表必须和委托的参数列表一致，不然我们就无法将方法作为参数进行传递。

7. 效果的逐字应用

前面已经完成了模板方法，接下来就可以准备让效果能够应用到单个字符上。

首先，我们需要一个 bool 类型的数组，其将用来确定当前字符是否可以应用效果。还需要一个整型变量，用来设置字符的可见数量。这两个变量新增完毕后，我们需要在 Start 函数中为其赋予初始值。

PS：

```
private bool[] effectEnable;
private int currentVisibleCount;
public int CurrentVisibleCount
{
    get => currentVisibleCount;
    private set
```

```
        {
            currentVisibleCount = (value > characterCount) ? characterCount : value;
            if (m_TextComponent != null)
            {
                m_TextComponent.maxVisibleCharacters = currentVisibleCount;
                m_TextComponent.ForceMeshUpdate();
            }
        }
    }
```

```
CurrentVisibleCount = 0;
effectEnable = new bool[characterCount];
```

图 29.　新增变量并在 Start 中赋初值

接下来我们就可以回到我们的模板方法中，来编写我们的逻辑代码了。

因为通过委托传递的方法，我们是必须要进行调用的，不论是逐字符应用还是整体应用。因此，我们的逐字符逻辑代码需要写在委托方法的前面。

我们首先新增一个 bool 类型变量，这将用于判断是否执行逐字符的逻辑代码。

PS：

```
[SerializeField, Tooltip("设置是否对网格整体应用效果")]
    private bool overall = true;
```

接下来，我们需要判断是否是第一个字符，且第一个字符是否还未应用逻辑。如果这些条件满足，我们就设置第一个字符的效果为开启状态，并让当前可见字符数为 1。

PS：

```
if (characterIndex == 0 && !effectEnable[characterIndex])
                {
                    CurrentVisibleCount = 1;
                    effectEnable[characterIndex] = true;
                    lerpValue[characterIndex] = 0;
                }
```

编写完第一个字符的处理逻辑后，我们就可以继续编写后续字符的逻辑了。

我们首先需要判断当前字符是不是还有后续字符，且是否可以应用效果。如果条件满足，我们就继续判断当前字符的 lerpValue 是否大于等于 1。只有当前面的字符完成了过渡动画后，我们才能继续执行后续字符的效果。

PS：

```
if (characterIndex == (CurrentVisibleCount - 1) && (characterIndex + 1) < characterCount
&& effectEnable[characterIndex])
                {
                    if (lerpValue[characterIndex] >= 1)
                    {
                        effectEnable[characterIndex + 1] = true;
```

```
                CurrentVisibleCount++;
                lerpValue[characterIndex + 1] = 0;
            }
        }
```

如果我们此时返回 unity 中使用逐字符方案，会发现文字都消失了。这是因为，我们之前在主体方法中跳过了不可见字符，而逐字符方案会在一开始就将所有字符设置为不可见。因此，我们需要返回主体中，去修改对于不可见字符的处理方法。

PS：

```
if (!characterInfo.isVisible && i < (characterCount - 1) && i != 0)
            {
                //这里是为了避免字符不可见时，逐字符效果就不继续应用了
                if (!overall)
                {
                    effectEnable[i] = true;
                    lerpValue[i] = 1;
                    if (lerpValue[i] >= 1 && i == (CurrentVisibleCount - 1))
                    {
                        effectEnable[i + 1] = true;
                        CurrentVisibleCount += 1;
                        lerpValue[i + 1] = 0;
                    }
                }
                continue;
            }
```

修改完成后回到 Unity 中，我们就可以看到正确的运行效果了。但是此时的边角缩放逐字效果，其原始位置都是从第一个字符处开始的，我们需要调整一下。让其可以从对应字符的边角处开始过渡，而不是都从第一个字符的边角开始。

我们只需要新增一个 bool 类型的变量即可，用这个变量去调整边角缩放的起始顶点位置。

PS：

```
private void CornerZoom(int characterIndex,int vertexIndex,int cornerIndex, Vector3[] vertices, Vector3[] copyVertices)
        {
            lerpValue[characterIndex] += 1 / overallLerpTime * Time.deltaTime;
            Vector3 corner = copyVertices[usePerCharCorner ? (vertexIndex + cornerIndex) : cornerIndex];
            //lerp 不允许返回的值超过 a 和 b 规定的范围，当 lerp 为 0 时返回值为 a，当 lerp 为 1 时返回值为 b。
            vertices[vertexIndex] = Vector3.Lerp(corner, copyVertices[vertexIndex], lerpValue[characterIndex]);
            vertices[vertexIndex + 3] = Vector3.Lerp(corner, copyVertices[vertexIndex + 3], lerpValue[characterIndex]);
            vertices[vertexIndex + 1] = Vector3.Lerp(corner, copyVertices[vertexIndex + 1], lerpValue[characterIndex]);
```

```
                vertices[vertexIndex + 2] = Vector3.Lerp(corner, copyVertices[vertexIndex + 2],
lerpValue[characterIndex]);
            }
```

8. 总体应用

十一. 读取 TXT 文本文件

注意：要读取的 TXT 文本文件的编码类型要为 utf-8，不然会出现中文乱码或者直接不显示，如果是其它编码方式可以把 Txt 文本文件另存为 UTF-8 的格式。

1. 使用 TextAsset 直接在 Inspector 面板赋值

在 Unity 中，Txt 文本文件的类型是 TextAsset，我们可以直接声明一个该类型的变量，然后在 Inspector 面板中为其直接赋值。

PS：

public TextAsset text;

图 30. TextAsset 资源使用

2. 使用相对路径加载资源

在 Unity 中，我们可以将需要动态加载的资源放置在 Resources 文件夹下，在代码中就可以使用 Resources.Load 去加载这一文件夹内的资源。

Resources.Load 通常是用于预制体的加载，但其实 Resources.Load 可以加载在 Resources 文件夹下几乎所有的资源，只是加载资源需要相对应的类型。对于 TXT 文本文件，使用的就是上面提到的 TextAsset 类型。

PS：

TextAsset textAsset = Resources.Load(...) as TextAsset;

Resources.Load 中的参数填的是 Resources 文件夹下的相对路径，不需要带文件的后缀名。

图 31. Recources 文件夹下的 Txt 文本文件

PS：

text = Resources.Load("TEST") as TextAsset;

3. 解析 Txt 文本文件

一个文本文件里的内容往往是不确定的，因为除了比较常用的 JSON、XML 等格式可以直接使用现成的工具或库进行解析以外，不同的人或项目所使用的格式都不确定。

但是，一个 Txt 文本文件里大概率是会有空格和换行的，这两种类型分别可以使用空字符' '和反斜杠 n '\n'来进行解析分离。不同的系统所对应的换行符会有所不同，为此在进行解析时需要将这些情况都进行一个考虑，它们包括回车换行'\r\n '、回车'\r '、'\\n '。至于为什么要额外考虑一个'\\n '，则是因为字符串要对反斜杠 \ 进行转义，其结果就变为了 \\n。

我们需要先将加载完成的 TXT 文本文件转换为字符串，然后再从这个字符串中去分离出子串，分离字符串我们需要用到 Split。

PS：

```
string tmp = text.text;
textTemporary = tmp.Split("\n");
```

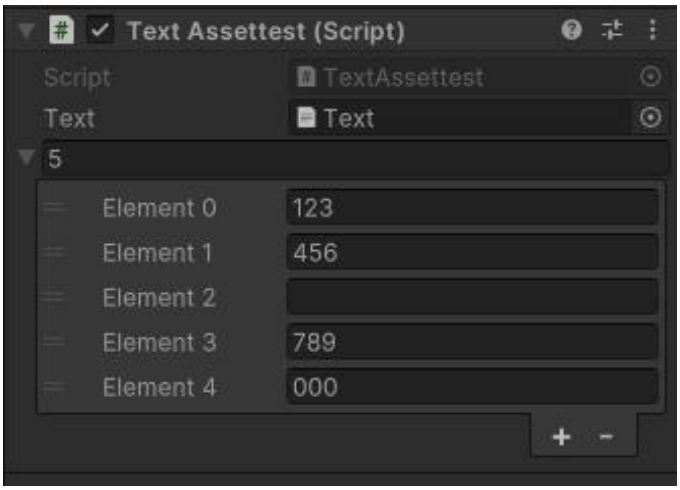

图 32.　对 TXT 文本文件里的内容按照换行符进行分割

同时，我们也注意到，Split 是具有多个重载方法的。我们可以设置其分割出的子串数量，参数设置为 1 就是保持原本的字符串不变，设置为 2 就是根据关键字去分割出两个子串。我们所能设置的上限，就是默认状态下所分离出的子串数量，超出这个值就无法再进行分割了。

PS：

```
string tmp = text.text;
textTemporary = tmp.Split("\n",2);
```

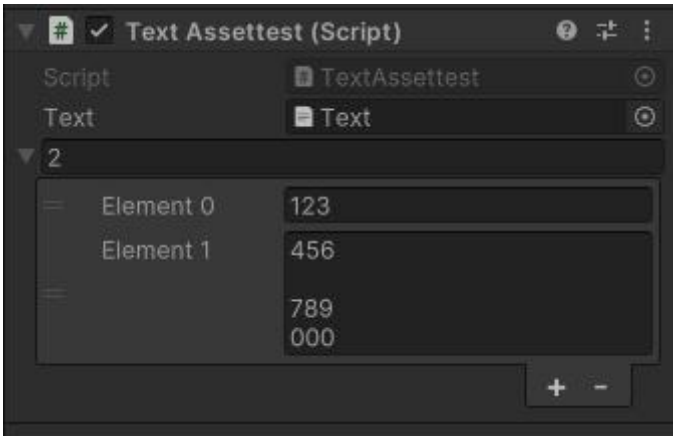

图 33.　设置 Splite 方法的最大分割数

在进行分割时，有时分割出的子串是一个空串，这时我们可以设置是否保留这种类型的子串。一旦我们设置为 RemoveEmptyEntries，就会将这些空的子串给忽略掉；设置为 None 就保留原始的分割内容，不做剔除处理。

PS：

```
string tmp = text.text;
textTemporary = tmp.Split(",",100, System.StringSplitOptions.None);// 不做剔除
```

PS：

string tmp = text.text;

textTemporary = tmp.Split(",",100, System.StringSplitOptions.RemoveEmptyEntries);//剔除空串

十二. Unity 中文乱码解决方案

当我们打开一个新的 Unity 项目，为其创建一个 C#脚本然后输入一些简单的中文注释，回到 Unity 中查看时，会发现我们代码中中文字都变成了乱码。其原因在于， Visual Studio 默认保存脚本的编码格式为 GB2312，而 Unity 中默认使用 UTF-8 进行解码，所以会出现乱码。

1. 使用 Visual Studio 修改单个脚本的格式

我们可以直接在 Visual Studio 中，修改这些产生中文乱码脚本的编码格式，这需要使用到 Visual Studio 中的高级保存选项。因此，我们的第一步就是要将 Visual Studio 隐藏的高级保存选项菜单显示出来。

在 Visual Studio 中找到工具选项卡下的自定义选项。

图 34. Visual Studio 工具选项卡下的自定义选项

在出现的弹窗中选择命令选项卡，然后在右侧点击添加命令。

图 35.　自定义弹窗下点击添加命令

在添加命令弹窗界面里找到文件下的高级保存选项，然后点击确定，我们就可以看到 Visual Studio 界面上方的工具条里出现了高级保存选项。

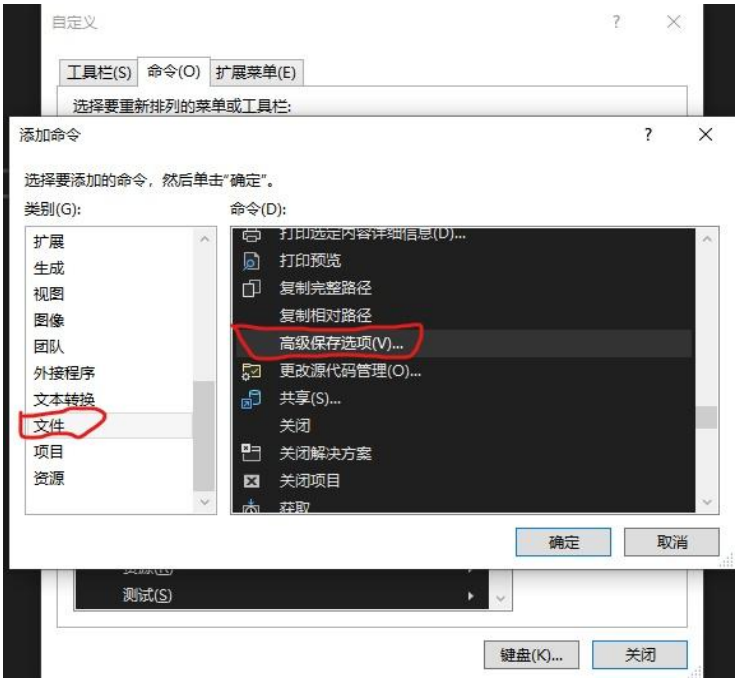

图 36.　添加命令弹窗界面里找到文件下的高级保存选项

现在我们就可以点击高级保存选项卡，去修改该脚本文件的编码格式为 UTF-8。当我们打开高级选项卡时，也可以清楚的看到当前脚本的编码格式。

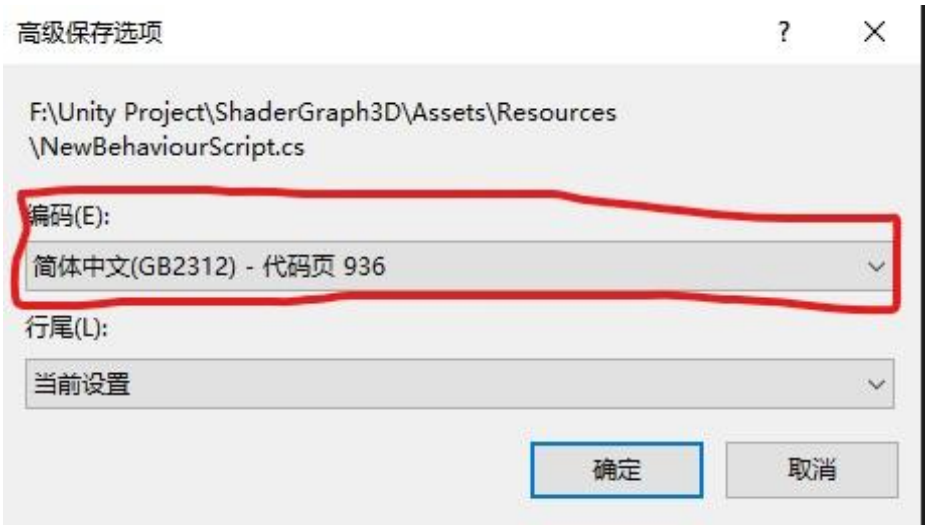

图 37.　Visual Studio 的默认编码格式

修改完成后一定要重新保存一次该脚本，只有这样才会修改脚本的编码格式。同时，这个方法仅仅只修改一个脚本，如果我们又新创建一个脚本，那么就需要重新修改新脚本的编码格式。

