协同程序

运行在每一帧的Update之后

遇到yield return语法时 根据条件进行判定(延迟固定时间 延迟1帧 延迟等待下载完成等)

当条件不满足时会挂起 记录协程内部的执行状态(执行位置 内部局部变量等)

当条件满足时 协程会继续往下执行 后续没有yieldreturn语法时 协程会在一帧执行完毕

yield return 语句：当前挂起这个方法 然后在一帧继续从这里开始

```
    IEnumerator Fun()
    {
        yield return 0;//延迟一帧
        yield return 100;//(任意数字)都代表延迟一帧
        yield return null;//延迟一帧

        yield return StartCoroutine("Coroutine2");//等待Coroutine2协程完全执行完毕后在继续

        WWW www = new WWW("");
        yield return www;//等待www操作完成（下载完毕）在继续执行

        yield return new WaitForSeconds(3.0f);//延迟一定的秒数 受Time.TimeScale影响

        yield return new WaitForSecondsRealtime(3.0f);//延迟一定的秒数 不受Time.TimeScale影响

        yield return new WaitForFixedUpdate();//等待FixUpdate执行之后在继续

        yield return new WaitForEndOfFrame();//等待帧结束 等待所有的摄像机和GUI被渲染完成 在该帧在屏幕中显示之前执行

        yield break;//直接结束协程的后续操作
    }
```

return null和int效果一样 都是等待一帧 如果是int值 值的大小不起作用

不使用waitForSeconds 实现延迟3秒的功能

```
 void Start()
{
    StartCoroutine(Fun());
}


IEnumerator Fun()
{
    for (float timer = 0; timer < 3.0f; timer += Time.deltaTime)
    {
        yield return 0;
    }
    Debug.Log("3.0f时间到了");
}
```

协程的嵌套

```
void Start()
{
    StartCoroutine(Fun());
}


IEnumerator Fun()
{
    yield return StartCoroutine(Coroutine2());
    Debug.Log("第一个协程");//后打印
}

IEnumerator Coroutine2()
{
    yield return new WaitForSeconds(2.0f);
    Debug.Log("第二个协程");//先打印
}
```

协程的参数传递

开启协程的形式： 方法调用形式和字符串形式

字符串形式：只能传递一个object类型的参数（可能涉及拆装箱）

方法调用形式：可以正常传递多个参数

```
    void Start()
    {
        StartCoroutine(StartByFun(100,"HelloWorld"));
        StartCoroutine("StartByString", "Game");
    }

    IEnumerator StartByFun(int x,string y)
    {
        Debug.Log(x);
        yield return 0;
        Debug.Log(y);
    }

    IEnumerator StartByString(string y)
    {
        yield return 0;
        Debug.Log(y);
    }
```

使用WWW下载网络图片给物体材质赋值

```
    void Start()
    {
        StartCoroutine(Load());

    }

    IEnumerator Load()
    {
        //WWW参数 传递超链接 网络图片的地址
        WWW www = new WWW("https://timgsa.baidu.com/timg?
image&quality=80&size=b9999_10000&sec=1542605727828&di=fad3c9dbf3f8a91
c9e8f84c9d05d5594&imgtype=0&src=http%3A%2F%2Fpic51.nipic.com%2Ffile%2F
```

```
20141030%2F12810920_115806345000_2.jpg");

        yield return www;//等待www下载资源完成
        GetComponent<Renderer>().material.mainTexture = www.texture;
    }
```

协程的知识整理：

1 在程序中通过StartCoroutine开启协程 但是StopCoroutine只能终止字符串形式开启的协程

2 多个协程可以同时存在 他们会根据各自的启动顺序来更新

3 协程可以嵌套任意多层

4 协程可以被定义为Static 让多个脚本访问

5 协程不是多线程 他们运行于同一个主线程中 默认在每帧Update之后进行更新

6 如果程序需要大量的计算 可以考虑使用一个随时间进行的协程进行处理

用协程实现场景的异步加载功能

LoadScene场景加载是瞬间加载

即 在一帧完成老场景的释放和新场景的载入

如果场景很大 一帧需要释放和实例化的物体对象过多 容易触及到内存的峰值 对性能产生影响 造成卡顿现象

解决这个问题就是通过异步加载(后台加载)

