技巧

一. 修改 Unity 缓存位置

1. 修改 Unity GICache

打开 Unity 的编辑器，找到 Editor/Preferences/GICache，修改缓存位置。

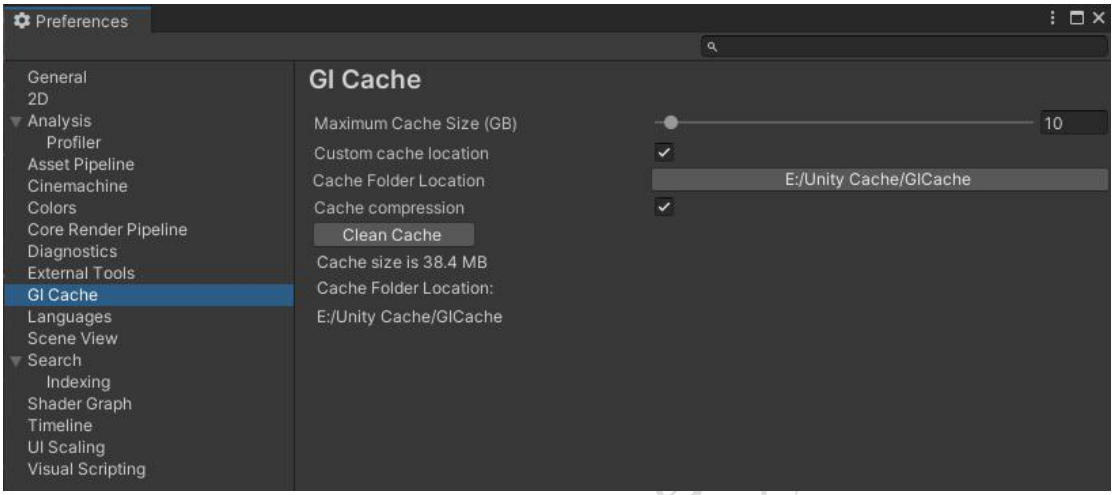

图 1. 图 1.1 示意图

找到 Cache Folder Location 后根据自己的需要，设置 GICache 的缓存位置即可。GICache 的默认缓存位置是 C 盘的 C:\Users\你的用户名\AppData\Local\Unity，AppData 文件是隐藏文件，需要手动设置显示隐藏文件。

图 2. 图 1.2 显示隐藏文件

2. 修改 Unity 的 Cache 缓存位置

1) 修改 Unity Cache

Unity 手册-全局缓存：https://docs.unity3d.com/cn/2021.2/Manual/upm-cache.html

Unity Package Manager 在下载包内容和元数据时，会将它们存储在全局缓存中。这样可以更有效地重复使用和共享包，甚至在脱机时也可以安装和更新存储的包。

在默认情况下，Unity 会将全局缓存存储在根目录中，根目录取决于操作系统（在 Windows 上还取决于用户帐户类型）：

操作系统	默认根目录
Windows（用户帐户）	%LOCALAPPDATA%\Unity\cache
Windows（系统用户帐户）	%ALLUSERSPROFILE%\Unity\cache
macOS	$HOME/Library/Unity/cache
Linux	$HOME/.config/unity3d/cache

图 3. 图 1.3 各操作系统默认根目录

Package Manager 使用两个不同的共享缓存，各自有不同的用途。它们存储在上述文件夹位置下的子目录中：

子文件夹	描述
npm	存储使用 npm 协议从注册表获得的数据。这包括包元数据和包 tarball。
packages	此缓存包含从注册表获取的包 tarball 的未压缩内容。

图 4.　　图 1.4 子文件夹描述

随着项目增加，我们的缓存会越来越大，导致我们的默认缓存位置空间不足。为了避免这种情况的出现，就需要修改 Unity 的缓存位置。

① 找到我的电脑，右键找到属性选项

图 5.　　图 1.5 找到我的电脑的属性选项

② 打开高级系统设置

图 6.　　图 1.6 找到高级系统设置

③ 打开环境变量

图 7.　图 1.7 环境变量

④　自定义共享缓存位置

可以使用两个环境变量来覆盖任一全局缓存文件夹：

- UPM_NPM_CACHE_PATH 覆盖 npm 文件夹路径。例如 (macOS):
  `UPM_NPM_CACHE_PATH=/dev/ssd/shared/Unity/cache/npm`
- UPM_CACHE_PATH 覆盖 packages 文件夹路径。例如 (macOS):
  `UPM_CACHE_PATH=/dev/ssd/shared/Unity/cache/packages`

图 8.　图 1.8 覆盖全局缓存文件夹

⑤　在系统变量中新建 2 个环境变量

变量名 1：UPM_CACHE_PATH

变量值 1：新的缓存文件夹地址

变量名 2：UPM_NPM_CACHE_PATH

变量值 2：新的缓存文件夹地址

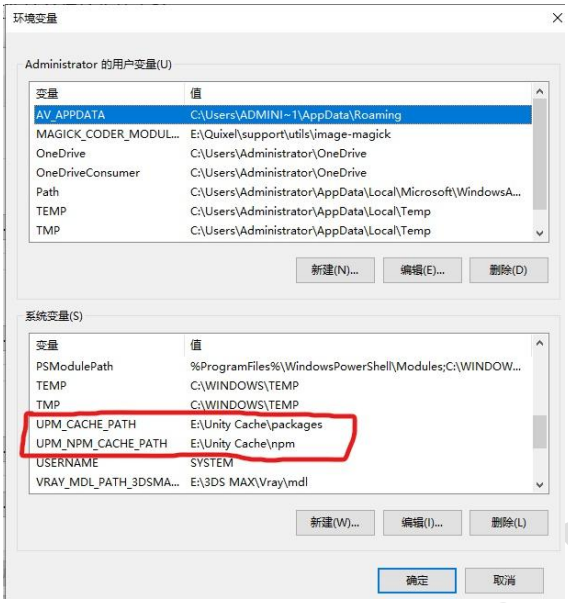

图 9.　　图 1.8 新建环境变量

⑥　重启电脑。

⑦　打开 UnityHub 和 Unity 项目，查看新的缓存目录里是否缓存了东西，如果有就可以将默认缓存文件夹删除。如果没有，就重复上述操作。

3. 修改 Unity Asset Store 的默认下载路径

①　在 Unity 商城下载的文件，默认存储路径是 C:\Users\用户名\AppData\Roaming\Unity\Asset Store-5.x

AppData 文件是隐藏文件，需要手动设置以显示隐藏文件。

图 10.　　图 2.1 显示隐藏文件

②　找到 Asset Store-5.x 文件夹后，将其整个剪贴到你想要存放的目录。

③　然后进 cmd，输入命令：

mklink /j "C:\Users\Administrator\AppData\Roaming\Unity\Asset Store-5.x" "F:\Unity Asset Store\Asset Store-5.x"

创建目录软链接。

二. 优化

1. 全平台优化方案

1) FixedUpdate 函数优化

在脚本中使用 Unity 的 FixedUpdate 函数时，一定要牢记尽可能不要写太多无需在物理帧重复调用的代码在方法内，因为默认情况下 Unity 的虚拟机在执行 FixedUpdate 函数时是以每秒执行 50-100 次该方法来处理场景中的每个脚本与每个对象的。FixedUpdate 函数的执行频率在 Unity 中是可以被修改的，只需要在菜单中依次选择 Edit→Project Setting→Time 命令，就可以在 Inspector 视图中显示 TimeManager 属性面板并修改默认的执行频率。

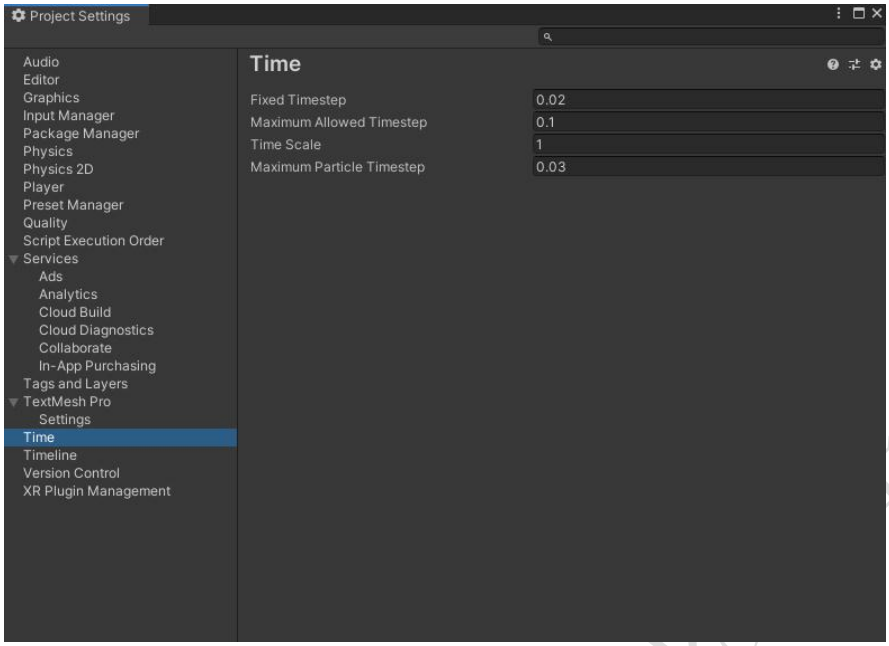

图 11.　图 3.1 TimeManager 属性面板

FixedUpdate 函数与 Update 函数的区别在于，Update 函数会在每次渲染新的一帧时执行（即每帧调用），它会受到当前渲染物体的影响，毕竟渲染的帧率并不是一成不变的，所以渲染的时间间隔也会变化，也就是说设备的性能会影响到 Update 函数的更新频率，而 FixedUpdate 函数则不会受到帧率的影响，在 TimeManager 面板可以看出它是以固定的时间间隔被调用的。使用在用法的处理上，FixedUpdate 更多适用于处理物理引擎。Update 因为受渲染物体的影响，使用更多地把 Update 用于脚本逻辑的控制上。

2) 新建脚本时删除不需要函数

当开发者在 Unity3D 中创建新的脚本时，Unity 默认会生成空的 Update 函数，但是并不是每一个脚本都需要使用到它。我们可以在引擎安装的对应位置找到默认脚本代码，引擎更目录->Editor->Data->Rescources->ScriptTemplates 下的 81-C# Script-NewBehaviourScript.cs。找到默认的脚本代码后使用记事本将其打开，删除对应的 Update 函数并将其保存后退出，这样在 Unity 创建新的脚本时就不会在默认添加 Update 函数了。