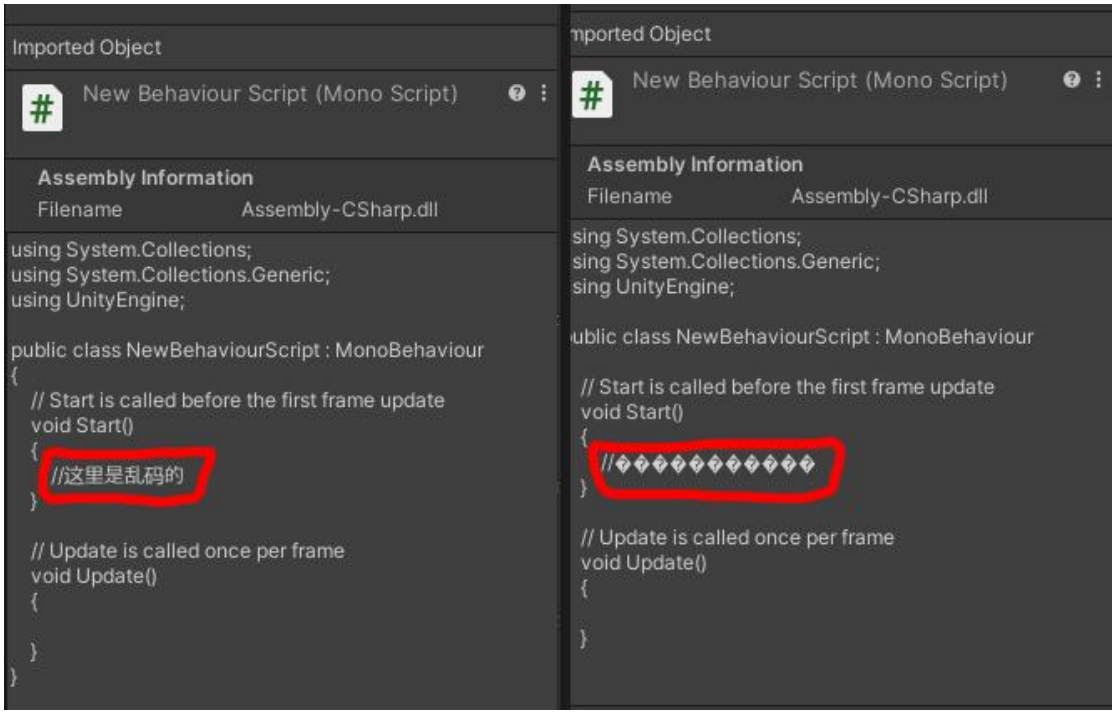

图 38.　修改文件后的中文字符预览与修改前的对比

2. 使用 Visual Studio 扩展插件

前面的方法只能修改单个脚本的编码格式，那么有没有方便一点的呢？一劳永逸的方法肯定是有的，那就是直接使用 Visual Studio 的插件。我们只需要在 Visual Studio 中，找到扩展选项卡下的管理扩展，在弹出的界面内搜索 UTF-8 即可找到对应的插件安装即可。

应用了插件以后，在保存脚本时其编码格式就会自动修改为 UTF-8，我们就无需重复进行前面的操作了。

3. 修改 Unity 脚本模板

还有一种方法也可以让新创建的 C#脚本是 UTF-8 的编码格式，那就是修改 Unity 的脚本模板。其实 Unity 脚本模板的编码格式就是 UTF-8，只是在转换为 C#脚本时发生改变，

导致我们的编码格式不再是 UTF-8，从而使中文字符发生了乱码。

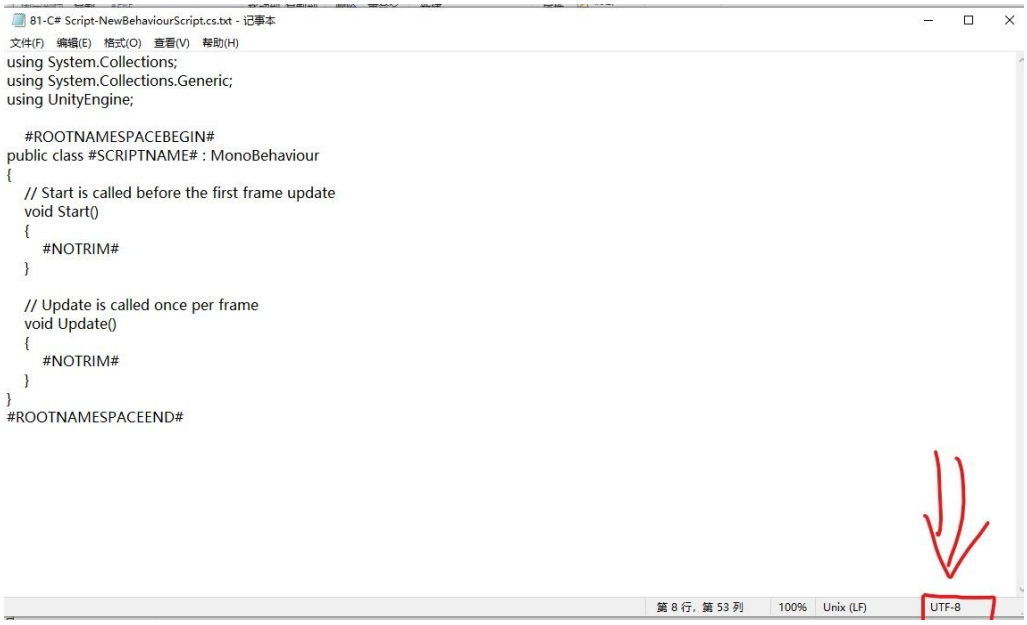

图 39.　Unity C#脚本模板的编码格式

其实我们可以在脚本模板里添加一些中文字符，来避免这一变化。也就是说，脚本模板为纯英文时，创建的新脚本编码格式就为 GB2312,；当脚本模板中含有中文字符时，新脚本的编码格式就为 UTF-8。

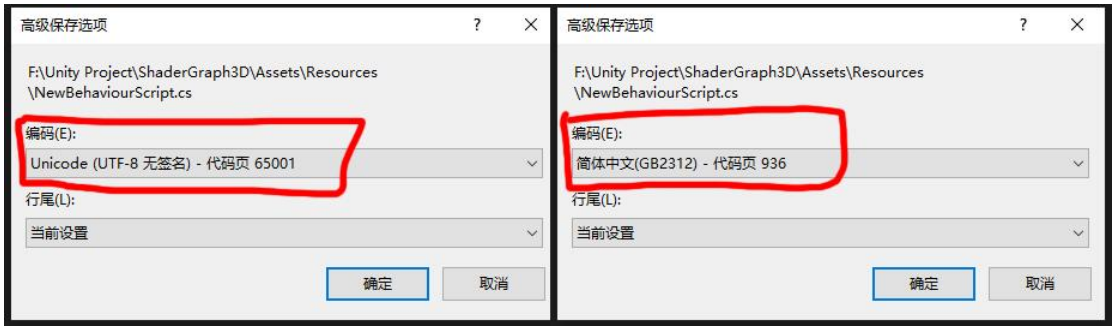

图 40.　脚本模板含中文字符与不含中文字符编码格式对比

十三.　Scene 与 Game 视图同步

1. Scene 视图同步到 Game 视图

在未运行游戏的情况下，想要将 Game 视图中主摄像机显示的内容与 Scene 视图同步，我们可以在 Hierarchy 窗口中选择主摄像机。然后，在 Unity 的工具条中找到 GameObject，选择里面的 Align With View，我们就可以将 Scene 视图的显示同步到 Game 视图了。

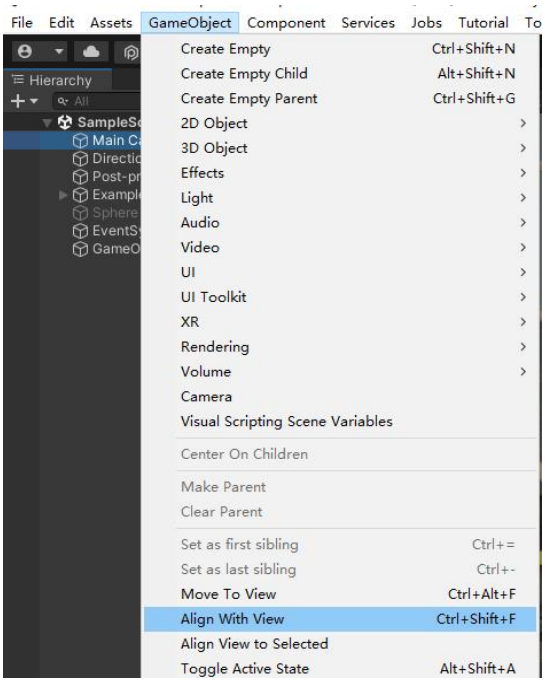

图 41.　选择主摄像机并使用 Align With View

那么有没有方便一点的呢，这肯定是有的，不过就需要我们去编写代码了。

要在运行状态下保证 Game 视图的内容和 Scene 视图同步，首先我们需要先引入一个命名空间，也即 UnityEditor，因为我们需要获取到 Scene 窗口后才能去设置主摄像机的位置。

因为 Game 视图的显示内容与主摄像机有关，因此我们必须在代码中声明一个 Camera 类型的变量，然后在 Awake 方法中去取得主摄像机。当我们有了主摄像机后，就可以在 Update 函数中去执行同步方法了。

在 Update 函数中获取到最近激活过的场景，也就是我们之前操作过的界面。只有当这个变量不会空时，我们才能执行后续的方法，避免在执行过程中出现报错。在将 Scene 视图同步到 Game 视图时只同步了位置和角度，其他的我不想进行同步，如有需要可自己补充。

PS：

```
cameraMain.transform.position =
SceneView.lastActiveSceneView.camera.transform.position;
                cameraMain.transform.rotation =
SceneView.lastActiveSceneView.camera.transform.rotation;
```

2. Game 视图同步到 Scene 视图

既然可以将 Scene 视图的内容同步到 Game 视图，那么我们也可以将其反过来，也就是将 Game 视图的内容同步到 Scene 视图中。为此，我们新增一个 bool 类型的变量，用这个变量去判断应该执行哪种同步方法。

在这里我们就需要将主摄像机的参数传递给 Scene 视图。由于 SceneView 的 Position 是 Rec，因此我们无法直接将主摄像机的值传递过去，在这里需要设置 SceneView 的 Pivot（锚点），也即设置在“场景”视图中绕行的中心点。

当我们执行这个方法时，需要开启主摄像机的控制脚本。

PS：

```
sceneView.cameraSettings.nearClip = cameraMain.nearClipPlane;
                sceneView.cameraSettings.fieldOfView = cameraMain.fieldOfView;
                sceneView.pivot = cameraMain.transform.position +
```

```
                    cameraMain.transform.forward * sceneView.cameraDistance;
                sceneView.rotation = cameraMain.transform.rotation;
```

3. 总代码

PS：

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

public class CameraSynchronize : MonoBehaviour
{
    public Camera cameraMain;
    [Tooltip("将 Scene 视图同步到 Game 视图")]
    public bool sceneToGame = false;

    // 加载脚本实例时调用 Awake
    private void Awake()
    {
        cameraMain = Camera.main;
    }