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

public class LoadSync : MonoBehaviour {

    private AsyncOperation async;
    private float progress;
    void Start () {
        StartCoroutine("LoadAsync");
    }

```

```
    void Update()
    {
        //progress = async.progress * 100f;

        //operation的Progress最大值是0.9f
        //if (async.progress >= 0.9f)
        //{
          //  progress = 100f;
            //新场景使用的数据读取

          //  async.allowSceneActivation = true;
        //}
    }

    IEnumerator LoadAsync()
    {
        async=SceneManager.LoadSceneAsync("Scene2");
        async.allowSceneActivation = false;
        //yield return async;
        yield return new WaitForSeconds(2.0f);
        progress = 40;
        yield return null;
        progress = 43;
        yield return new WaitForSeconds(2.1f);
        progress = 60;

        yield return new WaitForSeconds(1.0f);
        progress = 100;

        async.allowSceneActivation = true;
    }

    void OnGUI()
    {
        GUI.Label(new Rect(0, 0, 200, 160), "进度:" +
progress.ToString() + "%");
    }
}
```

Unity的生命周期函数

Unity脚本工作原理：

所有继承于MonoBehaviour的类(脚本)

在游戏运行时都处于执行或者等待状态

Unity的后台 像一个管理者

把控制权和执行权传递给某一个脚本

当函数执行完毕后 控制权又被Unity后台收回

传递的方法就是通过Unity的事件函数

当Unity响应事件时 事件函数会被调用

这就是Unity的生命周期函数（用函数的方式 记录了一个脚本从生到死的一切事件）

事件函数一般用于绘制和计算

唤醒---激活---死亡

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

public class EventFun : MonoBehaviour {

    //类似于构造方法 唤醒 仅调用一次
    void Awake()
    {
        Debug.Log("Awake");
    }

    //在第一次Update调用之前
    //OnEnable之后调用
    void Start () {
        Debug.Log("Start");
    }

    //当物体被激活时调用一次 跟状态有关系 也可能被多次调用
    //当物体唤醒时 先调用Awake 然后调用一次OnEnable
    void OnEnable()
    {
```

```
        Debug.Log("OnEnable");
    }

    //当物体被设置成非激活时调用一次 跟状态有关系 也可能被多次调用
    //当一个物体被销毁时 先调用OnDisable 再调用OnDestroy
    void OnDisable()
    {
        Debug.Log("OnDisable");
    }

    //当物体要被销毁的时候调用一次
    void OnDestroy()
    {
        Debug.Log("OnDestroy");
    }
}

//绑定在空物体上
public class UI : MonoBehaviour {
    public GameObject cube;
    void OnGUI()
    {
        if (GUILayout.Button("Cube激活"))
        {
            cube.SetActive(!cube.activeSelf);
        }
        if (GUILayout.Button("Cube销毁"))
        {
            Destroy(cube);
        }
    }
}
```

Awake&Start

不管场景中有多少个脚本 只有当所有脚本的Awake全部执行完以后

才会调用脚本的Start

调用顺序和叫调用脚本Awake的顺序一致 顺序一旦确定除了有脚本被移除 否则不会更改

在Start方法里可以任意且放心的调用在Awake方法里初始化的数据

{

脚本1 Awake;

脚本2 Awake;

脚本3 Awake;

脚本1 Start;

脚本2 Start;

脚本3 Start;