如果并未修改新建脚本中自带的默认方法，当该新建脚本并不需要用到 Update 函数时，应该将 Update 函数从脚本中删除。如果要在场景中查找对象，应该在游戏初始化时进行查找，即尽量在 Start 或 Awake 函数中执行查找，而不是在初始化完成后再在 Update 函数内执行 Find、FindObjectOfType、FindGameObjectsWithTag 这些寻找物体的函数，这一行为会极大的消耗设备的性能。

3) 优化数学计算

因为在程序世界中除法的运算比乘法要复杂，因此能够使用乘法运算时优先考虑乘法运算而不是除法运算。另外，如果可以在代码中使用整型（int）就不要优先考虑浮点型（float），尽量少用复杂的数学函数比如三角函数等，使用复杂的数学函数会增加性能的消耗。

4) 减少临时变量使用

当脚本中需要使用到变量时，应当优先考虑全局变量而不是局部变量，特别是在 Update 等实时调用的函数中。

5) 减少不常用方法使用次数

当一个程序不必要每帧都执行时，就可以考虑将其剥离出 Update 等实时调用的函数，

然后在 Coroutines（协程）函数中执行函数体；当函数需要被定时重复的调用时，可以使用 InvokeRepeating 函数实现。

```
Void Start()
{
//启动 1.5 秒后每隔 1 秒执行一次 DoSomeThing 函数
InvokeRepeating(“DoSomeThing”,1.5f,1.0f);
}
```

2. 脚本性能优化

1) 减少 Update 内 GetComponent 的调用

当开发者在脚本中使用 GetComponent 方法或者 unity 内置的其他组件访问器（如 Find 等），应该在脚本生命周期最开始的时候就定义它，即在 Enable、Start、Awake 函数中使用。因为这些函数会带来明显的性能开销，如果在程序运行的过程中去使用会带来卡顿等现象，影响用户的体验。

2) 尽量避免动态内存的分配

为数组、表单等分配内存应尽可能的在脚本生命周期最开始的时候，在程序的运行过程中应该避免再分配新的对象，特别是在 Update 这样的实时调用函数中。在实时调用函数中分配新的内存，不仅仅会增加内存的开销，与此同时也会增加内存回收的性能开销。

3) 减少 GUILayout 布局的使用

老子曾经说过，有得必有失，有失必有得，事多无兼得者。GUILayout 虽然能让开发者极为便捷的调整间距，但是不可避免的会带来一些性能的损失。对于不需要过多变化的可以使用 GUI 手动处理布局，以减少性能上的开销，如果想要完全静止 GUILayout 布局的使用可以在脚本中设置 useGUILayout 为 false。

3. 物理性能优化

1) 减少网格碰撞体使用

在开发过程中能够使用球体(Sphere)、盒体(Box)等碰撞体来尽可能地接近网格的形状，就应少的使用网格碰撞体(Mesh Collider)，因为网格碰撞体的数据过于复杂会造成较大的性能消耗。

另外，父对象的子碰撞体，在刚体中将作为复合碰撞体(Composite Collider)成为一个整体，在场景中被视为一个集体进行控制。

2) 精度需求不高应避免大量使用自带 Mesh 组件

Unity3D 引擎内置的多边形的模型精度比较高，当对于物体的精细度要求不高时，因避免大量使用 Unity 自带的 Sphere 等 Mesh 组件，可以导入一系列简单的 Low Poly 风格的 3D 模型代替。

3) 使用 Profiler

Unity 场景中的物理计算的总量，是由场景中非休眠刚体和碰撞体的数量以及碰撞体的复杂度决定的。在程序运行时可以使用 Unity 的性能分析工具—Profiler，来查看所制作的游戏在各方面所耗费的时间，分析当前游戏在各种情况下 GPU、CPU 性能，为后续的开发和优化提供数据参考。

4. Shader 优化方案

对数据、文字、声音、图形、图形和视频等信息进行加工处理就是计算机的基本功能，在这之中的数据可以分为两个大类：一类是数值数据，这类数据是表示数量大小的数据，如+314，-3.14 等，有“量”的概念；而另外一类是非数值数据，这类数据并不会用来表示数

值的大小，通常情况下也不会对这类数据进行算术运算，比如字符、字符串、图形符号等。在现实生活中，人们通常使用的是十进制数进行数据的统计与计算，但是计算机内的直接存储和运算却难以直接使用十进制。因此，计算机便将十进制数作为人机交互的中介，对于数据的存储和运算就采用易于物理实现、运算规则简单、可靠性高逻辑判断方便的二进制数。

因为计算机中的数据都是以二进制形式进行存储和运算的，而文字、声音、图形、图形和视频等信息并不是以二进制进行表示的，因此要将它们进行数字化（即将非二进制数转换为二进制数）。

1) 数的加减乘除

① 加法运算

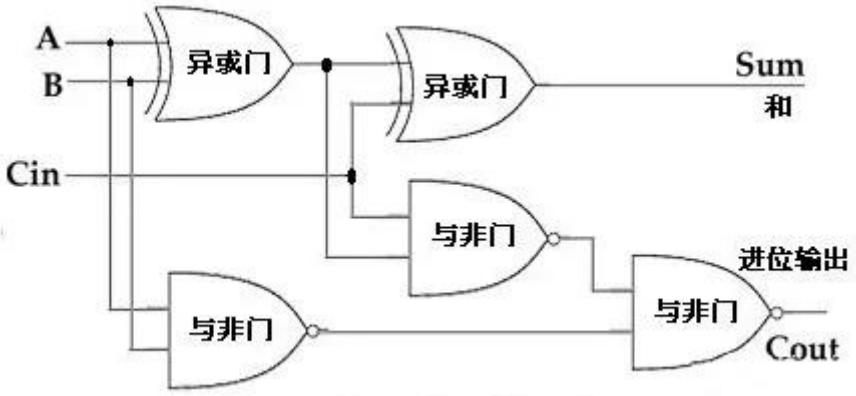

图 12.　图 6.2 全加器逻辑电路

计算机在实际运算过程中，加法二进制的最终数据会在计算机的全加器逻辑电路进行，也就是物理层。其中，异或门输出的是 Y=A^B（即两个值不相同结果为 1，相同则为 0），与非门的输出是先进行与运算(&，即两个数值相同时返回的结果为 1，不同时返回的结果为 0)再进行取反的非运算(~)，即 Y=!(A&B)。通过推导可以得到公式：

$$Sum = A\hat{}B\hat{}Cin \quad Count = (A\hat{}B\&Cin)|(A\&B) \quad (15)$$

```
public int GetSum(int a,int b)
{
        if (b == 0) //判断参数 b 的值
        {
            return a;//返回数值 a
        }
        int sum, carry;
        sum = a ^ b;
        carry = (a & b) << 1;
        return GetSum(sum, carry);
}//递归运算
```

“按位与”运算符（&）的规则是，当两个数值进行与运算时，先将两个数值转换为位数相同的二进制数，再依次对每一位数进行比较，只有当对应的两个二进位均为 1 时，返回的结果才会为 1；如果两个二进位有一个不为 1，返回的结果就是 0。

01010100&00111011 = 00010000

“左移”运算符（<<）的运算规则是，在“<<”左边的运算数的各二进位全部被左移若干位，在“<<”右边的数值指定各二进位被移动的位数，高位丢弃，低位补 0。

如果 x=15，也就是 00001111，左移 2 位得 00111100 即 x<<2=60

由前面可以知道，加法运算其实就是不断的异或(^)加上移位操作的结果。

② 减法运算

当在进行减法运算时，首先想到的是加上一个负数。但是在计算机中，用于区分正负的符号位是不能直接参与运算的，在运算时必须分别计算符号位和数值位。当两个数值进行加法运算时，如果这两个数值是同号的（即都是正数或负数），则数值相加即可；如果这两个数值是异号的（即一正一负），则要进行减法运算。在进行减法运算时需要比较两者的绝对值大大小，用较大的数值去减去较小的数值，在得出运算结果时还要选择恰当的符号。

上述这些问题可以通过使用补码解决，补码可以把减法运算转换为加法运算。正数的补码最高位为符号 0，数值位和原码保持一致；而负数的补码最高位是符号 1，数值位是原码逐位取反得到反码，然后再在反码的最低位加一。数值 0 的补码只有一种形式，就是 N 位 0。

a = 3 = 0000 0011,b = 2 = 0000 0010

c = a-b=a+(-b)

c = 0000 0011 + 1111 1110

c = 1 0000 0001

因为是 8 位计算，最高位进位得到的 1 加入其中就是 9 位，超过了 8 位限制因此进位得到的 1 被舍弃，最终得到的结果就不是 1 0000 0001 而是 0000 0001。

在计算机中，由于机器码的位数是有限的，当最高位因为计算而进位超出了给定的取值范围，就被称为溢出。当两个正数进行加法运算，所得到的最终结果大于了机器码所能表示的最大正数，这一情况就被称为正溢。当两个负数进行加法运算时，所得到的最终结果小于了机器码所能表示的最小负数，这一情况就被称为负溢。在开发过程中出现溢出会导致计算的最终结果出现错误，因此正常情况下溢出是不被允许的。

③ 乘法运算

在计算机中乘法有两种计算方式，一种就是循环加法，一种是乘法器，乘法器分为模拟乘法器和硬件乘法器。

2*3=0000 0010 * 0000 0011

3 的第 0 位是 1,2 左移 0 位，结果为 0000 0010 = 2

3 的第 1 位是 1,2 左移 1 位，结果为 0000 0100 = 4

3 的其他位都是 0，因此不再移位

两次左移的结果相加，即 0000 0010 + 0000 0100 = 2+4 = 6

可以看出乘法器是根据乘数的 bit 位的 0 与 1，来将被乘数进行左移多少位，然后累加得到最终的结果。

④ 除法运算

最简单的除法运算可以通过循环减法来实现，比如 5 除以 2,5 一直减去 2，直到减完的结果小于 2，得到的就是除法的结果，余数就是剩下的 1；除法也具有专门做除法的逻辑电路。

50 / 20 = 0011 0010 / 0001 0100

取 50 的最高位 001|1 0010，001 < 0001 0100

左移一位 0011| 0010, 0011 < 0001 0100

左移一位 0011 0|010，0 0110 < 0001 0100

左移一位 0011 00|10, 00 1100 < 0001 0100

左移一位 0011 001|0, 001 1001 > 0001 0100, 商 1，余 101|0

将大于 20 的数位记录下来就是 0000 0010，结果为 2，余数 1010 = 10

所以结果就是 50 / 20 = 2 余 10

可以看出除法器是根据从被除数高位开始取数，每次移动一位取到数后和除数比较，如果大于除数就记录当前被除数的数位，然后将记录的结果变为二进制数就是最后的结果。

虽然现在在进行乘法运算和除法运算时，获得了除法器和乘法器的帮助使得计算的效率比过去的累加和累减高得多，但是这些操作依然是非常消耗性能的操作，尤其是在进行除法运算的时候，还必须要对数值进行大于小于的判断。因此在片元作色器的编写过程中应该尽可能的避免高性能消耗的除法操作，转而使用其他低性能消耗的运算进行组合替代。