    // Update is called once per frame
    void Update()
    {
        SceneView sceneView = SceneView.lastActiveSceneView;//获取最近激活的场景
        if (sceneView != null)
        {
            if (sceneToGame)
            {
                cameraMain.transform.position = sceneView.camera.transform.position;
                cameraMain.transform.rotation = sceneView.camera.transform.rotation;
            }
            else
            {
                sceneView.cameraSettings.nearClip = cameraMain.nearClipPlane;
                sceneView.cameraSettings.fieldOfView = cameraMain.fieldOfView;
                sceneView.pivot = cameraMain.transform.position +
                    cameraMain.transform.forward * sceneView.cameraDistance;
                sceneView.rotation = cameraMain.transform.rotation;
            }
        }
    }
```

}

十四. C#事件管理器清空所有监听

十五. 将对象的所有组件深拷贝与粘贴

在 Unity 中，我们时常需要将一个对象或预制体上的组件拷贝到另一个对象上，虽然 Unity 在选项菜单中为我们提供了一个 Copy Component 功能。

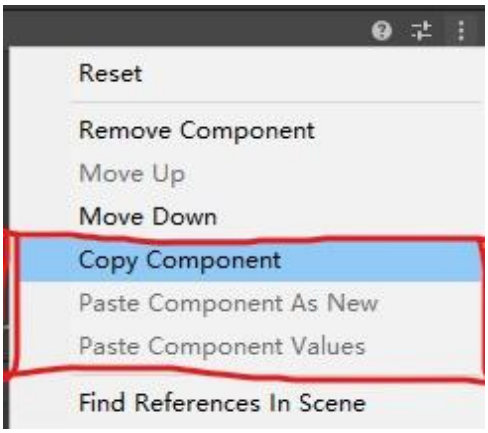

图 42. Unity 选项菜单里的组件复制

但是，当这个功能只允许我们拷贝单个组件，并不允许我们拷贝对象上的所有组件。

1. 当前层所有组件的拷贝与粘贴

1) 拷贝

就是获取当前选中的对象的组件，然后存储于变量中。

PS：

```
static Component［］copiedComponents；
［MenuItem( " GameObject/Copy Current Components ＃＆C " )］
static void Copy（）
{
copiedComponents Selection.activeGameobject.GetComponents＜Component＞()；
}
```

2) 粘贴

```
［MenuItem（ " GameObject/Paste Current Components ＃＆P " ）］
static void Paste（）
{foreach （var targetGameobject in Selection.gameobjects）

{if （targetGameobject copiedComponents ＝null）continue；

foreach （var copiedComponent in copiedComponents）
{if （copiedComponent）continue；
UnityEditorInternal.Componentutility.CopyComponent（copiedComponent）；
UnityEditorInternal.ComponentUtility.PasteComponentAsNew（targetGameObject）；
}
}
```

十六. 拖动 2D 和 3D 物体

十七. UI 坐标互转

UI 坐标之间的转换涉及层级，首先我们需要了解的到是 RectTransform 的 anchoredPosition 是通过父节点坐标计算出来的。因此，一般我们使用 WorldToScreenPoint 首先将坐标转换到屏幕坐标然后再通过 ScreenPointToLocalPointInRectangle 转换到指定 RectTransform 下的坐标。

PS：

```
#region UI 坐标互转
    /// <summary>
    ///  重新计算 UI 坐标位置
    /// </summary>
    /// <param name="from">需要转换的 UI 坐标系下的物体</param>
    /// <param name="at">转换到的最终物体</param>
    /// <param name="uiCamera">ui 摄像机</param>
    /// <param name="result">结果</param>
    public static void ReCalculateUIPosition(RectTransform from ,RectTransform at,Camera uiCamera,out Vector2 result)
    {
        //将 from 转换到屏幕坐标
        Vector2 inScreen = RectTransformUtility.WorldToScreenPoint(uiCamera, from.transform.position);
        //将屏幕坐标转换到 at 的局部坐标中
        RectTransformUtility.ScreenPointToLocalPointInRectangle(at, inScreen, uiCamera, out result);
    }
```

十八. 时间工具-TimeUtility

PS：

```
using System;
using UnityEngine;

namespace GameUtil
{
    public enum TimeConversion
    {
        Year2Day = 365,
        Year2Huor = 8766,
        Year2Minute = 525960,
        Year2Second = 31557600,
        Day2Huor = 24,
        Day2Minute = 1440,
        Day2Second = 86400,
        Huor2Minute = 60,
        Huor2Second = 3600,
```

```
        Minute2Second = 60,
    }

    /// <summary>
    /// 时间工具类
    /// </summary>
    public static class TimeUtility
    {
        #region 获取时间戳间隔
        /// <summary>
        /// 获取两个时间戳时间间隔（单位秒）
        /// </summary>
        /// <param name="timestapMax">时间戳 1(10 位-秒级)</param>
        /// <param name="timestapMin">时间戳 2(10 位-秒级)</param>
        /// <returns>返回的结果为四舍五入的</returns>
        public static int GetTwoTimeStampInterval_1(long timestapMax, long
timestapMin)
        {
            int resultTime = Mathf.RoundToInt((timestapMax - timestapMin));
            return resultTime;
        }

        /// <summary>
        /// 获取日期之间的天数
        /// </summary>
        /// <param name="startDateTime"></param>
        /// <param name="endDataTime"></param>
        /// <returns>日期之间的间隔天数</returns>
        public static int GetTwoDateTimeIntervalDay(DateTime startDateTime, DateTime
endDataTime)
        {
            //对 2008-1-20 11:44:47 使用 ToShortDateString，结果就是 2008-1-20，因
为我们只关心相差的天数
            DateTime start = Convert.ToDateTime(startDateTime.ToShortDateString());
            DateTime end = Convert.ToDateTime(endDataTime.ToShortDateString());
            TimeSpan result = end - start;
            return result.Days;
        }

        /// <summary>
        /// 获取当前时间戳（13 位）
        /// </summary>
        /// <returns>返回数值的单位是毫秒</returns>
        public static long GetCurrentUnixTimeStamp()
```

```
        {
            return (DateTime.Now.ToUniversalTime().Ticks - 621355968000000000) /
10000000;
        }
        #endregion

        #region 类型转换
        /// <summary>
        /// DateTime 转换为 long
        /// </summary>
        /// <param name="_dateTime"></param>
        /// <returns></returns>
        public static long ConvertDateTimeToLong(DateTime _dateTime)
        {
            //获取当地的格林尼治时间
            DateTime dtStart = TimeZone.CurrentTimeZone.ToLocalTime(new
DateTime(1970, 1, 1));
            //获取标准时间差
            TimeSpan toNowTime = _dateTime - dtStart;
            long timeStamp = toNowTime.Ticks;
            timeStamp = long.Parse(timeStamp.ToString().Substring(0,
timeStamp.ToString().Length - 4));
            return timeStamp;
        }

        /// <summary>
        /// long 转换为 DateTime
        /// </summary>
        /// <param name="d"></param>
        /// <returns></returns>
        public static DateTime ConvertLongToDateTime(long d)
        {
            //获取当地的格林尼治时间
            DateTime dtStart = TimeZone.CurrentTimeZone.ToLocalTime(new
DateTime(1970, 1, 1));
            long lTime = long.Parse(d + "0000");
            TimeSpan toNowTime = new TimeSpan(lTime);
            DateTime dtResult = dtStart + toNowTime;
            return dtResult;
        }

        /// <summary>
        /// DateTime 转换为 Unix 时间戳
        /// </summary>
```

```
        /// <param name="dateTime"></param>
        /// <param name="isSeconds">是否是秒级时间戳，false 就是毫秒级</param>
        /// <returns></returns>
        public static long DateTimeToUnix(DateTime dateTime,bool isSeconds = true)
        {
            // 当地时区时间
            DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1));
            // 相差秒数
            long timeStamp = isSeconds ? (long)(dateTime - startTime).TotalSeconds :
(long)(dateTime - startTime).TotalMilliseconds;
            return timeStamp;
        }

        /// <summary>
        /// Unix 时间戳转换为 DateTime
        /// </summary>
        /// <param name="unixTimeStamp">Unix 时间戳（10 位-秒级）</param>
        /// <returns></returns>
        public static string UnixToDateTime(long unixTimeStamp)
        {
            // 当地时区
            DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new
System.DateTime(1970, 1, 1));
            DateTime dt = startTime.AddSeconds(unixTimeStamp);
            return dt.ToString("yyyy/MM/dd HH:mm:ss:ffff");
        }
        #endregion

        #region 显示时间格式转换
        /// <summary>
        /// 计时时间转 00:00 格式
        /// </summary>
        /// <param name="time">转换时间</param>
        /// <param name="showHour">显示小时</param>
        /// <returns></returns>
        public static string SecondToString_1(int time, bool showHour = false)
        {
            if (showHour)
            {
                return (time / 3600 > 9 ? "" : "0") + time / 3600 + ":" + ((time % 3600) /
60 > 9 ? "" : "0") +
                                (time % 3600) / 60 + ":" + (time % 60 > 9 ? "" : "0") + time %
60;
```

```
                }
                else
                {
                    return (time / 60 > 9 ? "" : "0") + time / 60 + ":" + (time % 60 > 9 ? "" :
"0") + time % 60;
                }
            }