}

更新

```
public class EventFun : MonoBehaviour {

    /// <summary>
    /// Update 最常用的更新函数 每一帧渲染和计算前执行一次
    /// 用于改变游戏物体的位置和状态等
    /// 适合在开发中做一些控制 但是不适合做物理更新
    /// Update受帧率影响 调用频率和设备性能，工作量有关
    /// 帧率下降时 调用次数随着下降 每帧的调用时间也不同(Time.deltaTime)
    /// </summary>
    void Update()
    {
        Debug.Log("Update");
    }

    /// <summary>
    /// FixedUpdate 每固定帧(物理帧)绘制时执行一次
    /// 不受游戏帧率的影响 固定的间隔时间调用(默认0.02 可以在TimeManager里设置)
    /// 所以 一般来说 给刚体添加作用力 更好的方式是在FixedUpdate里调用
    /// FixedUpdate更适合进行物理引擎的计算 可以得到最为精确的结果
    /// </summary>
    void FixedUpdate()
    {
        Debug.Log("FixedUpdate");
```

```
    }

    /// <summary>
    /// LateUpdate 和Update一样 每一帧调用 但是调用时机是在最后
    /// 当场景中所有的物体调用了Update和FixedUpdate函数且所有的动画都计算完成
    /// 在所有的功能基本完成 需要做一些逻辑补充 使用LateUpdate
    /// 1 摄像机跟随 人物先在Update里移动或者在FixedUpdate施加力 摄像机在LateUpdate里跟随 不会出现空帧的情况
    /// 2 动画效果改变 物体已经执行完物理计算 仍然需要一些通过动画改变的动作适用
    /// </summary>
    void LateUpdate()
    {
        Debug.Log("LateUpdate");
    }
}
```

Application事件

OnApplicationFocus//应用程序失去焦点

OnApplicationPause//应用程序暂停

OnApplicationQuit//应用程序退出

```
public class EventFun : MonoBehaviour {

    /// <summary>
    /// 当玩家获得或失去焦点时发送给所有的游戏物体
    /// 当游戏暂停时 先调用Pause在调用Focus
    /// 恢复暂停时  先调用Focus再调用Pause
    /// </summary>
    /// <param name="f"></param>
    void OnApplicationFocus(bool f)
    {
        if(f)
            Debug.Log("应用程序获得焦点");
        else
            Debug.Log("应用程序失去焦点");
```

```
    }

    void OnApplicationPause(bool p)
    {
        if (p)
            Debug.Log("应用程序暂停");
        else
            Debug.Log("应用程序继续运行");
    }

    void OnApplicationQuit()
    {
        Debug.Log("应用程序退出");
    }
}
```

物理引擎计算的相关事件函数

碰撞器

OnCollisionEnter//碰撞开始

OnCollisionStay//碰撞逗留

OnCollisionExit//碰撞退出

触发器

OnTriggerEnter//碰撞开始

OnTriggerStay//碰撞逗留

OnTriggerExit//碰撞退出

OnControllerColliderHit

角色控制器 和任意碰撞器产生碰撞的回调函数

OnParticleCollision

粒子碰撞函数

鼠标事件回调

```
//脚本必须绑定在一个有碰撞器的物体上
public class EventFun : MonoBehaviour {
    //当鼠标在一个碰撞器或GUI点下时 调用一次
    void OnMouseDown()
    {
        Debug.Log("Down");
    }

    //当鼠标在一个碰撞器或GUI移动拖拽时 调用多次
    void OnMouseDrag()
    {
        Debug.Log("Drag");
    }
    //当鼠标在一个碰撞器或GUI抬起时 调用一次
    void OnMouseUp()
    {
        Debug.Log("Up");
    }
     //当鼠标在一个碰撞器或GUI进入时 调用一次
    void OnMouseEnter()
    {
        Debug.Log("Enter");
    }

    //当鼠标在一个碰撞器或GUI停留时 调用多次
    void OnMouseOver()
    {
        Debug.Log("Over");
    }
    //当鼠标在一个碰撞器或GUI离开时 调用一次
    void OnMouseExit()
    {
        Debug.Log("Exit");
    }
}
```

物体的渲染回调

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

```
using UnityEngine;


//脚本必须绑定在一个有碰撞器的物体上
public class EventFun : MonoBehaviour {

    //当一个物体的renderer 不被任何相机所照射时 调用一次
    void OnBecameInvisible()
    {
        Debug.Log("不被照射");
    }
    //当一个物体的renderer 被任何相机所照射时 调用一次
    void OnBecameVisible()
    {
        Debug.Log("被照射");
    }


    //当物体即将被摄像机渲染前调用 每一帧都调用
    void OnWillRenderObject()
    {
        Debug.Log("我将要被渲染");
    }

    //多次调用 渲染Scene窗口的几何体
    void OnDrawGizmos()
    {
        Gizmos.DrawCube(Vector3.zero, Vector3.one);
        Gizmos.DrawWireCube(Vector3.one, Vector3.one);
    }

    //绘制GUI的回调函数 多次调用
    void OnGUI()
    {
    }
}