2) 逻辑判断

CPU 的主要功能是解释计算机指令以及处理计算机软件中的数据，而逻辑判断是所有程序中必不可少的操作，一般情况下的逻辑判断都是在 CPU 上运行的，因此 CPU 是一台计算机的运算核心和控制核心。而 GPU 则和 CPU 不同，GPU(Graphics Processing Unit)作为计算机中的图像处理器，它比 CPU 更加的专门化，GPU 处理器专门用来进行图像运算，在执行高复杂度的数学计算和几何计算时，运行效率远远高于 CPU 的运行效率。换句话说，GPU 图像处理器就是专为执行复杂的数学和几何计算而诞生的。

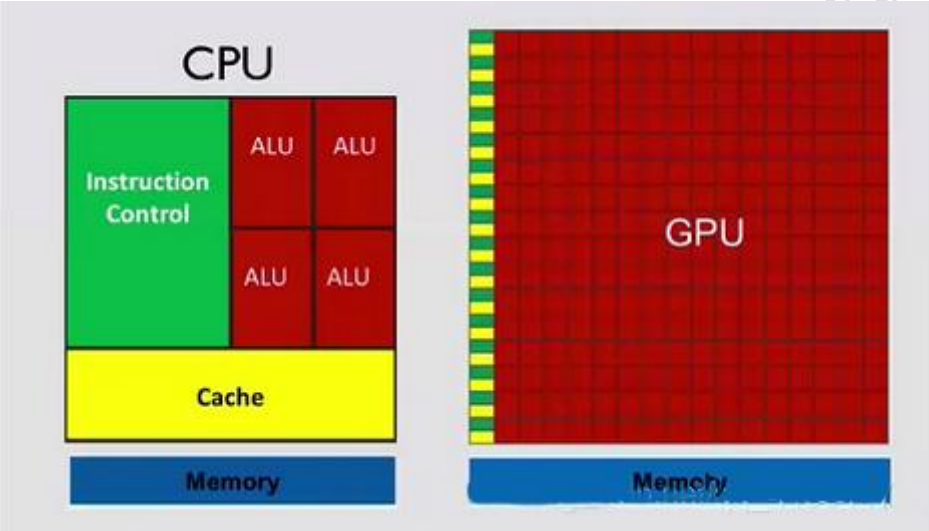

图 13.　图 6.3 架构图

CPU 和 GPU 拥有各自擅长的领域，GPU 擅长处理那些具有极高的可预测性和大量相似的运算以及高延迟、高吞吐的架构运算。而 CPU 所擅长处理的就是需要快速完成响应的实时信息，因此 CPU 便需要针对延迟进行特别的优化，所以晶体管数量和能耗都被用在了分支预测、乱序执行、低延迟缓存等控制部分。

在 Unity Shader 中编写的三目运算最后也会被编译为 if-else 语句，毕竟三目运算实际上仅仅只是简化了的 if-else 语句。虽然 Shader 中的每个片元依然会执行逻辑判断的每个分支，但是寄存器的写入只在片元采取行动的分支上执行。因此在 Shader 可以用 Shader 变体来进行不同状态下的渲染分支，用一些函数去代替 if-else 的判断，不推荐在 Shader 中使用逻辑判断，因为会带来不必要的性能消耗。

例如可以用 setp()函数来做大小与的判断：

```
a = step(b, c);// 等价于 if (b <= c) a = 1 else a = 0
```

当需要为角色添加一个描边光效时，可以根据 NdotV 的结果来设定描边光效越靠近边缘越强，然后使用一个变量去控制描边光效在角色表面的范围。

在 Shader 代码编写之初，使用的是三目运算进行判断：

```
fixed3 c = (1 - NdotV)   < _StatusColorEdge ? fixed3(0,0,0) : blendColor;
```

经过一次优化，将三目运算变更为 clamp 函数和 ceil 函数的组合，clamp 函数会保证 NdotV 和边界控制的结果永远都是在 0-1 的范围，保证值的正确性；而 ceil 函数会返回大于或者等于指定表达式的最小整数，即将结果向下取整变为 0 或者 1，然后再对颜色进行插值运算：

```
fixed temp = ceil( clamp(1 - NdotV - _StatusColorEdge, 0, 1));
fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);
```

最后用 step 函数代替两个函数，同时减法也换成了加法：

```
fixed temp = step(NdotV + _StatusColorEdge, 1);
fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);
```

当开发者在 Shader 脚本中使用 if 进行判断的时候，可以将其拆分为两种情况：

```
if (a > b)
{x}else
{x}
```

判断条件中的 b 为静态条件，即 const float 或者 define。当 b 为静态条件时，编译器会对这类情况做优化，使其直接进行批量计算，因为同一批像素从 Shader 中获得的都是相同的数值。

判断条件中的 b 为动态数值，也就是说具体的值在运行前无法得知的，只有在程序运行的时候才会知道具体的值。就会导致 GPU 做分批处理，无法最大化利用同时计算，因为无法确定的值就会导致同一批像素在一个区间内浮动，那么同一时间有的返回 true 有的返回 false。

5. 简易总结

1) 物理引擎优化

这块化没有太多的空间了，用其它的技术去替代不用物理引擎，减少物理引擎的迭代参数,减少计算量，减少物理刚体的数目。

2) 网络优化

异步 IO 代替同步 IO，多线程处理网络消息,，protobuf 序列化与反序列化优化网络包体体积。KCP 替换传统的 TCP。

3) 包体优化

优化图片，声音体积，通过改变压缩参数来降低这些资源的体积大小。可以使用服务器上部署资源包来实现打空包机制进一步减少包体体积。

4) 内存优化

① 减少资源的内存占用，不用的资源卸载掉，吃入显存的纹理可以采用平台支持的压缩格式。缓存池，减少内存碎片，减少对象的反复构建，避免 GC 峰值冲击等。

② 警惕配置表的内存占用。

③ 检查 ShaderLab 内存占用:

避免使用 Standard 材质，做好相应的 variant skip。

排查项目冗余的 Shader。

使用 shader_feature 替代 multi_compile，这样只会收集项目里真正使用的变体组合，避免变体翻倍。

④ 检查纹理资源的尺寸、格式、压缩方式、mipmap、Read & Write 选项使用是否合理。

⑤ 检查 Mesh 资源的 Read & Write 选项、顶点属性使用是否合理。

⑥ 代码级别的检查，如 Cache 预分配空间、容器的 Capacity、GC 等。

⑦ 使用 Profiler 定位下 GC，特别是 Update 类函数里的。如：字符串拼接、滥用容器等。

⑧ 合理控制 RenderTexture 的尺寸。

⑨ 优化动画 Animation 的压缩方式、浮点精度、去除里面的 Scale 曲线数据。

⑩ 减少场景 GameObject 节点的数量，最好支持工具监控。

5) 模型优化

通过细节增强,法线贴图，高度贴图，凹凸纹理等减少模型面数的同时获得很好的效果。

6)功耗发热优化

① 控制全屏后处理的使用数量、关键的性能参数、分辨率等。
② 支持自动省电模式：如当玩家 1min 没有操作时自动进入省电模式。

常用的省电优化项：

① 降低 FPS。
② 将分辨率缩放到最低(如 1280 * 720)。
③ 降低 Update / LateUpdate 等高频函数的执行频率。
④ 关闭一些显示效果，如后处理皮肤、bloom、抗锯齿等。
⑤ 如果开了实时阴影，考虑降低更新频率、调低效果参数等。

7)CPU 端性能优化

① 逻辑和表现尽可能分离开，这样逻辑层的更新频率可以适当降低些。
② 对于一些热点函数，如 mmo 的实体更新、实例化，使用分帧处理，分摊单帧时间消耗。
③ 做好同屏实体数量、特效数量、距离显隐等优化。
④ 完善日志输出，避免没必要的日志输出，同时警惕日志字符串拼接。
⑤ 使用骨骼烘焙 + GPUSkinning + Instance 降低 CPU 蒙皮骨骼消耗和 drawcall。
⑥ 开启模型的 Optimize GameObjects 减少节点数量和蒙皮更新消耗。
⑦ UI 拼预制做好动静分离，对于像血条名字这种频繁变动的 ui，做好适当的分组。
⑧ 减少 C#和 lua 的频繁交互，尽量精简两者传递的参数结构。
⑨ 使用 stringbuilder 优化字符串拼接的 gc 问题。
⑩ 删除非必要的脚本功能函数，特别是 Update/LateUpdate 类高频执行函数，因为会产生 C++到 C#层的调用开销. 对于 Update 里需要用到的组件、节点等提前 Cache 好。
⑪ 场景里频繁使用的资源或数据结构做好资源复用和对象池。
⑫ 对于频繁显示隐藏的 UI,可以先移出到屏幕外,如果长时间不显示再进行 Deactive。
⑬ 合理拆分 UI 图集，区分共用图集和非共用图集，共用图集可以常驻内存，非共用图集优先按功能分类,避免资源冗余。
⑭ 使用 IL2CPP, 编译成 C++版本能极大的提升整体性能。
⑮ 避免直接使用 Material.Setxxx/Getxxx 等调用，这些调用会触发材质实例化消耗，可以考虑使用 SharedMaterial / MaterialPropertyBlock 代替。
⑯ 合并 Shader 里的 Uniform 变量。

三. 快捷键

1. Unity

① Ctrl + Shift + F 移动物体与视窗对齐(Align With View)
② Ctrl + Alt + F 将物体移动到 Scene 视图中央
③ V 顶点捕捉
④ Ctrl+D 复制
⑤ Ctrl + Shift 自动捕捉平面
⑥ 按住 Alt 键在 hierarchy 窗口可快速完全展开和完全收起
⑦ Shift+Space 最大化窗口

⑧ Shift+F 动态聚焦所选物体（复按可以调整聚焦距离）
⑨ Shift+Alt+A 改变所选物体的可见性
⑩ Ctrl + P 运行游戏，运行状态下使用可结束运行
⑪ Ctrl + Shift + P 暂停游戏运行，在暂停状态下我们可以逐帧查看
⑫ Ctrl+Alt+P--逐帧播放按钮 单帧进行预览

2. Visual Studio

① Ctrl + Shift + 空格键 查看代码重载（需要光标在括号中间）
② Ctrl + R 重命名
③ Ctrl + F 查找
④ Ctrl + K + C 快速注释选中，未选中就注释该行
⑤ Ctrl + K + U 快速取消选中的注释，未选中就取消该行的注释
⑥ Ctrl+键盘右键 跳过该单词
⑦ Shift+任意方向键 范围选取
⑧ Shift + Alt +方向键 多行编辑
⑨ Ctrl + G 转到指定行
⑩ Ctrl + F12 转到声明
⑪ Alt + 上/下方向键 选中内容向上/下移动，无选中则对行进行操作
⑫ Ctrl + Enter 在上面插入一行