            public static string SecondToString_2(int time)
            {

                return "";
            }
            #endregion

            #region  时间戳检查相关
            /// <summary>
            ///  两个日期是否是同一天
            /// </summary>
            /// <param name="t1"></param>
            /// <param name="t2"></param>
            /// <returns></returns>
            public static bool IsSameDay(string DateTime1, string DateTime2)
            {
                DateTime t1 = Convert.ToDateTime(DateTime1);
                DateTime t2 = Convert.ToDateTime(DateTime2);
                return (t1.Year == t2.Year && t1.Month == t2.Month && t1.Day == t2.Day);
            }
            #endregion
        }
    }
```

十九. 其他工具-OtherUtility

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
//使用正则表达式匹配需引用的命名空间
using System.Text.RegularExpressions;

namespace GameUtil
{
    public static class OtherUtility
    {
```

```
#region Property
/// <summary>
///  十万
/// </summary>
private const int Hundred_Thousand = 100000;
/// <summary>
///  亿
/// </summary>
private const int Hundred_Million = 100000000;

#endregion

#region  获得数值的个位、十位、百位数值
/// <summary>
///  获取个位数值
/// </summary>
/// <param name="number"></param>
public static int ReturnSplitUnit(int number)
{
    return number % 10;
}

public static int ReturnSplitTen(int number)
{
    return Mathf.FloorToInt(number * 0.1f) % 10;
}

public static int ReturnSplitHundred(int number)
{
    return Mathf.FloorToInt(number * 0.01f) % 10;
}

public static int ReturnSplitThousand(int number)
{
    return Mathf.FloorToInt(number * 0.001f) % 10;
}

public static void ReturnSplitToTen(int number, out int unit, out int ten)
{
    unit = number % 10;
    //Mathf.FloorToInt  向下取整
    ten = Mathf.FloorToInt(number * 0.1f) % 10;
}
```

```
        public static void ReturnSplitToHundred(int number, out int unit, out int ten, out int
hundred)
        {
            unit = number % 10;
            ten = Mathf.FloorToInt(number * 0.1f) % 10;
            hundred = Mathf.FloorToInt(number * 0.01f) % 10;
        }

        public static void ReturnSplitToThousand(int number, out int unit, out int ten, out
int hundred, out int thousand)
        {
            unit = number % 10;
            ten = Mathf.FloorToInt(number * 0.1f) % 10;
            hundred = Mathf.FloorToInt(number * 0.01f) % 10;
            thousand = Mathf.FloorToInt(number * 0.001f) % 10;
        }
        #endregion

        #region 数值转换成字符串
        /// <summary>
        /// 将大数值转换为带有对应单位的字符串
        /// </summary>
        /// <param name="number">数值，不一定为整型，且 long / int = long /
long</param>
        /// <returns></returns>
        public static string BigNumberToUnitString(double number)
        {
            if (number >= Hundred_Million)
            {
                return $"{GetPreciseDecimal((float)(number / Hundred_Million), 2)}亿
";
            }
            else if (number >= Hundred_Thousand)
            {
                //后面所接文字，后续可修改为多语言
                return $"{GetPreciseDecimal((float)(number / Hundred_Thousand), 2) *
10}万";
            }
            else
            {
                return number.ToString();
            }
        }
```

```
/// <summary>
/// 对应数值保留几位小数,float 精度大约为 6-9 位数字，double 精度大约 15-17
位数字
/// </summary>
/// <param name="number">原始数值</param>
/// <param name="decimalPlaces">保留小数位数</param>
/// <returns></returns>
public static float GetPreciseDecimal(float number, int decimalPlaces = 0)
{
    if (decimalPlaces < 0)
        decimalPlaces = 0;

    int powerNumber = (int)Mathf.Pow(10, decimalPlaces);
    float tmeporary = number * powerNumber;
    return (float)Math.Round(tmeporary / powerNumber, decimalPlaces);
}

/// <summary>
/// 对应数值保留几位小数,float 精度大约为 6-9 位数字，double 精度大约 15-17
位数字
/// </summary>
/// <param name="number">原始数值</param>
/// <param name="decimalPlaces">保留小数位数</param>
/// <returns></returns>
public static double GetPreciseDecimal(double number, int decimalPlaces = 0)
{
    if (decimalPlaces < 0)
        decimalPlaces = 0;

    int powerNumber = (int)Mathf.Pow(10, decimalPlaces);
    double tmeporary = number * powerNumber;
    //Math.Round，参数二将双精度浮点值舍入到指定数量的小数位数。
    return Math.Round(tmeporary / powerNumber, decimalPlaces);
}

/// <summary>
/// 数字 0-9 转换为中文数字
/// </summary>
/// <param name="num"></param>
/// <returns>中文大写数值</returns>
public static string SingleDigitsNumberToChinese(int num = 0)
{
    num = Mathf.Clamp(num, 0, 9);
    //切分出字符数组
```

```
            string[] unitAllString = "零,一,二,三,四,五,六,七,八,九".Split(',');
            return unitAllString[num];
        }

        /// <summary>
        ///  数字转换为中文，可以适用任何 3 位及其以下数值
        /// </summary>
        /// <param name="number"></param>
        /// <returns></returns>
        public static string NumberToCNString(int number)
        {
            string numStr = number.ToString();
            string resultStr = "";
            int strLength = numStr.Length;
            //切分出字符数组
            string[] unitAllString = GetSpliteStringFormat("零,一,二,三,四,五,六,七,八,
九");
            string units = "", tens = "", hundreds = "";
            string tenStr = "十";
            string hundredStr = "百";

            for (int i = 1; i <= strLength; i++)
            {
                //Substring 第一个参数为从第几个字符索引位置开始截取， 参数二
为截取的长度
                int sNum = Convert.ToInt32(numStr.Substring(i - 1, 1));
                string cnStr = unitAllString[sNum];
                if (i == 1)
                {
                    units = cnStr;
                    resultStr = cnStr;
                }
                else if (i == 2)
                {
                    tens = cnStr;
                    //判断十位是否是 0
                    if (tens == unitAllString[0])
                    {
                        if (units == unitAllString[1])
                            resultStr = tenStr;
                        else
                            //例如二十，三十
                            resultStr = units + tenStr;
                    }
```

```
                else
                {
                    if (units == unitAllString[1])
                        resultStr = tenStr + tens;
                    else
                        resultStr = units + tenStr + tens;
                }
            }
            else if (i == 3)
            {
                hundreds = cnStr;
                //判断百位是否是 0
                if (hundreds == unitAllString[0])
                {
                    if (tens.Equals(unitAllString[0]))
                        resultStr = units + hundredStr;
                    else
                        //例如一百一，二百一
                        resultStr = units + hundredStr + tens;
                }
                else if (tens == unitAllString[0])
                    resultStr = units + hundredStr + tens + hundreds;
                else
                    resultStr = units + hundredStr + tens + tenStr + hundreds;
            }
        }
        return resultStr;
    }

    /// <summary>
    ///  数值转成英文序号格式（小于 100 的数值）
    /// </summary>
    /// <param name="num"></param>
    /// <returns></returns>
    public static string NumberToSequence(long num = 1)
    {
        string temporaryStr = "";
        if (num > 100)
            return temporaryStr;
        if (num % 10 == 1)
            temporaryStr = "ST";
        else if (num % 10 == 2)
            temporaryStr = "ND";
        else if (num % 10 == 3)
```

```
                temporaryStr = "RD";
            else
                temporaryStr = "TH";
            return num.ToString() + temporaryStr;
        }
        #endregion

        /// <summary>
        ///  设置字符串颜色
        /// </summary>
        /// <param name="inputStr">我</param>
        /// <param name="inputColor">十六进制颜色</param>
        /// <returns></returns>
        public static string SetTextColor(string inputStr, string inputColor = "FFFFFF")
        {
            if (IsBase16ColorFormat(inputColor))
                return $"<color=#{inputColor}>{inputStr}</color>";
            else
                Debug.LogWarning("SetTextColor 不是使用的十六进制颜色");
            return "";
        }

        #region  正则表达式检测相关
        #region  数值相关
        /// <summary>
        ///  是否是正整数（不包含 0）
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsPositiveInteger(string inputStr)
        {
            //这里是我们使用的匹配规则，使用@（逐字字符串标识符）将原义解释
为字符串
            Regex reg = new Regex(@"^\+?[1-9]\d*$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        ///  是否是负整数（不包含 0）
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsNegateInteger(string inputStr)
        {
```

```
        //这里是我们使用的匹配规则
        Regex reg = new Regex(@"^-[1-9]\d*$");
        return reg.IsMatch(inputStr);
    }

    /// <summary>
    ///  是否是整数
    /// </summary>
    /// <param name="inputStr"></param>
    /// <returns></returns>
    public static bool IsInteger(string inputStr)
    {
        //这里是我们使用的匹配规则
        Regex reg = new Regex(@"^(?:0|(?:-?[1-9]\d*))$");
        return reg.IsMatch(inputStr);
    }

    /// <summary>
    ///  是否是浮点数
    /// </summary>
    /// <param name="inputStr">例如 1.01</param>
    /// <returns></returns>
    public static bool IsFloat(string inputStr)
    {
        //这里是我们使用的匹配规则
        Regex reg = new Regex(@"^(-?[1-9]\d*\.\d+|-?0\.\d*[1-9]\d*|0\.0+)$");
        return reg.IsMatch(inputStr);
    }

    /// <summary>
    ///  是否是浮点数（严格）
    /// </summary>
    /// <param name="inputStr">例如 1.01</param>
    /// <returns></returns>
    public static bool IsStrictlyFloat(string inputStr)
    {
        //这里是我们使用的匹配规则
        Regex reg = new Regex(@"^(-?[1-9]\d*\.\d+|-?0\.\d*[1-9]\d*|0\.0+)$");
        return reg.IsMatch(inputStr);
    }

    /// <summary>
    ///  是否只包含数字
    /// </summary>
```

```
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsOnlyNumber(string inputStr)
        {
            Regex reg = new Regex(@"^\d+$");
            return reg.IsMatch(inputStr);
        }
        #endregion

        #region 计算机网络相关
        /// <summary>
        /// 是否是电子邮箱
        /// </summary>
        /// <param name="inputStr">例如 10086@qq.com</param>
        /// <returns></returns>
        public static bool IsEmail(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new
Regex(@"^[A-Za-z0-9\u4e00-\u9fa5]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是计算机域名（非网址, 不包含协议）
        /// </summary>
        /// <param name="inputStr">例如 www.baidu.com</param>
        /// <returns></returns>
        public static bool IsInternetDomainName(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^([0-9a-zA-Z-]{1,}\.)+([a-zA-Z]{2,})$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 必须是带端口号的网址(或 ip)
        /// </summary>
        /// <param name="inputStr">例如 https://www.qq.com:8080</param>
        /// <returns></returns>
        public static bool IsIpAddressHavePort(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^((ht|f)tps?:\/\/)?[\w-]+(\.[\w-]+)+:\d{1,5}\/?$");
```

```
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        ///  是否是统一资源标识符
        /// </summary>
        /// <param name="inputStr">例如 www.qq.com</param>
        /// <returns></returns>
        public static bool IsURL(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new
Regex(@"^(((ht|f)tps?):\/\/)?([^!@#$%^&*?.\s-]([^!@#$%^&*?.\s]{0,63}[^!@#$%^&*?.\s])?\.)+
[a-z]{2,6}\/?");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        ///  是否是子网掩码（不包含 0.0.0.0）
        /// </summary>
        /// <param name="inputStr">例如 255.255.255.0</param>
        /// <returns></returns>
        public static bool IsSubNetMask(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new
Regex(@"^(254|252|248|240|224|192|128)\.0\.0\.0|255\.(254|252|248|240|224|192|128|0)\.0\.0|255\.
255\.(254|252|248|240|224|192|128|0)\.0|255\.255\.255\.(255|254|252|248|240|224|192|128|0)$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        ///  是否是迅雷链接
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsThunderURL(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^thunderx?:\/\/[a-zA-Z\d]+=$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
```

```
        /// 是否是磁力链接
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsMagneticLinkURL(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^magnet:\?xt=urn:btih:[0-9a-fA-F]{40,}.*$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是 IPV4 端口
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsIPV4Port(string inputStr)
        {
            Regex reg = new
Regex(@"^((\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])(?::(?:[0-9]|[1
-9][0-9]{1,3}|[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5]))?$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是 Mac 地址
        /// </summary>
        /// <param name="inputStr">例如：38:f9:d3:4b:f5:51 或
00-0C-29-CA-E4-66</param>
        /// <returns></returns>
        public static bool IsMacAddress(string inputStr)
        {
            Regex reg = new
Regex(@"^((([a-f0-9]{2}:){5})|(([a-f0-9]{2}-){5}))[a-f0-9]{2}$");
            return reg.IsMatch(inputStr);
        }
        #endregion

        #region 输入检测、密码检测相关
        /// <summary>
        /// 是否是 md5 格式（32 位）
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
```

```
public static bool CheckMD5Format(string inputStr)
{
    Regex reg = new Regex(@"^[a-fA-F0-9]{32}$");
    return reg.IsMatch(inputStr);
}

/// <summary>
/// QQ 号格式是否正确
/// </summary>
/// <param name="inputStr">例如 1210420078</param>
/// <returns></returns>
public static bool CheckQQFormat(string inputStr)
{
    Regex reg = new Regex(@"^[1-9][0-9]{4,10}$");
    return reg.IsMatch(inputStr);
}

/// <summary>
/// 账号是否合法（字母开头，允许 5-16 字节，允许字母数字下划线组合）
/// </summary>
/// <param name="inputStr"></param>
/// <returns></returns>
public static bool CheckAccountIsLegitimate(string inputStr)
{
    Regex reg = new Regex(@"^[a-zA-Z]\w{4,15}$");
    return reg.IsMatch(inputStr);
}

/// <summary>
/// 是否是数字和字母组成
/// </summary>
/// <param name="inputStr"></param>
/// <returns></returns>
public static bool IsNumberAndLetterStruct(string inputStr)
{
    Regex reg = new Regex("^[A-Za-z0-9]+$");
    return reg.IsMatch(inputStr);
}

/// <summary>
/// 是否是小写英文字母组成
/// </summary>
/// <param name="inputStr">例如 russel</param>
/// <returns></returns>
```

```
        public static bool IsLowercaseLetterStruct(string inputStr)
        {
            Regex reg = new Regex("^[a-z]+$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否由大写英文字母组成
        /// </summary>
        /// <param name="inputStr">例如 ABC</param>
        /// <returns></returns>
        public static bool IsUppercaseLetterStruct(string inputStr)
        {
            Regex reg = new Regex("^[A-Z]+$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 密码强度校验，最少 6 位，包括至少 1 个大写字母，1 个小写字母，1 个数字，1 个特殊字符
        /// </summary>
        /// <param name="inputStr">例如 Zc@bilibili66666</param>
        /// <returns></returns>
        public static bool CheckPasswordStrength(string inputStr)
        {
            Regex reg = new
Regex(@"^\S*(?=\S{6,})(?=\S*\d)(?=\S*[A-Z])(?=\S*[a-z])(?=\S*[!@#$%^&*? ])\S*$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否由中文和数字组成
        /// </summary>
        /// <param name="inputStr">例如：你好 6 啊</param>
        /// <returns></returns>
        public static bool IsChineseAndNumberStruct(string inputStr)
        {
            Regex reg = new
Regex(@"^((?:[\u3400-\u4DB5\u4E00-\u9FEA\uFA0E\uFA0F\uFA11\uFA13\uFA14\uFA1F\uFA21\uFA23\uFA24\uFA27-\uFA29]|[\uD840-\uD868\uD86A-\uD86C\uD86F-\uD872\uD874-\uD879][\uDC00-\uDFFF]|\uD869[\uDC00-\uDED6\uDF00-\uDFFF]|\uD86D[\uDC00-\uDF34\uDF40-\uDFFF]|\uD86E[\uDC00-\uDC1D\uDC20-\uDFFF]|\uD873[\uDC00-\uDEA1\uDEB0-\uDFFF]|\uD87A[\uDC00-\uDFE0])|(\d))+$");
            return reg.IsMatch(inputStr);
```

```
        }

        /// <summary>
        ///  是否不包含字母
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsWithoutAlphabet(string inputStr)
        {
            Regex reg = new Regex("^[^A-Za-z]*$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        ///  大写字母，小写字母，数字，特殊符号 `@#$%^&*`~()-+=` 中任意 3 项密码
        /// </summary>
        /// <param name="inputStr">例如：a1@,A1@</param>
        /// <returns></returns>
        public static bool CheckPasswordHaveHowManyType(string inputStr)
        {
            Regex reg = new Regex(@"^(?![a-zA-Z]+$)(?![A-Z0-9]+$)(?![A-Z\W_!@#$%^&*`~()-+=]+$)(?![a-z0-9]+$)(?![a-z\W_!@#$%^&*`~()-+=]+$)(?![0-9\W_!@#$%^&*`~()-+=]+$)[a-zA-Z0-9\W_!@#$%^&*`~()-+=]");
            return reg.IsMatch(inputStr);
        }
        #endregion

        #region  身份证检测、国家及地区检测
        /// <summary>
        ///  是否是中国的省份
        /// </summary>
        /// <param name="inputStr">例如浙江、台湾</param>
        /// <returns></returns>
        public static bool IsChineseProvinces(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex("^浙江|上海|北京|天津|重庆|黑龙江|吉林|辽宁|内蒙古|河北|新疆|甘肃|青海|陕西|宁夏|河南|山东|山西|安徽|湖北|湖南|江苏|四川|贵州|云南|广西|西藏|江西|广东|福建|台湾|海南|香港|澳门$");
            return reg.IsMatch(inputStr);
        }
```

```
        /// <summary>
        /// 是否是身份证号（1 代，15 位数字）
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsIdentityCard_1(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new
Regex(@"^[1-9]\d{7}(?:0\d|10|11|12)(?:0[1-9]|[1-2][\d]|30|31)\d{3}$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是身份证号（2 代，18 位数字，最后一位是校验位,可能为数字或字符
X）
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsIdentityCard_2(string inputStr)
        {
            Regex reg = new
Regex(@"^[1-9]\d{5}(?:18|19|20)\d{2}(?:0[1-9]|10|11|12)(?:0[1-9]|[1-2]\d|30|31)\d{3}[\dXx]$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是香港身份证
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsHongkongIdentityCard(string inputStr)
        {
            Regex reg = new Regex(@"^[a-zA-Z]\d{6}\([\dA]\)$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是澳门身份证
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static bool IsMacauIdentityCard(string inputStr)
        {
```

```
            Regex reg = new Regex(@"^[1|5|7]\d{6}\(\d\)$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是台湾身份证
        /// </summary>
        /// <param name="inputStr">例如：U193683453</param>
        /// <returns></returns>
        public static bool IsTaiwanIdentityCard(string inputStr)
        {
            Regex reg = new Regex(@"^[a-zA-Z][0-9]{9}$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是中国的邮政编码
        /// </summary>
        /// <param name="inputStr">例如 100101</param>
        /// <returns></returns>
        public static bool IsChinesePostalCode(string inputStr)
        {
            Regex reg = new
Regex(@"^(0[1-7]|1[0-356]|2[0-7]|3[0-6]|4[0-7]|5[1-7]|6[1-7]|7[0-5]|8[013-6])\d{4}$");
            return reg.IsMatch(inputStr);
        }
        #endregion

        #region 字符串拆分与替换
        /// <summary>
        /// 按照,#+|，将字符串进行拆分
        /// </summary>
        /// <param name="inputStr"></param>
        /// <returns></returns>
        public static string[] GetSpliteStringFormat(string inputStr)
        {
            Regex reg = new Regex("[,#+|]");
            return reg.Split(inputStr);
        }

        /// <summary>
        /// 用特定格式替换字符串中的，所有空格、换行符、新行、tab 字符
        /// </summary>
        /// <param name="inputStr"></param>
```

```
        /// <param name="replaceStr">要替换成的字符</param>
        /// <returns></returns>
        public static string ReplaceSpaceByFormat(string inputStr, string replaceStr = "")
        {
            Regex reg = new Regex(@"\s");
            return reg.Replace(inputStr, replaceStr);
        }

        /// <summary>
        /// 仅以指定格式替换左右两端的空格，string 里的 Trim 也可删除左右两端空格
        /// </summary>
        /// <param name="inputStr"></param>
        /// <param name="replaceStr">要替换成的字符</param>
        /// <returns></returns>
        public static string ReplaceTrimSpaceByFormat(string inputStr, string replaceStr = "")
        {
            Regex reg = new Regex(@"(^\s*)|(\s*$)");
            return reg.Replace(inputStr, replaceStr);
        }

        /// <summary>
        /// 仅以指定格式替换左端的空格，string 里的 TrimStart 也可删除左端空格
        /// </summary>
        /// <param name="inputStr"></param>
        /// <param name="replaceStr">要替换成的字符</param>
        /// <returns></returns>
        public static string ReplaceTrimStartByFormat(string inputStr, string replaceStr = "")
        {
            Regex reg = new Regex(@"(^\s*)");
            return reg.Replace(inputStr, replaceStr);
        }

        /// <summary>
        /// 仅以指定格式替换右端的空格，string 里的 TrimEnd 也可删除右端空格
        /// </summary>
        /// <param name="inputStr"></param>
        /// <param name="replaceStr">要替换成的字符</param>
        /// <returns></returns>
        public static string ReplaceTrimEndByFormat(string inputStr, string replaceStr = "")
        {
```

```
            Regex reg = new Regex(@"(\s*$)");
            return reg.Replace(inputStr, replaceStr);
        }
        #endregion

        #region 其他
        /// <summary>
        /// 版本号(version)格式必须为 X.Y.Z
        /// </summary>
        /// <param name="inputStr">例如 16.3.10</param>
        /// <returns></returns>
        public static bool CheckVersionForamt(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^\d+(?:\.\d+){2}$");
            return reg.IsMatch(inputStr);
        }

        /// <summary>
        /// 是否是 16 进制颜色格式
        /// </summary>
        /// <param name="inputStr">例如#f00,#000,#fe9de8</param>
        /// <returns></returns>
        public static bool IsBase16ColorFormat(string inputStr)
        {
            //这里是我们使用的匹配规则
            Regex reg = new Regex(@"^#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3})$");
            return reg.IsMatch(inputStr);
        }
        #endregion

        #endregion
    }
}
```