```

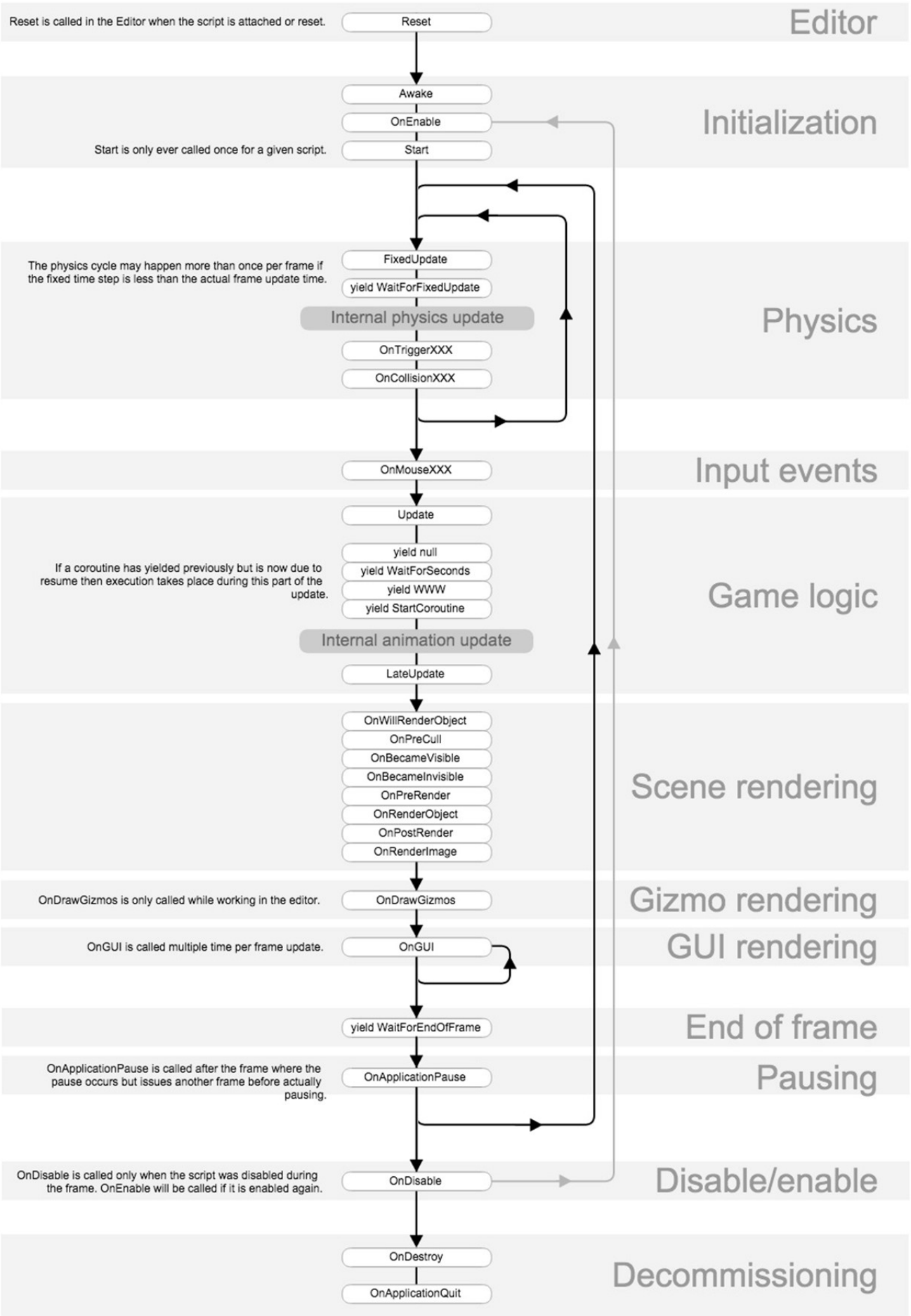
摄像机渲染的回调

```
    // <summary>
    /// 必须绑在摄像机上
    /// 渲染函数 多次调用
    /// 在消隐场景之前调用
    /// 消隐场景决定了哪些物体对于摄像机来说 是可见的
    /// </summary>
    void OnPreCull()
    {
        Debug.Log("场景的消隐");
    }

    //在相机渲染场景之前调用
    void OnPreRender()
    {
        Debug.Log("开始渲染场景");
    }

    //当渲染场景完成后调用
    void OnPostRender()
    {
        Debug.Log("渲染场景完成");
    }
```