四. 脚本栏的特性扩展

1. System 命名空间

1) **序列化**-[Serializable]
表示类或结构可以序列化，其在 System 命名空间下。

2) **反序列化**-[NonSerialized]
反序列化一个变量，使其在监视面版上隐藏，其在 System 命名空间下。

2. UnityEngine 命名空间

1) **强制序列化**-[SerializeField]
使 Private 变量在 Inspector 窗口中可见。

2) **隐藏属性**-[HideInInSpector]
使 Public 变量在 Inspector 窗口中不可见。

3) **属性标题**-[Header]
为后续区域代码拟定一个标题，用于区分和概述该区域代码含义。

4) **属性提示**-[Tooltip]
实现在 inspector 窗口，鼠标位于该变量名字上时，提示该变量的描述信息。

5) **空行属性**-[Space]
在 Inspector 窗口创建空行，以隔开上下可视参数。

6) **范围值属性**-[Range()]
使得变量的值仅在该范围内修改，并且可以在 Inspector 窗口呈现滑动条修改变量的效果。

7) **多行**-[Multiline]
该特性可以让 string 变量在 Inspector 面板上多显示几行，具体能多显示几行取决于所传递的特性参数。特性参数的数值越大于 4，在 Inspector 面板上能显示出的行数误差越大。

PS：

```
//特性参数为 5，就是多显示 5 行
[Multiline(5)]
public string testStr = "";
```

图 14.　使用 Multiline 特性，使字符串显示多行

8) **输入域**-[TextArea]

对于字数较长的字符串，扩展其在 Inspector 中的编辑区大小。虽然看起来和前面的 Multiline 特性差不多，但是 TextArea 具有一个滑动条，可以帮助我们快速移动。

9) [RequireComponent(typeof(xxx))]

可要求我们的脚本对象必须含有一个该组件，若脚本对象没有该组件，则会为该脚本对象自动添加 xxx 组件。

10) [CreateAssetMenu(fileName = "xx", menuName = "xx", order = x)]

将 ScriptableObject 派生类型标记为自动在 Assets/Create 子菜单中列出，以便可以轻松创建该类型的实例并将其作为“.asset”文件存储在项目中。

11) **永远执行**-[ExecuteAlways]

使脚本的实例始终执行，无论是作为播放模式的一部分还是在编辑时。

默认情况下，MonoBehaviours 仅在播放模式下执行，并且仅在它们位于包含用户场景的主舞台的游戏对象上时执行。它们不会在编辑模式下执行，也不会在预制模式下编辑的对象上执行，即使在播放模式下也是如此。通过添加此属性，MonoBehaviour 的任何实例都将始终执行其回调函数。

注意区分 Edit Mode 和 Play Mode 代码执行逻辑，Edit Mode 下 Update 仅在 Scene 更改时执行，OnGUI 仅在接收到"仅限 Editor 之外事件"时（如：EventType.ScrollWheel）且不响应 Editor 快捷键，OnRenderObject 和其他渲染回调方法在场景重绘时调用。

12) **禁止多组件**-[DisallowMultipleComponent]

保证该脚本在脚本对象上的唯一性，用于 MonoBehaviour 或其子类，如果物体上已经有了这个组件，再次添加就会弹窗提示，并且该次添加的操作将被无效化。

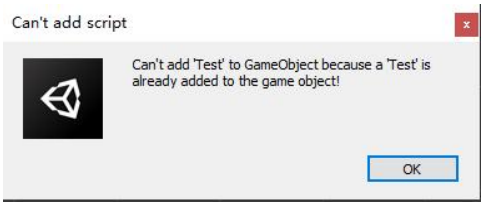

图 15.　使用 DisallowMultipleComponent 特性后，再次添加组件弹出提示

13) **添加组件到菜单**-[AddComponentMenu]

AddComponentMenu 属性允许您将脚本放置在“组件”菜单中的任何位置，而不仅仅是“组件->脚本”菜单。

14) **在编辑模式下执行**-[ExecuteInEditMode]

使脚本的所有实例在编辑模式下执行。

默认情况下，MonoBehaviours 仅在播放模式下执行。通过添加此属性，MonoBehaviour 的任何实例都将在编辑器处于编辑模式时执行其回调函数。

15) 修改 Color 的配置-[ColorUsage]

可以修改 Color 的配置，是否显示 Alpha 通道，或者是否使用 HDR 模式。

参数位置	描述
1	是否显示透明度 Alpha
2	是否使用 HDR 模式、若为 true，需要下面四个参数
3	最小亮度
4	最大亮度
5	最小曝光
6	最大曝光

16) 脚本右键菜单添加一个自定义方法-[ContextMenu]

给脚本右键菜单添加一个自定义方法，不能是静态的。添加完成后，当我们在脚本组件上点击鼠标右键，就会看到我们添加一个自定义方法，只要点击就会执行。

PS：

```
[ContextMenu("Rest")]
private void RestScript()
{
    Debug.LogError("RestScript");
}
```

在上面的代码中，我们使用了 ContextMenu 特性。ContextMenu 后面所跟的字符串，就是我们在脚本组件上点击鼠标右键时，在菜单中所能看到的选项名称。名为 Rest 的选项，指向了代码中的 RestScript 方法，我们点击 Rest 选项就会在 Console 窗口中看到打印。

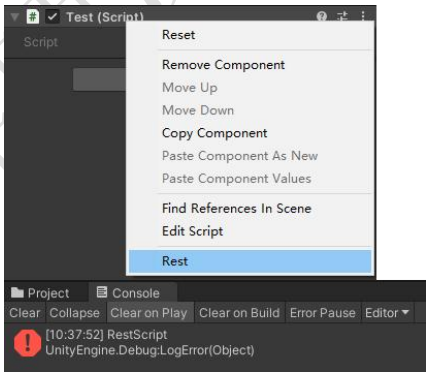

图 16.　使用 ContextMenu 特性

17) 给字段右键菜单添加一个自定义方法-[ContextMenuItem]

ContextMenuItem 特性是针对字段的一个特性，在一个字段上面使用该特性后，在 Inspector 窗口中找到脚本组件里对应字段的位置。当我们将鼠标移动到这个字段上时，我们就可以点击鼠标右键，以使用这个字段所对应的选项菜单。

ContextMenuItem 的第一个参数是选项菜单中的名字，第二个参数是点击选项后所调用的方法名称。

PS：

```
[ContextMenuItem("RestValue", "RestTestValue")]
public int testValue = 1;
```

```
private void RestTestValue()
{
    testValue = 1;
}
```

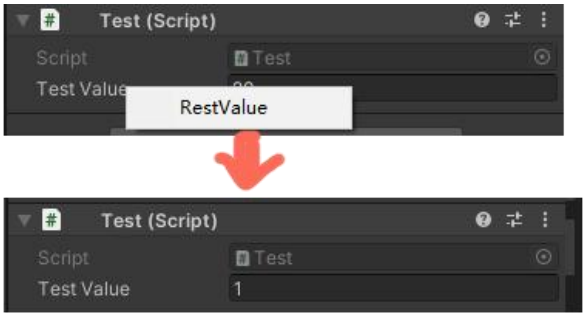

图 17.　使用 ContextMenuItem 特性

如上图所示，修改后的 TestValue 字段值为 20，我们可以直接在这个字段上单击鼠标右键，调用 Rest 方法来重置该字段的值。

18) 延迟变动-[Delayed]

用于 float、int、或 string 变量，当我们修改这三种变量的值时，不再是立即响应改变，而是只有按了回车或鼠标离开字段才会返回新值。

PS：

```
[Delayed]
public int testValue = 1;
[Delayed]
public float testFloat = 2.2f;
[Delayed]
public string testString = "我带你学";
```

Delayed 特性，对于 int 和 float 类型的变量影响较为直观。如果我们没有对 int 和 float 变量应用 Delayed 特性，那么我们是可以直接在字段上左右拖动来改变值。一旦应用了 Delayed 特性，我们就不再能拖动，只有采用输入的方式才能改变值。

19) 帮助链接-[HelpURL]

给类提供一个自定义的帮助 URL。一旦脚本组件使用了该特性，我们在 Inspector 窗口点击问号图标时就会调整到这个链接地址。

PS：

```
[HelpURL("https://www.bilibili.com/")]
public class Test : MonoBehaviour
{

}
```

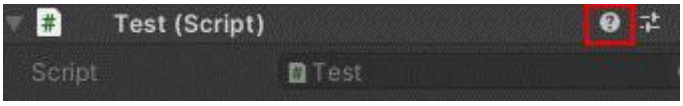

图 18.　点击问号图标，跳转到帮助 URL

20) 运行时自动调用方法-[RuntimeInitializeOnLoadMethod]

不用作为组件添加到对象当中，也可以在运行游戏时直接自动调用方法。要求方法必须

为静态的，类、方法可以为私有。当游戏运行时就会立即调用，但当有多个这种特性的方法进行调用时，其调用顺序是不能确定的。

21) 是否选中父节点-[SelectionBase]

带这个特性的 GameObject，如果点击本身就一定是选中本身，即便父对象也有这特性；如果子对象没有带这个特性，则在场景中点击子对象时，选中的是带特性的父对象；如果父对象和父父对象都有这特性，选中子对象的父节点。