二十. 资源路径工具

```
#region 资源路径相关
    /// <summary>
    /// 获取指定 Asset 资源全路径
    /// </summary>
    /// <param name="obj">Asset 资源</param>
    /// <returns>全路径</returns>
    public static string GetAseetFullPath(Object obj)
    {
        if (obj == null)
```

```
            {
                return string.Empty;
            }
            return AssetPathToFullPath(AssetDatabase.GetAssetOrScenePath(obj));
        }

        /// <summary>
        ///  获取指定 Asset 资源所在的文件夹名字
        /// </summary>
        /// <param name="obj">Asset 资源</param>
        /// <returns>文件夹名字</returns>
        public static string GetAssetFloderName(Object obj)
        {
            if (obj == null)
            {
                return string.Empty;
            }
            var path = FileUtils.GetFloderName(GetAseetFullPath(obj));
            return path;
        }

        /// <summary>
        /// Asset 内相对路径转换为全路径
        /// </summary>
        /// <param name="assetPath">Asset 内相对路径</param>
        /// <returns>全路径</returns>
        public static string AssetPathToFullPath(string assetPath)
        {
            return FileUtils.FormatToUnityPath(assetPath).Replace("Assets",
Application.dataPath);
        }

        /// <summary>
        ///  全路径转换为 Asset 内相对路径
        /// </summary>
        /// <param name="fullPath">全路径</param>
        /// <returns>Asset 内相对路径</returns>
        public static string FullPathToAssetPath(string fullPath)
        {
            return FileUtils.FormatToUnityPath(fullPath).Replace(Application.dataPath,
"Assets");
        }

        public static string[] GetFilesPaths(string dirPath, string searchPattern = "*",
```

```
SearchOption searchOption = SearchOption.AllDirectories)
        {
            var paths = new List<string>();
            var files = FileUtils.GetFiles(dirPath, searchPattern, searchOption);
            string tempStr;
            if (files != null && files.Length > 0)
            {
                for (int j = 0; j < files.Length; j++)
                {
                    tempStr = FullPathToAssetPath(files[j].FullName);
                    paths.Add(tempStr);
                }
            }
            return paths.ToArray();
        }

        /// <summary>
        /// string 转 byte[]
        /// </summary>
        /// <param name="str"></param>
        /// <returns></returns>
        public static byte[] FromHexString(string str)
        {
            string[] byteStrings = str.Split(",".ToCharArray());
            byte[] byteOut = new byte[byteStrings.Length];
            for (int i = 0; i < byteStrings.Length; i++)
            {
                byteOut[i] = byte.Parse(byteStrings[i]);
            }
            return byteOut;
        }
        #endregion
```

二十一. 使用 RenderTexture 在 UI 显示和旋转模型

在游戏开发中，我们有时会遇到一种需求，就是将模型渲染到 UI 面板上，并使其可以拖拽旋转。要让模型显示到 UI 面板上，我们就需要使用 RenderTexture。

1. 创建 RenderTexture 并让其在 UI 面板上显示

要创建一个 RenderTexture，我们只需要在 Unity 中点击鼠标右键，在 Create 里找到 RenderTexture 并点击它。

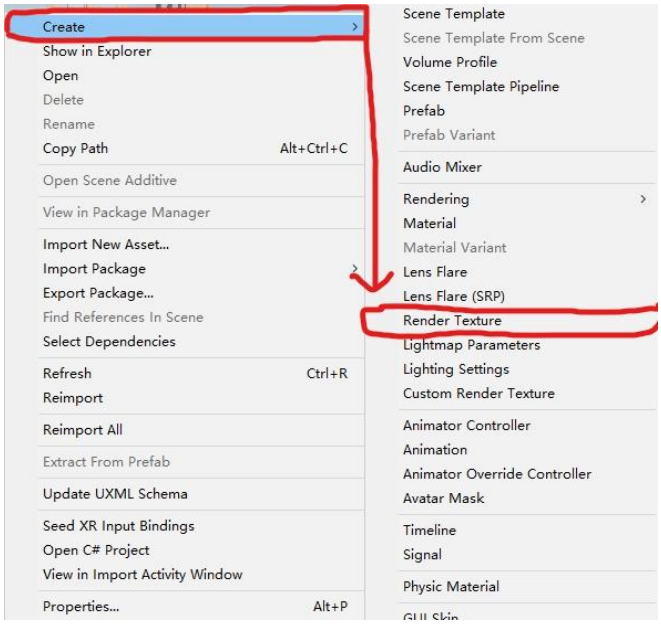

图 43.　　创建 RenderTexture

我们创建了一个 RenderTexture 后，还需要创建对应的摄像机去捕捉我们的画面。这个摄像机会将它所看到的场景，输出到我们所创建的 RenderTexture 上。一旦我们的 3D 场景被渲染输出到 RenderTexture 上，那么它就可以被当做图片进行使用了。

在创建这个摄像机之前，我们先创建用于显示的 UI 面板。

在 Unity 的 Hierarchy 窗口中点击鼠标右键，在 UI 选项中找到 RawImage 并点击它。因为此时我们并没有在 Canvas 中创建它，因此我们创建 RawImage 的同时会自动构建一个 Canvas 作为其父节点。

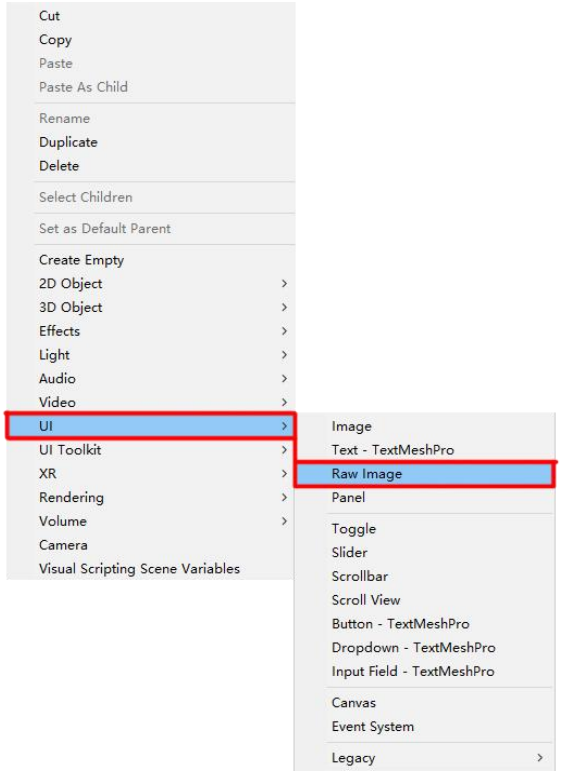

图 44.　　创建一个 RawImage 作为背景

我们选择一张背景图片，赋予给刚刚创建出来的 RawImage。然后在这个 Raw Image 下

再新建一个 Raw Image，然后将之前创建的 RenderTexture 赋予给它。

现在已经完成了基础的构建，下面就是使用 Camera 去捕捉模型，并将捕获的画面输出到 RenderTexture 上。

创建的 Camera 中有一些选项需要注意一下，尤其是 Camera 的 Culling Mask。

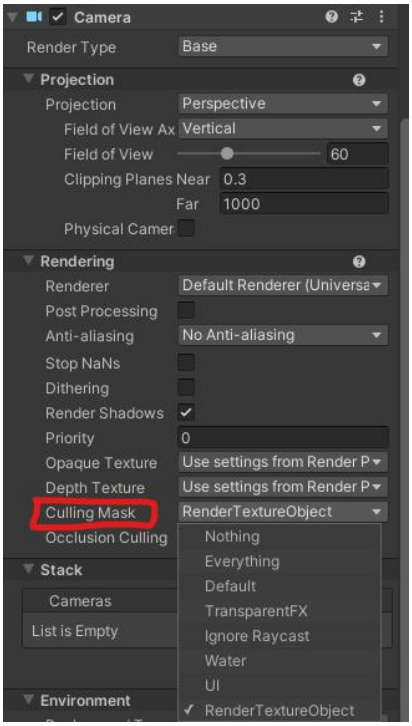

图 45.　设置 Camera 的 Culling Mask

只有当对象的 Layer 包含在 Culling Mask 勾选的菜单项里，这个对象才会被渲染到 Camera 里。在这里，我们只希望摄像机捕捉模型，为此我们需要单独添加一个层级将其用于要渲染到 RenderTexture 的对象上。

摄像机添加完成后，我们就可以将模型拖拽进场景了。为了使 Game 视图的输出和 Scene 视图的统一，我们可以在 Scene 视图中先移动到合适的位置，然后选中摄像机并按下 Ctrl+Shift+F，就可以快速的将摄像机移动到现有位置。