生命周期流程图



在Unity的生命周期中 对内存消耗最大的两个函数:Instantiate和Destroy

任何一个GameObject的实例化 都需要对物体身上的所有组件进行初始化

不仅仅是我们实现的脚本 还包括Unity自带的组件都需要初始化

Destroy同理

所以在开发过程中 进量减少实例化和销毁的函数调用

当遇到需要频繁实例化和销毁的对象时 使用缓存池解决

缓存池 pool

缓存池简单理解就是预加载 预加载完成后不激活物体

当需要实例化时 不优先去使用Instantiate方法进行实例化

而是先去pool池里查找 如果pool池中有物体 就激活物体

当需要销毁物体时 也不直接去Destroy

首先看pool池中能不能存放下对象 如果放得下 就把物体设置为不激活 存入pool池

PoolManager 缓存池脚本

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

public class PoolManager : MonoBehaviour {
    //缓存池 所有需要缓存的道具都存放在List中 方便管理和查找
    private List<GameObject> objects = new List<GameObject>();

    //当前缓存池的最大个数(也不能预加载太多 因为内存的容量也是有限的)
    private const int maxCount=10;
    void Awake () {
        //预加载 在刚运行时 实例化MaxCount个道具 添加到缓存池
        for (int i = 0; i < maxCount; i++)
        {
           GameObject obj=Instantiate(Resources.Load("Water")) as
GameObject;
           obj.SetActive(false);
           objects.Add(obj);
        }
```

```
    }

    public GameObject CreateObjByPool()
    {
        //判断当前缓存池中有没有可用对象
        if (objects.Count <= 0)
        {
            GameObject obj = Instantiate(Resources.Load("Water")) as
GameObject;
            return obj;
        }
        else
        {
            //设置物体激活 从缓存池中移除
            objects[0].SetActive(true);
            GameObject obj = objects[0];
            objects.Remove(objects[0]);

            return obj;
        }
    }


    public void DestroyObjByPool(GameObject obj)
    {
        //销毁方法
        //当缓存池已满  直接删除物体
        if (objects.Count >= maxCount)
        {
            Destroy(obj);
            return;
        }
        //缓存池没有满 添加到pool池
        objects.Add(obj);
        obj.SetActive(false);
    }
}

```

生成道具的脚本

```
public class GameControl : MonoBehaviour {

    public GameObject water;
    private PoolManager pool;

    void Start () {

        pool = GetComponent<PoolManager>();
       // Invoke("CreateWater", 3.0f);  延迟3秒调用
        //延迟3秒调用 以后没延迟0.1秒调用一次
        InvokeRepeating("CreateWater", 3.0f, 0.05f);
    }

    void CreateWater () {
       //通过缓存池取得道具对象
       GameObject obj= pool.CreateObjByPool();
       obj.transform.position = transform.position;
    }
}
```

道具移除脚本

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

public class DestroyControl : MonoBehaviour {

    void OnBecameInvisible()
    {
        GameObject poolObj=GameObject.FindGameObjectWithTag("PoolManager");
        PoolManager pm = poolObj.GetComponent<PoolManager>();
        pm.DestroyObjByPool(gameObject);
    }
}

```

PATH-O-LOGICAL GAMES

PoolManager

$24.95

★★★★★ 56 user reviews

Add to Cart

Taxes/VAT calculated at checkout

Save at least another 25% when you bundle!

PoolManager, the original and best instance pooling solution for Unity, manages instances more efficiently to increase performance, organize the scene hierachy and is simple to