下面我们就将用具体的案例来说明这些特性。首先创建一个测试用的脚本，这个脚本将使用 SelectionBase 特性。

PS：

```
[SelectionBase]
public class Test : MonoBehaviour
{

}
```

然后我们在场景当中创建一些长方体，它们都以圆为父节点。下图名称中有 HaveAttribute 的，表示该对象挂载了我们前面创建的测试用脚本的。

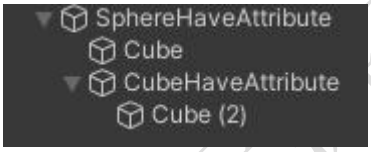

图 19.　Hierarchy 窗口的层级结构

当我们首次选择球中的对象时，我们就可以看到 SelectionBase 特性的作用了。

我们首先用鼠标选择 Cube，这时会在 Scene 窗口中看到实际选中的是球。

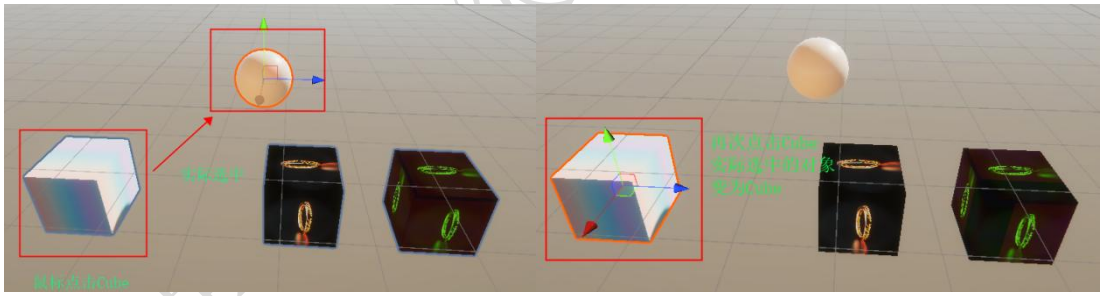

图 20.　第一次实际选中球，第二次选中正确对象

下面我们再首次点击 CubeHaveAttribute，因为这个对象具有 SelectionBase 特性，当我们选择它时就会正确的选择它本身。

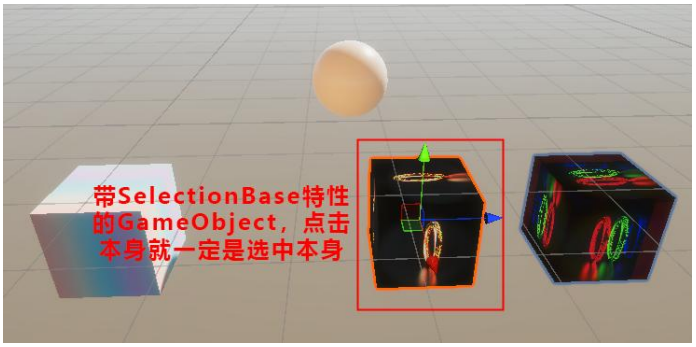

图 21.　带 SelectionBase 特性的 GameObject，点击本身就一定是选中本身

如果我们首次选中的是 Cube (2)这个对象，那么实际选中的对象将是 CubeHaveAttribute，

只有再次点击 Cube (2)对象才会选中它本身。

3. UnityEditor 命名空间

1) 启动时运行编辑器脚本代码-[InitializeOnLoad]

有时，为了能够在 Unity 启动后立即在项目中运行一些编辑器脚本代码，而无需用户执行操作。我们可以通过将 InitializeOnLoad 特性应用于具有静态构造函数的类来执行此操作。该静态构造函数是与类同名的函数，声明为静态且没有返回类型或参数。

PS：

```
using UnityEngine;
using UnityEditor;

[InitializeOnLoad]
public class Startup {
    static Startup()
    {
        Debug.Log("Up and running");
    }
}
```

静态构造函数始终保证在使用类的任何静态函数或实例之前被调用，但 InitializeOnLoad 属性可确保在编辑器启动时调用它。

如何使用这种技术的一个例子是在编辑器中设置一个常规回调（就像 Mono 中的帧更新）。一个类具有一个名为 update 的委托（不是 MonoBehaviour 中的 Update），该委托在编辑器运行时每秒调用多次。要在项目启动时启用此委托，可以使用如下代码。

PS：

```
using UnityEditor;
using UnityEngine;

[InitializeOnLoad]
class MyClass
{
    static MyClass ()
    {
        EditorApplication.update += Update;
    }

    static void Update ()
    {
        Debug.Log("Updating");
    }
}
```

如果存在多个被 InitializeOnLoad 特性修饰的类，则所有的构造方法都会执行一遍。

2) 加载时初始化编辑器类方法-[InitializeOnLoadMethod]

第一次打开 Unity 编辑器运行一次，之后每次进入 Play 模式都运行一次。

3) 可同时编辑多个对象-[CanEditMultipleObjects]

该特性可以使自定义编辑器支持同时编辑多个对象，一般配合 CustomEditor 类使用。

PS：

```
using UnityEngine;
using UnityEditor;

public class Test : MonoBehaviour
{
    public int value = 0;
    public string name;
}

[CanEditMultipleObjects]
[CustomEditor(typeof(Test))]
public class TestEditor : Editor
{
    public override void OnInspectorGUI()
    {
        Test test = (Test)target;
        test.value = EditorGUILayout.IntField(test.value);
        Debug.LogError($"这里是修改后的数值{test.value}");
    }
}
```

上面的代码是编写的一个 MonoBehaviour 和脚本所对应的自定义编辑界面。如果我们将这个脚本挂载在多个空对象上，那么我们就可以同时编辑这多个对象。

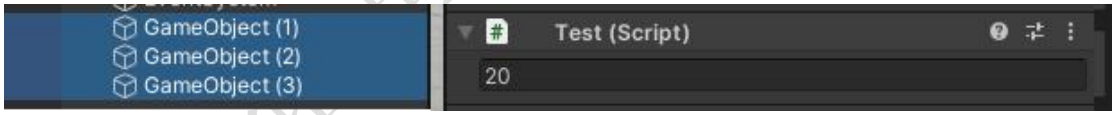

图 1. 使用 CanEditMultipleObjects 特性，自定义编辑器就支持同时编辑多个

如果我们将代码中的 CanEditMultipleObjects 特性注释掉，当我们选择编辑多个对象时，Unity 就会在 Inspector 窗口中提示我们不支持同时编辑多个。

图 2. 注释掉 CanEditMultipleObjects 特性，自定义编辑器就不支持同时编辑多个

4) 自定义编辑器-[CustomEditor]

如果我们想要在 Unity 中自定义编辑器，那么我们就需要 CustomEditor 特性。凡是使用了 CustomEditor 特性的脚本，都需要存放在 Editor 文件夹下。

5) 添加菜单项-[MenuItem]

使用 MenuItem 特性可以添加菜单项，必须是静态方法。第一个参数是菜单的路径；第二个参数是 true 则是给该菜单项添加验证，需分别标记两个函数，true 标记的函数作为 false 标记的函数能否启用并执行的验证，菜单名，优先级要相同；第三个参数就是菜单项的显示优先级，数值越小显示越靠前。

PS：

```
[MenuItem("Tools/游戏/游戏辅助工具", false, 101)]
        static void Init()
        {
            GameHelperEditorWindow window =
(GameHelperEditorWindow)EditorWindow.GetWindow(typeof(GameHelperEditorWindow),
false, "游戏辅助工具");
            window.Show();
        }
```

6) DrawGizmo

作用于静态方法，使任意组件支持 gizmo 渲染方法。

a. GizmoType

Pickable	在编辑器中 gizmo 可以被点选
NotInSelectionHierarchy	如果 Gizmo 未被选中并且也没有选择父项/祖先，则绘制 Gizmo
NonSelected	如果未选中 Gizmo，则绘制它
Selected	如果 Gizmo 被选中，则绘制它
Active	如果激活，则绘制
InSelectionHierarchy	如果 gizmo 被选中或者它是所选的子/后代，则绘制 gizmo。

4. UnityEngine.Scripting 命名空间

1) 禁止剥离代码-[Preserve]

当我们创建编译版本时，Unity 将尝试从项目中剥离那些未使用的代码。这有利于获取小型编译版本。但有时，即使某些代码看起来未曾使用，我们也不希望将其剥离出去。（如使用反射来调用一个方法或者实例化某个类的对象时）

我们可以将 Preserve]特性应用于类、方法、字段以及属性，这将会防止这些代码在编译时被剥离出去。

五. VisualStudio

//TODO: (未实现）……

//UNDONE:（没有做完）……

//HACK:（修改）……

等到再次打开 VS 的时候，找到 ：视图>任务列表,即可显示所有带有 TODO 注释的代码位置。

六. 资源网站

https://itch.io/

3d 模型和动画：https://www.mixamo.com/#/

https://opengameart.org/

https://www.spriters-resource.com/

Shader 学习：

https://learn.jettelly.com/course/unity-shader-bible/usb-chapter-1/forward-rendering/

模型资源下载：https://sketchfab.com/feed

特效纹理制作：http://mebiusbox.github.io/contents/EffectTextureMaker/

材质球/贴图获取：https://www.textures.com/

天空球获取：https://polyhaven.com/hdris

贴图获取：https://ambientcg.com/list

杂项资源获取：https://www.iiicg.com/

MatCap 获取：

① http://pixologic.com/zbrush/downloadcenter/library/#prettyPhoto

② https://www.google.com.hk/search?q=MatCap&newwindow=1&safe=strict&hl=zh-CN&biw=1575&bih=833&tbm=isch&tbo=u&source=univ&sa=X&ved=0ahUKEwju8JDTpZnSAhUGn5QKHawODTIQsAQIIg

③ https://www.pinterest.com/evayali/matcap/

七. 切片图集(Slice Atlas)

要在 unity 中对图集进行切片，需要下载 Unity 2D Sprite 相关组件。

1. 下载

如果你的项目中没有 Package Manager 界面，可以在 Unity 中找到 Window 下的 Package Manager，将 package Manager 界面显示出来。

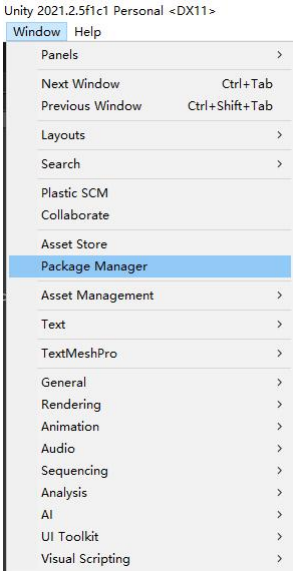

图 3.　　图 9.1 打开 PackageManager

打开了 package Manager 界面后，在 Unity Package 界面搜索 2D Sprite（在 Unity Registry 大类中）,找到并点击 Install 将其导入到 Unity 中。