最后一步，我们将 Camera 的 OutPut Texture 设置为我们之前创建的 RenderTexture，这样我们就可以将画面输出到 RenderTexture 中了。

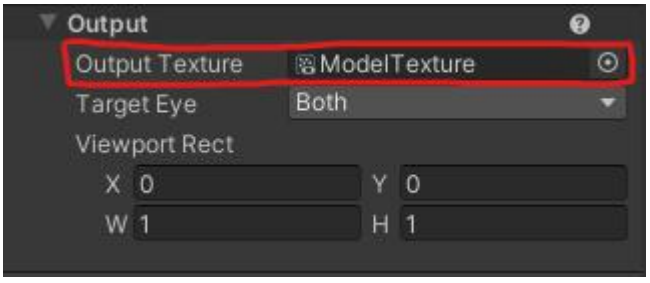

图 46.　设置摄像机的输出纹理

2. 使用按钮旋转模型

我们继续在画布中添加两个新的 Image 并为其赋予纹理，这两个 Image 就是我们向左和向右旋转的按钮。

完成后，我们便创建一个新的脚本用来执行交互的逻辑。这个脚本里的逻辑很简单，就是按下旋转的按钮时，模型就一直进行旋转，直到鼠标抬起才停止旋转。

在脚本中先创建好所需的变量，并在 Start 函数中检查那些外部赋值的变量。这里主要是检查是否已经赋予了模型节点，如果没有模型的话我们就无法执行旋转逻辑。

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;

public class ModelShowController :
MonoBehaviour,IDragHandler,IBeginDragHandler,IEndDragHandler
{
    /// <summary>
    /// 模型节点
    /// </summary>
    public GameObject modelGameObject = null;
    public float turnSpeed = 1.0f;
    /// <summary>
    /// 当前的状态是否是点击状态
    /// </summary>
    public bool curStateIsClick = false;
    /// <summary>
    /// 当前的欧拉角 Y 值
    /// </summary>
    private float currentEulerY = 0;
    /// <summary>
    /// 是否是向右旋转，主要用于按钮点击
    /// </summary>
    private bool isRitght = false;
    /// <summary>
    /// 输出的欧拉角
    /// </summary>
    private Vector3 outputAngle = Vector3.zero;

    private void Start()
    {
        if (modelGameObject == null)
        {
            //在场景中查找 RenderMondelRoot
            modelGameObject = GameObject.Find("RenderModelRoot");
            if (modelGameObject != null)
                currentEulerY = modelGameObject.transform.eulerAngles.y;
            else
                Debug.LogError("无法找到 RenderMondelRoot");
        }
```

```
        }
    }
```

接着我们创建用于旋转模型的协程。因为是想要让模型一直旋转下去的，所以 While 里的判断条件就为 true。在里面我们根据旋转方向的不同，去增减旋转角度。

PS：

```
    /// <summary>
    ///  旋转模型的协程
    /// </summary>
    /// <returns></returns>
    private IEnumerator TurnRightModel()
    {

        while (true)
        {
            //增减旋转角度
            if (isRitght)
                currentEulerY -= turnSpeed;
            else
                currentEulerY += turnSpeed;
            outputAngle.y = currentEulerY;
            modelGameObject.transform.eulerAngles = outputAngle;
            yield return null;
        }
    }
```

最后，我们再创建一个公共方法。这个公共方法将绑定到 UI 按钮上，在点击按钮和结束按钮的行为发生时，我们都会调用这个公共方法。

PS：

```
    /// <summary>
    ///  设置点击状态
    /// </summary>
    /// <param name="turnRitght">是否向右旋转</param>
    public void SetClickState(bool turnRitght)
    {
        curStateIsClick = !curStateIsClick;
        isRitght = turnRitght;
        if (curStateIsClick)
            StartCoroutine(TurnRightModel());
        else
            StopAllCoroutines();
    }
```

脚本编写完成后，我们就可以在 Unity 的 UI 面板中再创建一个空的对象，将我们的控制脚本挂载上去。我们的两个按钮，要想调用这个脚本中的方法，就需要挂载上 EventTrigger 组件，同时将 Point Down 事件和 Point UP 事件添加进去，最后在绑定上要调用的方法即可。

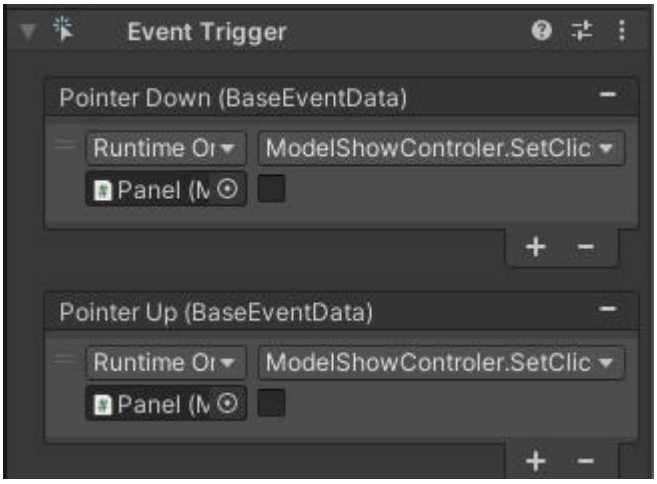

图 47.　在 EventTrigger 中调用控制脚本中的公共方法

3. 使用鼠标拖拽来旋转模型

现在，我们虽然可以使用按钮来旋转模型了，但这其实还不是非常的方便。我们还希望，能够直接使用鼠标拖拽的方式去旋转我们的模型。

我们在挂载旋转控制脚本的对象上再去新增一个 Image 组件，并将这个 Image 组件中的 Alpha 值设置为 0。

完成后再回到脚本中，准备在脚本中实现拖拽旋转的逻辑。在这里，我将使用 Unity 中的 UGUI 事件接口去完成这一功能。因为，UGUI 的事件接口是包含在 UnityEngine EventSystems 命名空间下的，我们在使用之前必须引入这一个命名空间。

PS：

```
using UnityEngine.EventSystems;
```

命名空间引入完成后，我们就可以在 MonoBehaviour 后新增三个接口了，它们分别是 IDragHandler、IBeginDragHandler 和 IEndDragHandler。因为使用的是接口，我们就必须实现接口中的所有方法，不论是显式实现还是隐式实现。

接口的方法添加完毕以后，就可以接着新增实现拖拽旋转所需的变量了。我们需要一个参数去控制位移与角度之间的转换比，一个存储上一次拖拽点位置信息的二维向量。

PS：

```
    /// 拖拽时的旋转偏移量缩放值
    /// </summary>
    public float offsetScale = 1;
    /// <summary>
    /// 开始拖拽时的起始鼠标位置
    /// </summary>
    private Vector2 startDragPoint;
```

所需的变量添加完成后，就开始正式编写逻辑代码了。

因为 OnBeginDrag 方法在 OnDrag 之前响应，我们就需要先在 OnBeginDrag 方法中，记录鼠标开始进行拖拽时的位置。接着再在 OnDrag 方法中计算 X 轴的位移量，并将此时的鼠标位置记录下来，以便用于下一次的计算。当鼠标抬起时我们执行 OnEndDrag 方法，在这个方法体内我将拖拽的初始位置设置为 0。

PS：

```
    public void OnDrag(PointerEventData eventData)
    {
```

```
            Vector2 currentDragPoint = Input.mousePosition;
            float offset = (startDragPoint.x - currentDragPoint.x) * offsetScale ;
            currentEulerY = modelGameObject.transform.localEulerAngles.y + offset;
            outputAngle.y = currentEulerY;
            //设置拖拽的起始位置为当前的鼠标位置
            startDragPoint = currentDragPoint;
            modelGameObject.transform.eulerAngles = outputAngle;
        }
        public void OnBeginDrag(PointerEventData eventData)
        {
            startDragPoint = Input.mousePosition;
        }

        public void OnEndDrag(PointerEventData eventData)
        {
            startDragPoint = Vector2.zero;
        }
```

现在，我们返回到 Unity 中，就可以通过拖拽鼠标来旋转模型了。

当然，其实我们也可以只使用 IDragHandler 接口。因为 PointerEventData 里的 delta，就已经包含了我们之前的计算，在 OnDrag 方法中可以直接使用它去增减旋转角度。

PS：

```
public void OnDrag(PointerEventData eventData)
        {
            currentEulerY -= (eventData.delta.x * offsetScale);
            outputAngle.y = currentEulerY;
            modelGameObject.transform.eulerAngles = outputAngle;
        }
```

4. 优化不可见的 Image

PS：

在 UI 中很多时候需要使用 Image 组件的 Raycast Target 来设置点击区域，但代价就是会动态生成一个 mesh。为了避免这部分的性能消耗，我们可以编写一个 EmptyGraphic 来替代 Image 组件，避免生成 mesh 网格。

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

namespace UnityEngine.UI
{
    // EmptyGraphic 是一个只在逻辑上响应 Raycast 但是不参与绘制的组件
    public class EmptyGraphic : MaskableGraphic
    {
        public override Material materialForRendering
        {
            get
```

```
            {
                return this.material;
            }
        }

        protected EmptyGraphic()
        {
            useLegacyMeshGeneration = false;
        }

        protected override void OnPopulateMesh(VertexHelper toFill)
        {
            //清除所有顶点，不进行清除的话，我们添加该组件就会出现一个面片
            toFill.Clear();
        }
    }
}
```

2019 及其以下的版本，建议那些需要响应 RayCast Target 的空 UI 内容，使用上面的脚本。（因为我只测试了 2019 以下的版本和 Unity 2022）

但是，在 2022 的版本中（具体是从哪个版本开始的我没有细查），要清除空 Image 中的 Mesh 网格就只需要将 Alpha 设置为 0 即可。因为 2022 版本的 Unity，优化了 Image 的这部分内容，使得 Alpha 为 0 时不再会生成 Mesh 网格。

5. 模型旋转控制脚本整体代码

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;

public class ModelShowController :
MonoBehaviour,IDragHandler,IBeginDragHandler,IEndDragHandler
{
    /// <summary>
    /// 模型节点
    /// </summary>
    public GameObject modelGameObject = null;
    public float turnSpeed = 1.0f;
    /// <summary>
    /// 当前的状态是否是点击状态
    /// </summary>
    public bool curStateIsClick = false;
    /// <summary>
    /// 拖拽时的旋转偏移量缩放值
    /// </summary>
```

```
public float offsetScale = 1;
/// <summary>
///  当前的欧拉角 Y 值
/// </summary>
private float currentEulerY = 0;
/// <summary>
///  是否是向右旋转，主要用于按钮点击
/// </summary>
private bool isRitght = false;
/// <summary>
///  开始拖拽时的起始鼠标位置
/// </summary>
private Vector2 startDragPoint;
/// <summary>
///  输出的欧拉角
/// </summary>
private Vector3 outputAngle = Vector3.zero;

private void Start()
{
    if (modelGameObject == null)
    {
        //在场景中查找 RenderMondelRoot
        modelGameObject = GameObject.Find("RenderModelRoot");
        if (modelGameObject != null)
            currentEulerY = modelGameObject.transform.eulerAngles.y;
        else
            Debug.LogError("无法找到 RenderMondelRoot");
    }
}

/// <summary>
///  旋转模型的协程
/// </summary>
/// <returns></returns>
private IEnumerator TurnRightModel()
{

    while (true)
    {
        //增减旋转角度
        if (isRitght)
            currentEulerY -= turnSpeed;
        else
```

```
                currentEulerY += turnSpeed;
            outputAngle.y = currentEulerY;
            modelGameObject.transform.eulerAngles = outputAngle;
            yield return null;
        }
    }

    /// <summary>
    ///  设置点击状态
    /// </summary>
    /// <param name="turnRitght">是否向右旋转</param>
    public void SetClickState(bool turnRitght)
    {
        curStateIsClick = !curStateIsClick;
        isRitght = turnRitght;
        if (curStateIsClick)
            StartCoroutine(TurnRightModel());
        else
            StopAllCoroutines();
    }

    #region  拖拽事件处理器
    public void OnDrag(PointerEventData eventData)
    {
        Vector2 currentDragPoint = Input.mousePosition;
        float offset = (startDragPoint.x - currentDragPoint.x) * offsetScale ;
        currentEulerY = modelGameObject.transform.localEulerAngles.y + offset;
        outputAngle.y = currentEulerY;
        //设置拖拽的起始位置为当前的鼠标位置
        startDragPoint = currentDragPoint;
        modelGameObject.transform.eulerAngles = outputAngle;
    }
    public void OnBeginDrag(PointerEventData eventData)
    {
        startDragPoint = Input.mousePosition;
    }

    public void OnEndDrag(PointerEventData eventData)
    {
        startDragPoint = Vector2.zero;
    }
    #endregion
}
```

二十二. Transform 扩展方法

1. 方法

1) 广度优先搜索遍历

PS：

```
public static TR BFSVisit < TP, TR > (this Transform root, System.Func <
Transform, TP, TR > visitFunc, TP para, TR failReturnValue = default(TR))
{
    TR ret = visitFunc(root, para);
    if (ret != null && !ret.Equals(failReturnValue))
        return ret;
    Queue<Transform> parents = new Queue<Transform> ();
    parents.Enqueue(root);
    while(parents.Count > 0)
    {
        Transform parent = parents.Dequeue();
        foreach(Transform child in parent)
        {
            ret = visitFunc(child, para);
            if (ret != null && !ret.Equals(failReturnValue))
                return ret;
            parents.Enqueue(child);
        }
    }
    return failReturnValue;
}
```

2) 深度优先搜索遍历

PS：

```
public static TR DFSVisit < TP, TR > (this Transform root, System.Func <
Transform, TP, TR > visitFunc, TP para, TR failReturnValue = default(TR))
{
    Stack<Transform> parents = new Stack<Transform> ();
    parents.Push(root);
    while(parents.Count > 0)
    {
        Transform parent = parents.Pop();
        TR ret = visitFunc(parent, para);
        if (ret != null && !ret.Equals(failReturnValue))
            return ret;
        for (int i = parent.childCount - 1; i >= 0; i--)
        {
            parents.Push(parent.GetChild(i));
        }
    }
```

```
            return failReturnValue;
        }
```

3) 根据名字查找并返回子孙-广度优先搜索

PS：

```
    public static T FindComponent_BFS<T>(this Transform trans, string childName) where T : Component
    {
        var target = BFSVisit<string, Transform>(trans,(t, str) =>
            {
                if (t.name.Equals(str))
                return t;
                return null;
            },childName);

        if (target == null)
        {
            Debug.LogError(string.Format("cann't find child transform {0} in {1}", childName, trans.gameObject.name));
            return null;
        }

        T component = target.GetComponent<T>();
        if (component == null)
        {
            Debug.LogError("Component is null, type = " + typeof(T).Name);
            return null;
        }
        return component;
    }
```

4) 根据 tag 名字查找并返回子孙-广度优先搜索

PS：

```
    public static Transform FindChild_ByTag(this Transform trans, string tagName)
    {
        var target = BFSVisit<string, Transform>(trans,(t, str) =>
            {
            if (t.tag.Equals(str)) return t; return null;
            },tagName);

        if (target == null)
        {
            return null;
        }

        return target;
```

```
    }
```

5) 根据名字查找并返回子孙-深度优先搜索

PS：

```
    public static T FindComponent_DFS<T>(this Transform trans, string childName) where T : Component
    {
        var target = DFSVisit<string, Transform>(trans,(t, str) =>
            {
            if (t.name.Equals(str)) return t; return null;
            },childName);

        if (target == null)
        {
            Debug.LogError(string.Format("cann't find child transform {0} in {1}", childName, trans.gameObject.name));
            return null;
        }

        T component = target.GetComponent<T>();
        if (component == null)
        {
            Debug.LogError("Component is null, type = " + typeof(T).Name);
            return null;
        }
        return component;
    }
```

6) 根据名字在子对象中查找组件

PS：

```
    public static T FindComponent<T>(this Transform trans, string name, bool reportError = true) where T : Component
    {
        Transform target = trans.Find(name);
        if (target == null)
        {
            if (reportError)
            {
                Debug.LogError("Transform is null, name = " + name);
            }

            return null;
        }

        T component = target.GetComponent<T>();
        if (component == null)
```

```
            {
                if (reportError)
                {
                    Debug.LogError("Component is null, type = " + typeof(T).Name);
                }

                return null;
            }

            return component;
        }
```

7) 初始化物体的相对位置、旋转、缩放

PS：

```
        public static void InitTransformLocal(this Transform trans)
        {
            if (trans == null)
            {
                Debug.LogError("Transform 里的 InitTransformLocal 扩展方法 Transform 为空");
                return;
            }
            trans.localPosition = Vector3.zero;
            trans.localScale = Vector3.one;
            trans.localRotation = Quaternion.identity;
        }
```

8) 直接设置物体 x 轴的世界坐标

PS：

```
        public static void SetPositionX(this Transform trans, float x)
        {
            var position = trans.position;
            position = new Vector3(x, position.y, position.z);
            trans.position = position;
        }
```

9) 直接设置物体 y 轴的世界坐标

PS：

```
        public static void SetPositionY(this Transform trans, float y)
        {
            var position = trans.position;
            position = new Vector3(position.x, y, position.z);
            trans.position = position;
        }
```

10) 直接设置物体 z 轴的世界坐标

PS：

```
public static void SetPositionZ(this Transform trans, float z)
{
    var position = trans.position;
    position = new Vector3(position.x, position.y, z);
    trans.position = position;
}
```

11) 直接设置物体 x 轴的本地坐标

PS：

```
public static void SetLocalPositionX(this Transform trans, float x)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(x, localPosition.y, localPosition.z);
    trans.localPosition = localPosition;
}
```

12) 直接设置物体 y 轴的本地坐标

PS：

```
public static void SetLocalPositionY(this Transform trans, float y)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(localPosition.x, y, localPosition.z);
    trans.localPosition = localPosition;
}
```

13) 直接设置物体 z 轴的本地坐标

PS：

```
public static void SetLocalPositionZ(this Transform trans, float z)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(localPosition.x, localPosition.y, z);
    trans.localPosition = localPosition;
}
```

14) 设置 Tag

PS：

```
public static void SetTag(this Transform trans, string tagName)
{
    if (trans == null)
    {
        Debug.LogError("SetTag 扩展方法里的 trans 为空");
        return;
    }
    trans.tag = tagName;
    if (trans.childCount > 0)
    {
        for (int i = 0; i < trans.childCount; i++)
```

```
                {
                    Transform child = trans.GetChild(i);
                    child.SetTag(tagName);
                }
            }
        }
```

2. 完整代码

PS：

```
namespace ExtendsFunction
{
    // Unity Transform 的扩展功能
    public static class TransformExtensions
    {
        /// <summary>
        /// 广度优先搜索遍历
        /// </summary>
        /// <param name="root"></param>
        /// <typeparam name="TP">遍历时调用的函数的参数的类型</typeparam>
        /// <typeparam name="TR">遍历时调用的函数的返回值的类型</typeparam>
        /// <param name="visitFunc">遍历时调用的函数
        /// <para>TR Function(Transform t, T para)</para>
        /// </param>
        /// <param name="para">遍历时调用的函数的第二个参数</param>
        /// <param name="failReturnValue">遍历时查找失败的返回值</param>
        /// <returns>遍历时调用的函数的返回值</returns>
        public static TR BFSVisit<TP, TR>(this Transform root, System.Func<Transform,
TP, TR> visitFunc, TP para, TR failReturnValue = default(TR))
        {
            TR ret = visitFunc(root, para);
            if (ret != null && !ret.Equals(failReturnValue))
                return ret;
            Queue<Transform> parents = new Queue<Transform>();
            parents.Enqueue(root);
            while (parents.Count > 0)
            {
                Transform parent = parents.Dequeue();
                foreach (Transform child in parent)
                {
                    ret = visitFunc(child, para);
                    if (ret != null && !ret.Equals(failReturnValue))
                        return ret;
                    parents.Enqueue(child);
                }
            }
```

```
            return failReturnValue;
        }

        /// <summary>
        /// 深度优先搜索遍历
        /// </summary>
        /// <param name="root"></param>
        /// <typeparam name="TP">遍历时调用的函数的参数的类型</typeparam>
        /// <typeparam name="TR">遍历时调用的函数的返回值的类型</typeparam>
        /// <param name="visitFunc">遍历时调用的函数
        /// <para>TR Function(Transform t, T para)</para>
        /// </param>
        /// <param name="para">遍历时调用的函数的第二个参数</param>
        /// <param name="failReturnValue">遍历时查找失败的返回值</param>
        /// <returns>遍历时调用的函数的返回值</returns>
        public static TR DFSVisit<TP, TR>(this Transform root, System.Func<Transform,
TP, TR> visitFunc, TP para, TR failReturnValue = default(TR))
        {
            Stack<Transform> parents = new Stack<Transform>();
            parents.Push(root);
            while (parents.Count > 0)
            {
                Transform parent = parents.Pop();
                TR ret = visitFunc(parent, para);
                if (ret != null && !ret.Equals(failReturnValue))
                    return ret;
                for (int i = parent.childCount - 1; i >= 0; i--)
                {
                    parents.Push(parent.GetChild(i));
                }
            }
            return failReturnValue;
        }

        /// <summary>
        /// 根据名字查找并返回子孙，广度优先搜索
        /// </summary>
        /// <param name="trans"></param>
        /// <param name="childName">要查找的子孙的名字</param>
        /// <returns>要查找的子孙的 Transform</returns>
        public static T FindComponent_BFS<T>(this Transform trans, string childName)
where T : Component
        {
            var target = BFSVisit<string, Transform>(trans, (t, str) =>
```

```
            {
                if (t.name.Equals(str))
                    return t;
                return null;
            }, childName);

            if (target == null)
            {
                Debug.LogError(string.Format("cann't find child transform {0} in {1}",
childName, trans.gameObject.name));
                return null;
            }

            T component = target.GetComponent<T>();
            if (component == null)
            {
                Debug.LogError("Component is null, type = " + typeof(T).Name);
                return null;
            }
            return component;
        }

        /// <summary>
        ///  根据 tag 名字查找并返回子孙，广度优先搜索
        /// </summary>
        /// <param name="trans"></param>
        /// <param name="tagName">要查找的子孙的 tag 名字</param>
        /// <returns>要查找的子孙的 Transform</returns>
        public static Transform FindChild_ByTag(this Transform trans, string tagName)
        {
            var target = BFSVisit<string, Transform>(trans, (t, str) =>
            {
                if (t.tag.Equals(str)) return t; return null;
            }, tagName);

            if (target == null)
            {
                return null;
            }

            return target;
        }

        /// <summary>
```

```
        /// 根据名字查找并返回子孙，深度优先搜索
        /// </summary>
        /// /// <param name="trans"></param>
        /// <param name="childName">要查找的子孙的名字</param>
        /// <returns>要查找的子孙的 Transform</returns>
        public static T FindComponent_DFS<T>(this Transform trans, string childName)
where T : Component
        {
            var target = DFSVisit<string, Transform>(trans, (t, str) =>
            {
                if (t.name.Equals(str)) return t; return null;
            }, childName);

            if (target == null)
            {
                Debug.LogError(string.Format("cann't find child transform {0} in {1}",
childName, trans.gameObject.name));
                return null;
            }

            T component = target.GetComponent<T>();
            if (component == null)
            {
                Debug.LogError("Component is null, type = " + typeof(T).Name);
                return null;
            }
            return component;
        }

        /// <summary>
        /// 根据名字在子对象中查找组件
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="trans"></param>
        /// <param name="name"></param>
        /// /// <param name="reportError"></param>
        /// <returns></returns>
        public static T FindComponent<T>(this Transform trans, string name, bool
reportError = true) where T : Component
        {
            Transform target = trans.Find(name);
            if (target == null)
            {
                if (reportError)
```

```
                {
                    Debug.LogError("Transform is null, name = " + name);
                }

                return null;
            }

            T component = target.GetComponent<T>();
            if (component == null)
            {
                if (reportError)
                {
                    Debug.LogError("Component is null, type = " + typeof(T).Name);
                }

                return null;
            }

            return component;
        }

        /// <summary>
        /// 初始化物体的相对位置、旋转、缩放
        /// </summary>
        /// <param name="trans"></param>
        public static void InitTransformLocal(this Transform trans)
        {
            if (trans == null)
            {
                Debug.LogError("Transform 里的 InitTransformLocal 扩展方法
Transform 为空");
                return;
            }
            trans.localPosition = Vector3.zero;
            trans.localScale = Vector3.one;
            trans.localRotation = Quaternion.identity;
        }

        /// <summary>
        /// 直接设置物体 x 轴的世界坐标
        /// </summary>
        /// <param name="trans"></param>
        /// <param name="x"></param>
        public static void SetPositionX(this Transform trans, float x)
```

```
{
    var position = trans.position;
    position = new Vector3(x, position.y, position.z);
    trans.position = position;
}

/// <summary>
///  直接设置物体 y 轴的世界坐标
/// </summary>
/// <param name="trans"></param>
/// <param name="y"></param>
public static void SetPositionY(this Transform trans, float y)
{
    var position = trans.position;
    position = new Vector3(position.x, y, position.z);
    trans.position = position;
}

/// <summary>
///  直接设置物体 z 轴的世界坐标
/// </summary>
/// <param name="trans"></param>
/// <param name="z"></param>
public static void SetPositionZ(this Transform trans, float z)
{
    var position = trans.position;
    position = new Vector3(position.x, position.y, z);
    trans.position = position;
}

/// <summary>
///  直接设置物体 x 轴的本地坐标
/// </summary>
/// <param name="trans"></param>
/// <param name="x"></param>
public static void SetLocalPositionX(this Transform trans, float x)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(x, localPosition.y, localPosition.z);
    trans.localPosition = localPosition;
}

/// <summary>
///  直接设置物体 y 轴的本地坐标
```

```
/// </summary>
/// <param name="trans"></param>
/// <param name="y"></param>
public static void SetLocalPositionY(this Transform trans, float y)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(localPosition.x, y, localPosition.z);
    trans.localPosition = localPosition;
}

/// <summary>
///  直接设置物体 z 轴的本地坐标
/// </summary>
/// <param name="trans"></param>
/// <param name="z"></param>
public static void SetLocalPositionZ(this Transform trans, float z)
{
    var localPosition = trans.localPosition;
    localPosition = new Vector3(localPosition.x, localPosition.y, z);
    trans.localPosition = localPosition;
}