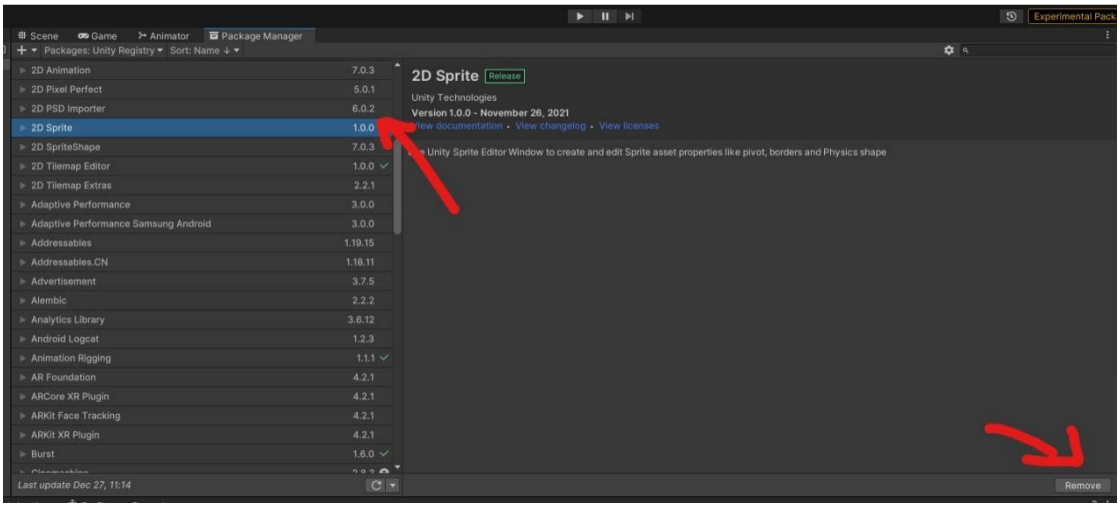

图 4.　　图 9.2 将 2D Sprite 导入

要将一张图集进行切片，只需要将图集的 SpriteMode 设置为 Multiple，然后进行切片。

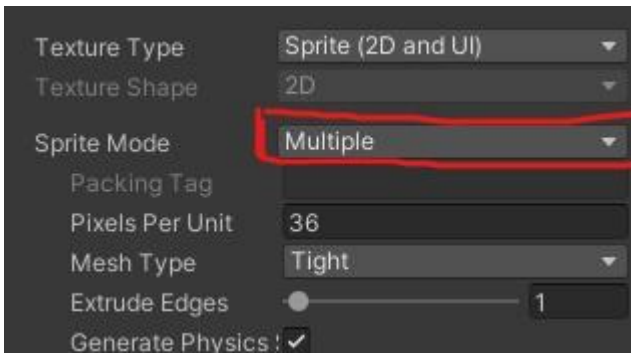

图 5.　　图 9.3 设置精灵图为多图模式

2. 切片类型

1) Automatic-自动

让 Unity 通过基于每个潜在精灵的周围透明度生成边界来自动将纹理切片为多个单独的精灵。

2) Grid by Cell Size-按单元格大小

Pixel Size 的 X 值和 Y 值分别确定切分的像素瓦片的高度和宽度。

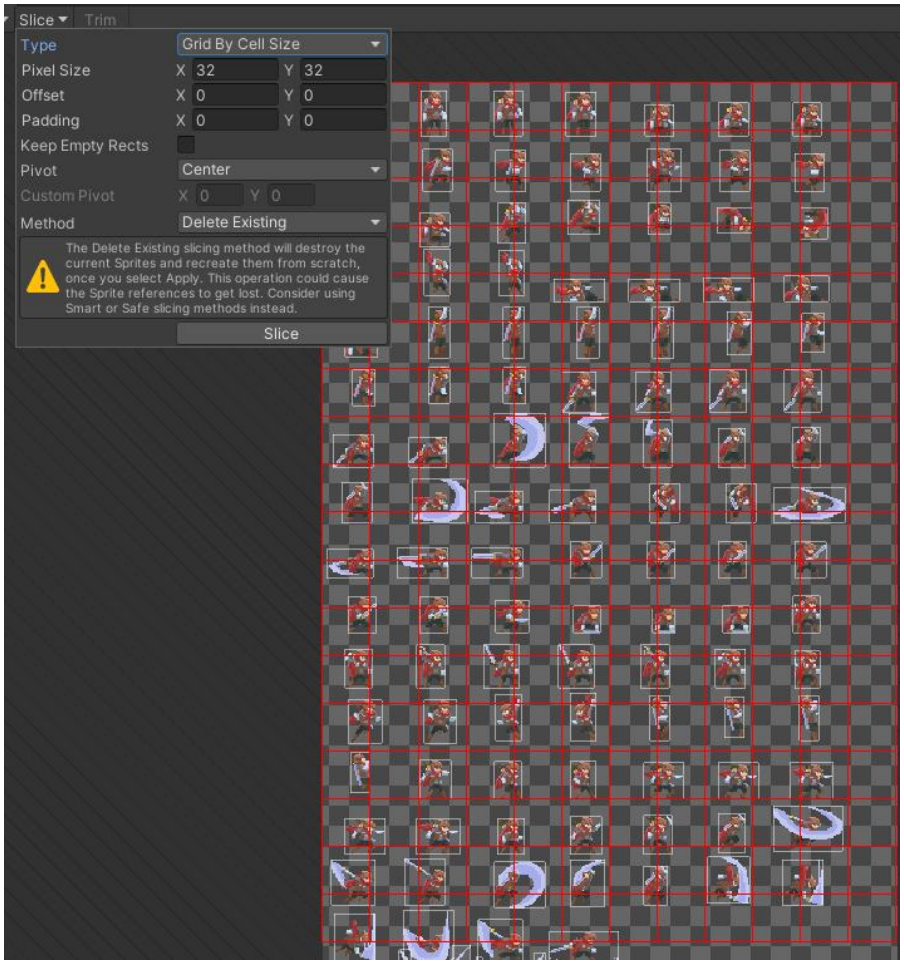

图 6.　　图 9.4 按单元格大小切片

3) Grid by Cell Count-按单元格数量

C 即 Column（列）,R 即 Row（行）；它们确定切分的列数和行数。

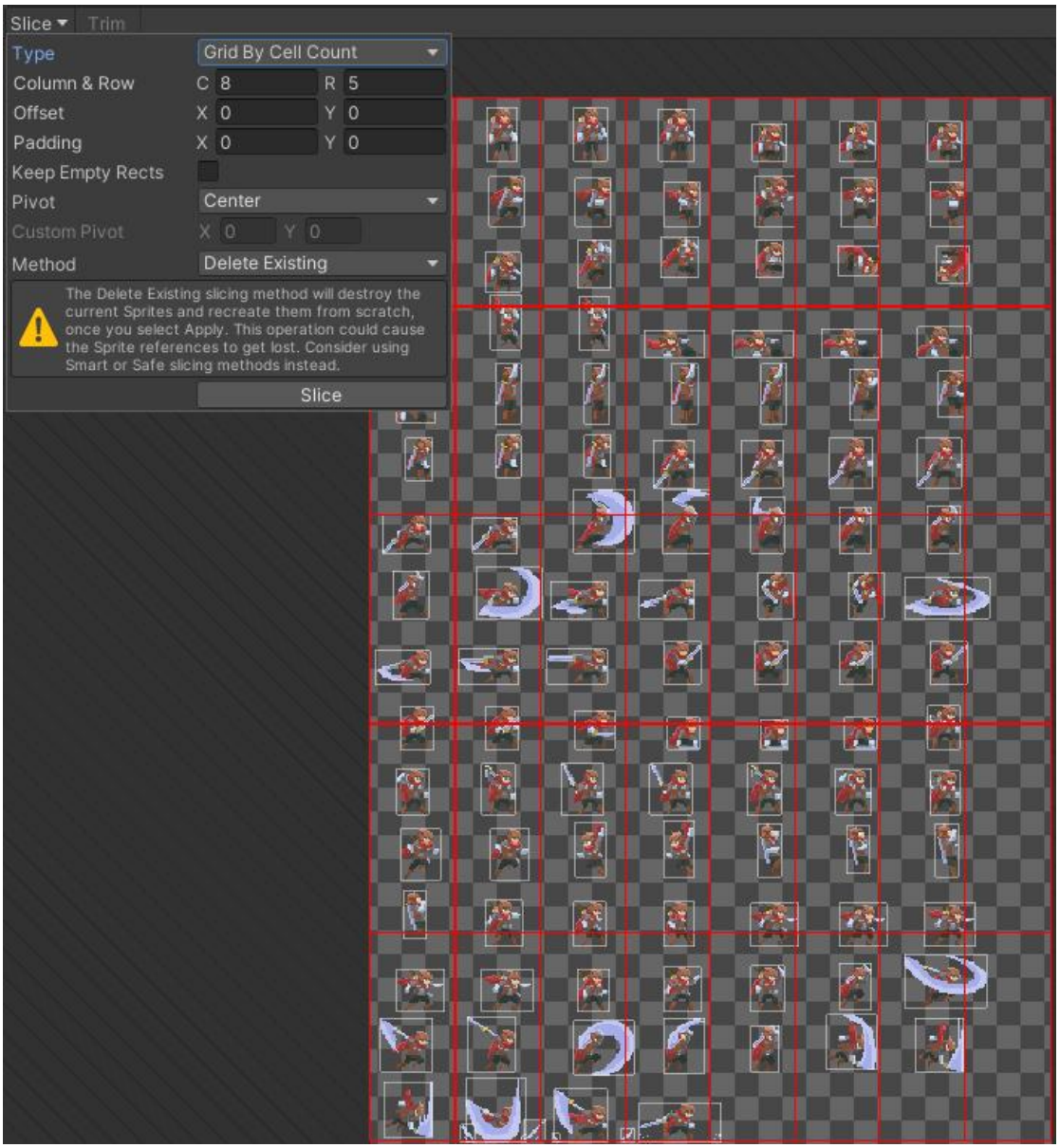

图 7.　　图 9.5 按单元格数量切片

八. 未运行游戏的状态监视 ShaderGraph

在未运行游戏的情况下，想要观察 ShaderGraph 的效果，一种方法是在 Scene 窗口持续按住鼠标右键。

另一种方法是开启 Unity Scene 窗口的 Always Refesh，得到实时刷新的效果。

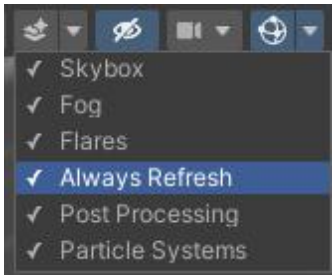

图 8.　　保持实时刷新

九. 组件 Preset

Preset 允许用户存储该类型组件的预设，减少用户重复调整参数所耗费的时间。

点击组件右上角的工具图标，然后点击 Save Current 即可将该组件的数据保存为一个预设。

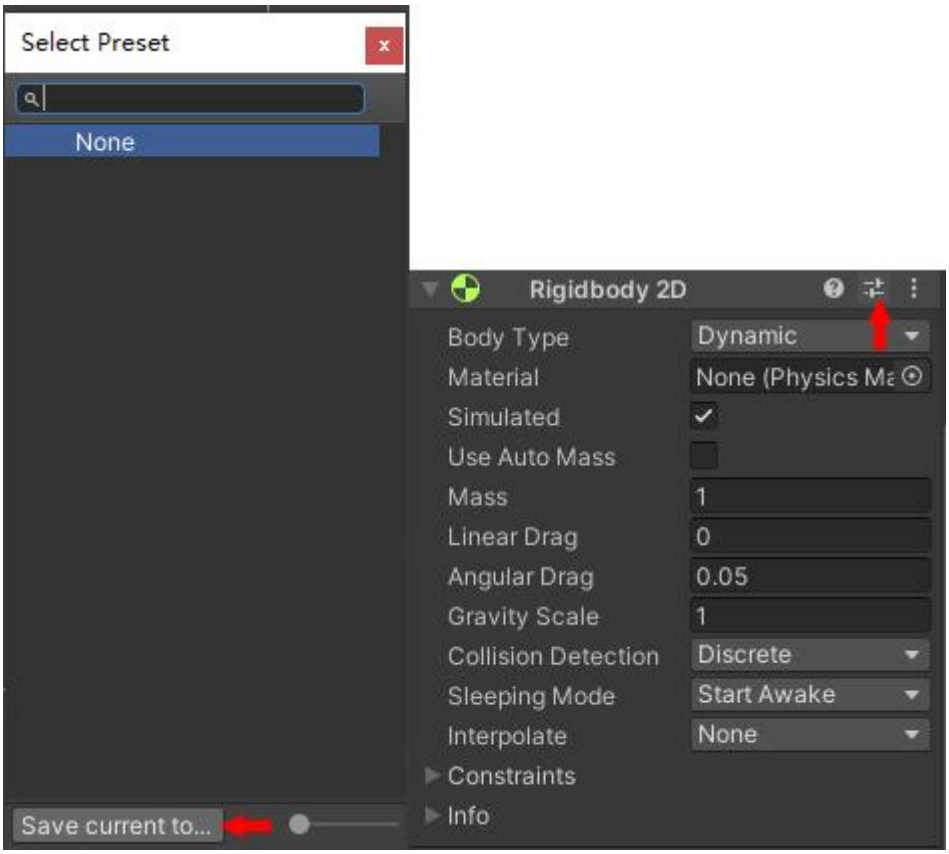

图 9.　　组件预设

十. 调整运行窗口颜色

我们可以更改 Unity 处于运行状态下的颜色，这可以帮助我们更容易的知道当前是否处于运行状态。

我们在 Edit->Preferences->Colors，找到 Playmode Tint，修改其颜色即可。

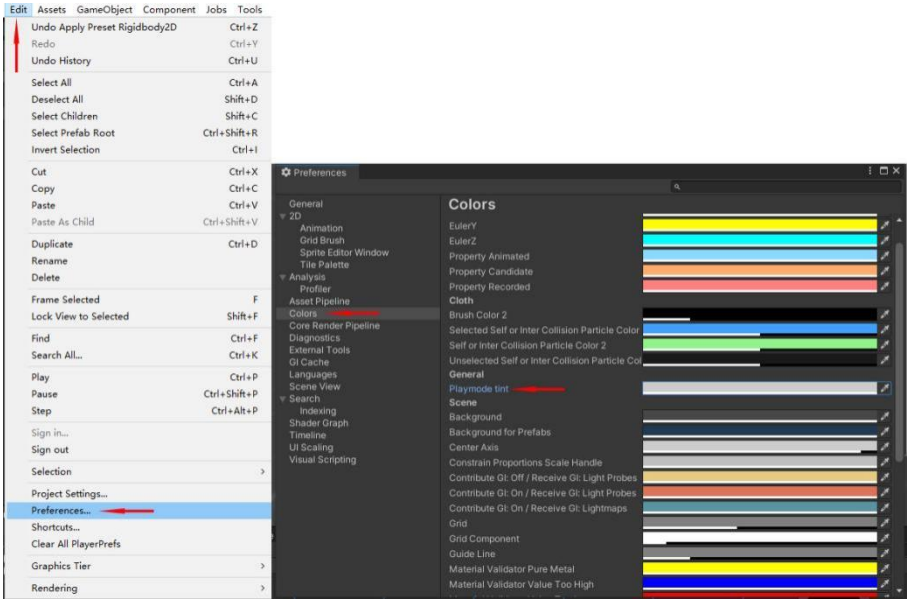

图 10.　　修改 Playmode 颜色

十一. 修改鼠标样式和启动图像