/// <summary>
///  设置 Tag
/// </summary>
/// <param name="trans"></param>
/// <param name="tagName">tag 名称</param>
public static void SetTag(this Transform trans, string tagName)
{
    if (trans == null)
    {
        Debug.LogError("SetTag 扩展方法里的 trans 为空");
        return;
    }
    trans.tag = tagName;
    if (trans.childCount > 0)
    {
        for (int i = 0; i < trans.childCount; i++)
        {
            Transform child = trans.GetChild(i);
            child.SetTag(tagName);
        }
    }
}
```

```
        }
    }
```

二十三. GameObject 扩展方法

1. 方法

1) 实例化对象

PS：

```
public static GameObject InstantiateSelf(this GameObject target, Transform parent)
{
    if (target == null)
    {
        return null;
    }
    if (parent != null)
    {
        return Object.Instantiate(target, Vector3.zero, Quaternion.identity,
parent);
    }
    return Object.Instantiate(target, Vector3.zero, Quaternion.identity);
}
```

2) 销毁对象

PS：

```
public static void DestroyGameObj(this GameObject target)
{
    if (target == null)
    {
        return;
    }
    Object.Destroy(target);
}
```

3) 延迟销毁对象

PS：

```
public static void DestroyGameObjDelay(this GameObject target, float time)
{
    if (target == null)
    {
        return;
    }
    Object.Destroy(target, time);
}
```

4) 清空子物体

PS：

```
public static void ClearChildren(this Transform target, int index = 0)
{
```

```
            if (target == null)
            {
                return;
            }
            int len = target.childCount;
            for (int i = len - 1; i >= index; i--)
            {
                Transform child = target.GetChild(i);
                Object.Destroy(child.gameObject);
            }
        }
```

5) 添加组件

PS：

```
        public static T GetOrAddComponent<T>(this GameObject obj) where T :
    Component
        {
            if (obj == null)
            {
                return null;
            }
            T t = obj.GetComponent<T>();
            if (t == null)
            {
                t = obj.AddComponent<T>();
            }
            return t;
        }
```

6) 获取子节点数量

PS：

```
        public static int GetChildCount(this GameObject target)
        {
            if (target == null)
            {
                return 0;
            }
            return target.transform.childCount;
        }
```

2. 完整代码

PS：

```
using UnityEngine;

namespace ExtendsFunction
{
```

```
        // Unity GameObject 的扩展功能
        public static class GameObjectExtensions
        {
            #region Instantiate 实例化
            public static GameObject InstantiateSelf(this GameObject target, Transform parent)
            {
                if (target == null)
                {
                    return null;
                }
                if (parent != null)
                {
                    return Object.Instantiate(target, Vector3.zero, Quaternion.identity,
parent);
                }
                return Object.Instantiate(target, Vector3.zero, Quaternion.identity);
            }

            public static GameObject InstantiateSelf(this GameObject target, Component
parent)
            {
                return InstantiateSelf(target, parent?.transform);
            }

            public static GameObject InstantiateSelf(this GameObject target, GameObject
parent)
            {
                return InstantiateSelf(target, parent?.transform);
            }

            public static GameObject InstantiateSelf(this Component target, Component
parent)
            {
                return InstantiateSelf(target?.gameObject, parent?.transform);
            }

            public static GameObject InstantiateSelf(this Component target, GameObject
parent)
            {
                return InstantiateSelf(target?.gameObject, parent?.transform);
            }
            #endregion

            #region 销毁
```

```
        #region DestroyObject
        public static void DestroyGameObj(this GameObject target)
        {
            if (target == null)
            {
                return;
            }
            Object.Destroy(target);
        }

        public static void DestroyGameObj(this Component target)
        {
            DestroyGameObj(target?.gameObject);
        }
        #endregion

        #region DestroyGameObjDelay
        public static void DestroyGameObjDelay(this GameObject target, float time)
        {
            if (target == null)
            {
                return;
            }
            Object.Destroy(target, time);
        }

        public static void DestroyGameObjDelay(this Component target, float time)
        {
            DestroyGameObjDelay(target?.gameObject, time);
        }
        #endregion

        #region ClearChildren
        /// <summary>
        ///  清空子节点
        /// </summary>
        /// <param name="target"></param>
        /// <param name="index">清空到第几个子节点为止，默认为 0，即全部清除
</param>
        public static void ClearChildren(this Transform target, int index = 0)
        {
            if (target == null)
            {
```

```
                return;
            }
            int len = target.childCount;
            for (int i = len - 1; i >= index; i--)
            {
                Transform child = target.GetChild(i);
                Object.Destroy(child.gameObject);
            }
        }

        public static void ClearChildren(this Component target, int index = 0)
        {
            ClearChildren(target?.transform, index);
        }

        public static void ClearChildren(this GameObject target, int index = 0)
        {
            ClearChildren(target?.transform, index);
        }
        #endregion
        #endregion

        #region  添加组件
        /// <summary>
        ///  获取与添加组件
        /// </summary>
        /// <typeparam name="T">要获取或添加的组件名称</typeparam>
        /// <param name="obj"></param>
        /// <returns></returns>
        public static T GetOrAddComponent<T>(this GameObject obj) where T :
Component
        {
            if (obj == null)
            {
                return null;
            }
            T t = obj.GetComponent<T>();
            if (t == null)
            {
                t = obj.AddComponent<T>();
            }
            return t;
        }
```

```
        public static T GetOrAddComponent<T>(this Component target) where T :
Component
        {
            return GetOrAddComponent<T>(target?.gameObject);
        }

        public static Component GetOrAddComponent(this GameObject target,
System.Type t)
        {
            if (target == null || t == null)
            {
                return null;
            }
            var tt = target.GetComponent(t);
            if (tt == null)
            {
                tt = target.AddComponent(t);
            }
            return tt;
        }

        public static Component GetOrAddComponent(this Component target,
System.Type t)
        {
            return GetOrAddComponent(target?.gameObject, t);
        }
        #endregion

        #region GetChildCount  获取子节点数量
        /// <summary>
        ///  获取子节点数量
        /// </summary>
        /// <param name="target"></param>
        /// <returns></returns>
        public static int GetChildCount(this Component target)
        {
            if (target == null)
            {
                return 0;
            }
            return target.transform.childCount;
        }
        /// <summary>
        ///  获取子节点数量
```

```
        /// </summary>
        /// <param name="target"></param>
        /// <returns></returns>
        public static int GetChildCount(this GameObject target)
        {
            if (target == null)
            {
                return 0;
            }
            return target.transform.childCount;
        }
        #endregion

        #region Vector 类型比较
        /// <summary>
        ///  二维向量 a 中的每一个分量是否都是小于 b 的
        /// </summary>
        /// <param name="a">二维向量 a</param>
        /// <param name="b">二维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
</param>
        /// <returns></returns>
        public static bool Vector2ALessThanB(this Vector2 a, Vector2 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return (a.x < b.x) && (a.y < b.y);
            else
                return (a.x < b.x) || (a.y < b.y);
        }

        /// <summary>
        ///  二维向量 a 中的每一个分量是否都是大于 b 的
        /// </summary>
        /// <param name="a">二维向量 a</param>
        /// <param name="b">二维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
</param>
        /// <returns></returns>
        public static bool Vector2AGreaterThanB(this Vector2 a, Vector2 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return a.x > b.x && a.y > b.y;
```

```
            else
                return a.x > b.x || a.y > b.y;
        }

        /// <summary>
        ///  三维向量 a 中的每一个分量是否都是小于 b 的
        /// </summary>
        /// <param name="a">三维向量 a</param>
        /// <param name="b">三维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
</param>
        /// <returns></returns>
        public static bool Vector3ALessThanB(this Vector3 a, Vector3 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return (a.x < b.x) && (a.y < b.y) && (a.z < b.z);
            else
                return (a.x < b.x) || (a.y < b.y) || (a.z < b.z);
        }

        /// <summary>
        ///  三维向量 a 中的每一个分量是否都是大于 b 的
        /// </summary>
        /// <param name="a">三维向量 a</param>
        /// <param name="b">三维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
</param>
        /// <returns></returns>
        public static bool Vector3AGreaterThanB(this Vector3 a, Vector3 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return a.x > b.x && a.y > b.y && a.z > b.z;
            else
                return a.x > b.x || a.y > b.y || a.z > b.z;
        }

        /// <summary>
        ///  四维向量 a 中的每一个分量是否都是小于 b 的
        /// </summary>
        /// <param name="a">四维向量 a</param>
        /// <param name="b">四维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
```

```
</param>
        /// <returns></returns>
        public static bool Vector4ALessThanB(this Vector4 a, Vector4 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return a.x < b.x && a.y < b.y && a.z < b.z && a.w < b.w;
            else
                return a.x < b.x || a.y < b.y || a.z < b.z || a.w < b.w;
        }

        /// <summary>
        ///  四维向量 a 中的每一个分量是否都是大于 b 的
        /// </summary>
        /// <param name="a">四维向量 a</param>
        /// <param name="b">四维向量 b</param>
        /// <param name="useAndOperate">是否对每一个分量都进行与运算（默认是与）
</param>
        /// <returns></returns>
        public static bool Vector4AGreaterThanB(this Vector4 a, Vector4 b, bool
useAndOperate = true)
        {
            if (useAndOperate)
                return a.x > b.x && a.y > b.y && a.z > b.z && a.w > b.w;
            else
                return a.x > b.x || a.y > b.y || a.z > b.z || a.w > b.w;
        }
        #endregion
    }
}
```

二十四. 前缀红点树

二十五. TimerManager

二十六. 图像透明区域点击无效

在项目中，我们的图像往往都会带有透明通道。大多数时候，我们并不会在点击时去考虑点击位置的 Alpha 值。但是，在一些特殊情况下我们就需要去考虑点击区域的 Alpha 值了，例如 2D 游戏主城区中的点击。

图 48.　　透明区域影响点击判断

上图是一个示例，色相环的层级高于 ICON，ICON 的右下方区域会有一部分被色相环的透明部分遮挡。当我们点击那部分 ICON 区域时，其实际响应的对象将会是色相环。

要使点击时的响应对象不被透明区域影响，我们就需要使用到图像中的 Alpha 通道值了。首先需要启用纹理的读写开关，只有开启了纹理的读写设置，我们才能在 C#脚本中读取到对应的纹理数据，进而使用纹理的 Alpha 通道。

接着我们看一下代码中的应用，这部分非常的简单。Unity 已经为我们考虑了这种情况，在 Iamge 组件中为我们提供了对应的属性 alphaHitTestMinimumThreshold。

设置了 alphaHitTestMinimumThreshold 后，点击区域的 Alpha 值小于设定的 Alpha 阈值，我们的点击事件就将继续穿透下去，这样就可以使点击事件正确的传递下去。

注意：开启纹理的 read/write enabled 选项，运行时纹理就会在内存中被保留两份。一份在 GPU 显存中，一份在 CPU 可寻址内存（CPU-addressable memory）中。（提示：这是因为，CPU 读取 GPU 显存是非常慢的。CPU 读取显存中的一张贴图到一个临时的缓存中也不怎么现实）。

3. 代码

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

[RequireComponent(typeof(Image))]
public class SpriteAlphaTest : MonoBehaviour,IPointerClickHandler
{
    public string debugLog = "6666666666";

    void Start()
    {
        Image theImage = GetComponent<Image>();
        if (theImage != null && theImage.mainTexture.isReadable)
        /*小于设定的 Alpha 阈值将导致射线投射事件通过图像，只有大于时才能执行事件。
        需开启纹理的 read/write enabled，如果不开启时执行下面的代码就会报错*/
            theImage.alphaHitTestMinimumThreshold = 0.01f;
```

```
        }

        void IPointerClickHandler.OnPointerClick(PointerEventData eventData)
        {
            Debug.LogError(debugLog);
        }
    }
```

二十七. 生成不重复随机数

用一个数组来保存索引号，先随机生成一个数组位置，然后把随机抽取到的位置的索引号取出来，并把最后一个索引号复制到当前的数组位置，然后使随机数的上限减一。

具体如：先把这 100 个数放在一个数组内，每次随机取一个位置（第一次是 1-100，第二次是 1-99，…），将该位置的数用最后的数代替。（生成上万条不重复随机数据时，建议使用该方法）

PS：

```
    static int[] UseDoubleArrayToNonRepeatedRandom(int length, int minValue, int maxValue)
    {
        int seed = Guid.NewGuid().GetHashCode();
        Random radom = new Random(seed);
        int[] index = new int[length];
        for (int i = 0; i < length; i++)
        {
            index[i] = i + 1;
        }

        int[] array = new int[length]; // 用来保存随机生成的不重复的数
        int site = length;                    // 设置上限
        int idx;                                  // 获取 index 数组中索引为 idx 位置的数据，赋给结果数组 array 的 j 索引位置
        for (int j = 0; j < length; j++)
        {
            idx = radom.Next(0, site - 1);   // 生成随机索引数
            array[j] = index[idx];               // 在随机索引位置取出一个数，保存到结果数组
            index[idx] = index[site - 1];    // 作废当前索引位置数据，并用数组的最后一个数据代替之
            site--;                                   // 索引位置的上限减一（弃置最后一个数据）
        }
        return array;
    }
```