在 Unity 中找到 Edit->Project Settings->Player

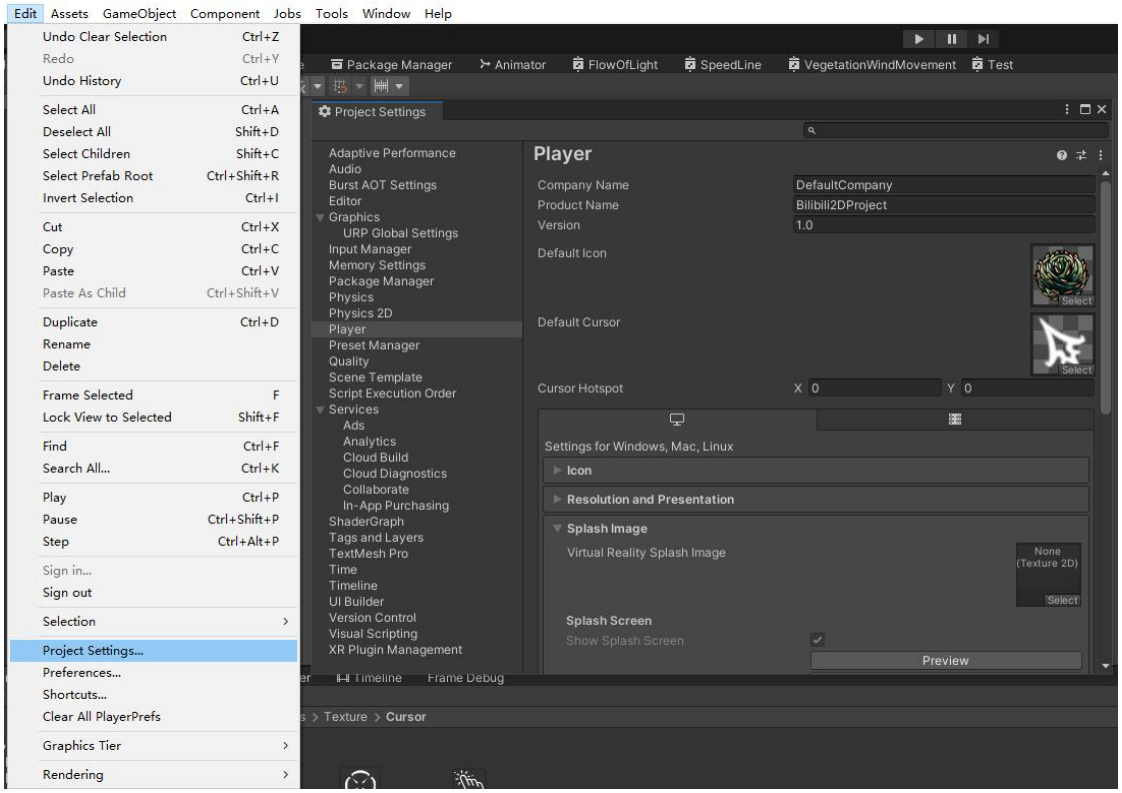

图 11. Payer 设置界面

1. 设置鼠标样式

属性	功能描述
公司名称-Company Name	输入您的公司名称。
产品名称-Product Name	输入应用程序运行时出现在菜单栏上的名称。Unity 也使用它来定位首选项文件。
版本-Version	输入您的应用程序的版本号。
默认图标样式-Default Icon	选择要用作每个平台上应用程序默认图标的 Texture 2D 文件。
默认鼠标样式-Default Cursor	设置默认鼠标样式
Cursor Hotspot	设置鼠标作用点的偏移量,默认左上角为原点。

注意，要将纹理设置为鼠标样式，该纹理在 Unity 中的类型应为 Cursor。

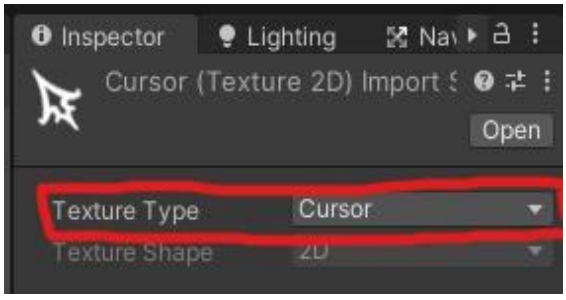

图 12. 鼠标样式的纹理类型

2. 设置启动图像

在 Player 设置中找到 Splash Image 设置。

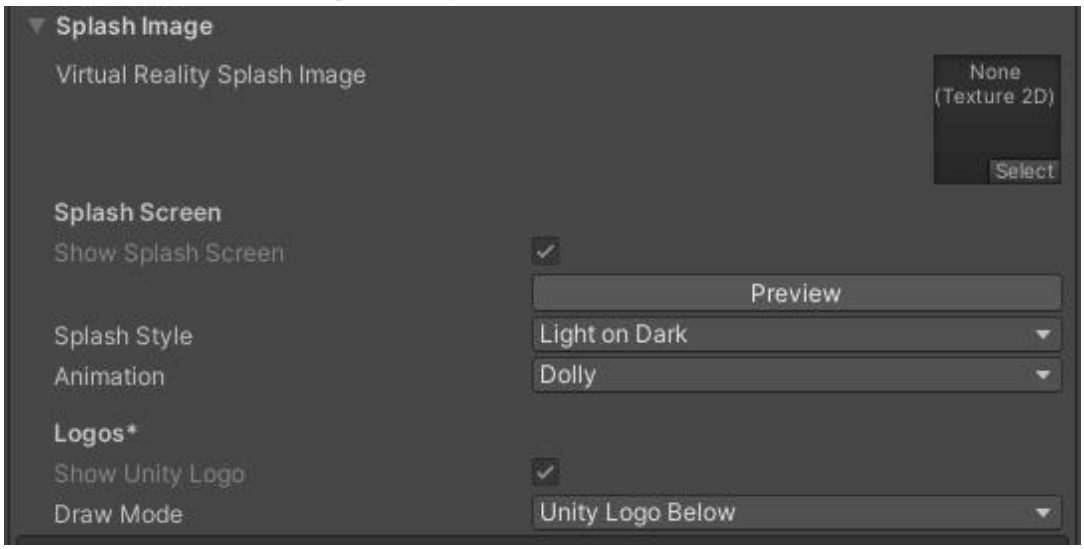

图 13. Splash Image

设置名称		描述
Virtual Reality Splash Image		设置要用于使用虚拟现实的应用程序的图像。图片必须是 2D Texture .
显示启动画面-Show Splash Screen		默认启用。禁用此设置以在应用程序启动时不显示程序启动画面。如果是 Unity 个人版，则无法禁用此设置。
预览-Preview		选择预览按钮以在游戏视图中查看启动画面的预览。
动画类型-Splash Style		Unity logo 的颜色样式设置。
	Light on Dark	会使用浅色版本的 Unity Logo。这适合在深色背景上显示，这是默认设置。
	Dark on Light	选择 Dark on Light Unity ，会使用深色版本的 Unity Logo，最适合在浅色背景上显示。
Animation		设置启动画面如何在屏幕上出现和消失。
	Static	不应用任何动画。
	Dolly	Logo 和背景会有缩放效果。这是默认设置。
自定义-Custom		选择此设置可自定义背景和 Logo 缩放量。
	徽标缩放-Logo Zoom	当 Logo 到达 Logo 动画总持续时间的末尾时，Logo 的目标缩放（从 0 到 1）。

	背景缩放-Background Zoom	当背景到达初始屏幕动画的总持续时间结束时的目标缩放（从 0 到 1）。
Show Unity Logo		默认启用。禁用此选项不会在初始屏幕上显示 Unity 徽标。如果有是 Unity 个人版，则无法禁用此设置。
绘图模式-Draw Mode		选择 Logo 出现在初始屏幕上的顺序模式
	Unity Logo Below	Unity 的 Logo 将显示在其他 Logo 之下。
	All Sequential	依次显示 Unity Logo 以及 Logo 列表中的所有 Logo。
	Logo Duration	设置每个 Logo 出现在屏幕上的时间长度。值在 2 到 10 秒之间。
叠加不透明度-Overlay Opacity		调整覆盖不透明度设置的值以使 Logo 突出。这会影响 Logo 的背景颜色和图像颜色，具体取决于您设置 Splash 样式（Light on Dark 或 Dark on Light）的方式。 将不透明度设置为较低的值可以以减少此效果。也可以通过将其设置为 0 来禁用该效果。例如，如果 Splash Style 为 Light on Dark，具有白色背景，如果将 Overlay Opacity 设置为 1，则背景变为灰色，如果设置为 0，则背景变为白色。 如果是 Unity 个人版，此设置的最小值为 0.5。
模糊背景图像-Blur Background Image		启用此设置以模糊您设置的背景图像。如果禁用此设置，它将显示没有模糊效果的背景图像。
背景图像-Background Image		设置对 Sprite 图像以用作背景。Unity 会调整背景图像以填充屏幕。

十二. 图片跟随文本自适应

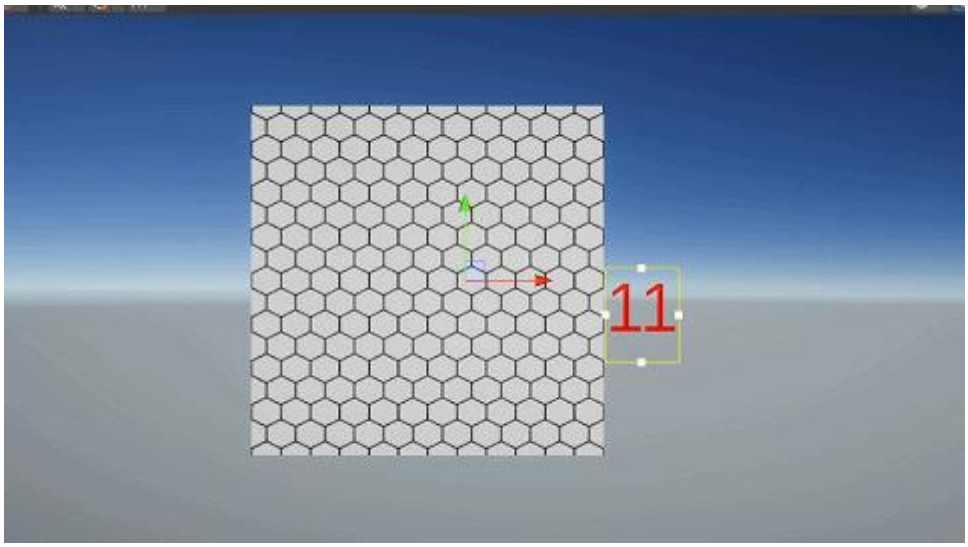

图 14. 图像跟随文本移动（这是一个 gif 图，可另存为查看）

在 Unity 中我们可能经常会遇到一种需求，就是让图像跟随着文本进行移动。

要实现这种效果，我们可以使用代码去实现，但这会麻烦许多。我们就可以考虑使用 Unity 中的内置组件，也就是 Content Size Fitter 组件。

首先我们需要创建一个 Image 和一个 Text，如下图所示。

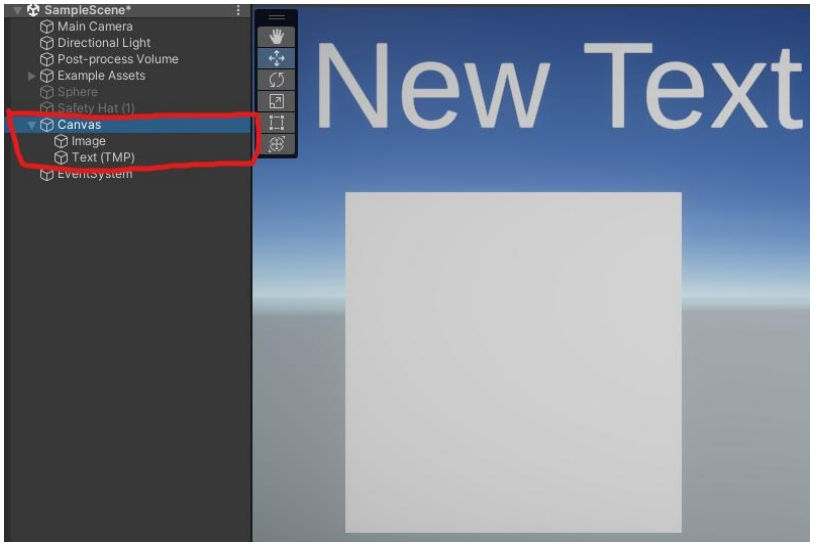

图 15. 创建图像和 Text 文本（我这里使用的 Text Mesh Pro）

下面就可以让 Image 成为 Text 的子物体，接着为 Text 添加一个 Content Size Fitter 组件，并设置 Horizontal Fit 为 Preferred[1] Size。

[1] adj. 更合意的；优先（考虑）的；被偏爱的 v. 偏爱；宁可（prefer 的过去式和过去分词）

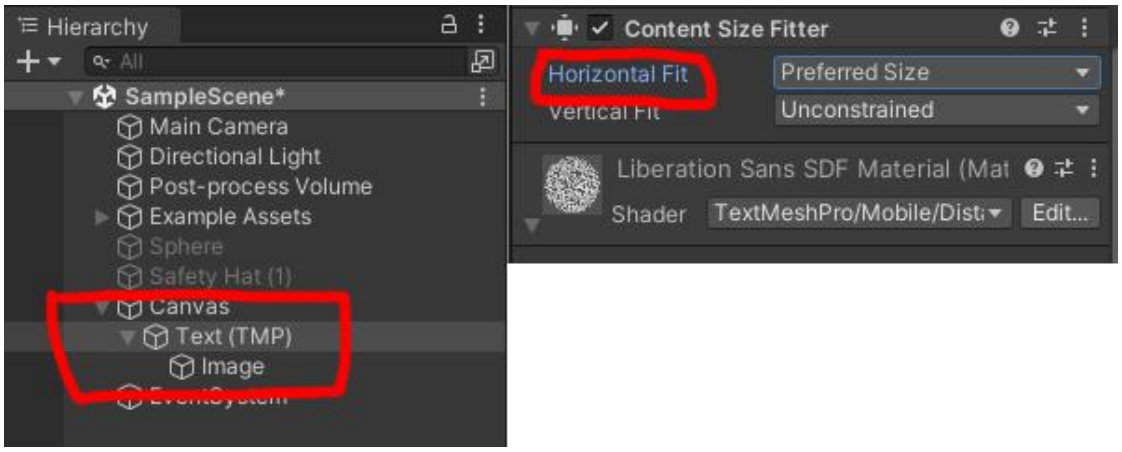

图 16.　让 Image 成为 Text 的子物体

最后我们需要设置 Image 的 Pivot 位置。如果将 pivot 设置在左边，它就会以 Text 的左边为基准进行移动；将 pivot 设置在右边，它就会以 Text 的右边为基准进行移动。

十三.　修改 Unity 的脚本模板

在 Unity 中新建脚本时，我们新建的脚本里往往就已经有了些基础的内容，这就是通过脚本模板去实现的。但是，在实际使用的过程中，我们往往会删除 Unity 默认构建的这些内容。

下面我们就需要去修改脚本模板，去自定义新建脚本里的默认内容。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
      Unity 脚本|0 个引用
5   public class NewBehaviourScript : MonoBehaviour
6   {
7       // Start is called before the first frame update
          Unity 消息|0 个引用
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
          Unity 消息|0 个引用
14      void Update()
15      {
16
17      }
18  }
19
```

图 17.　新建 C#脚本时的默认内容

我们找到存放 Unity 引擎的根目录，引擎版本->Editor->Data->Resources->ScriptTemplates[2]。在 ScriptTemplates 里找到 NewBehaviourScript，这个就是我们新建 C#脚本时的内容模板，我们只需要修改它里面的内容即可。

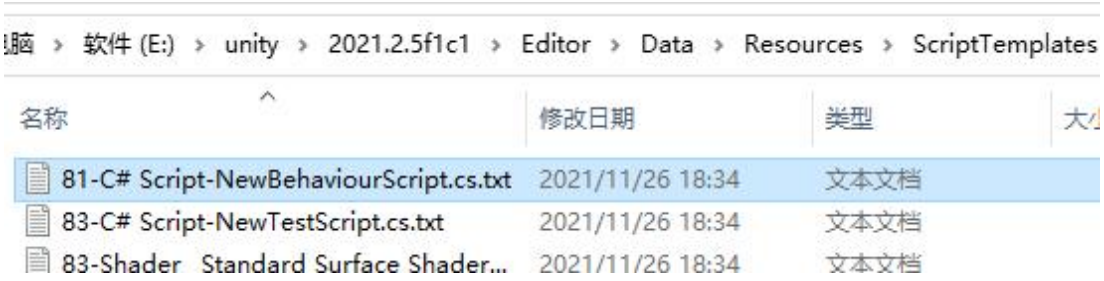

图 18. Unity C#模板文件

我们可以看到，这个目录文件下还有其他的模板，就拿 Shader 的模板举例吧。在 Build In 渲染管线下，Unity 新建的默认 Shader 脚本，其包含的基础内容符合我们的要求，我们不需要进行大量的更改。

但是在 URP 和 HDRP 渲染管线下，我们就需要修改 Shader 脚本里的内容了，因为 HDRP 和 URP 是使用的 HLSL 语言，不是 CG 语言。我们所需要引入的相关内容包也不相同，且我们需要告知 Shader 使用的是哪一个渲染管线，并删除那些不适用于 URP 和 HDRP 的内容。

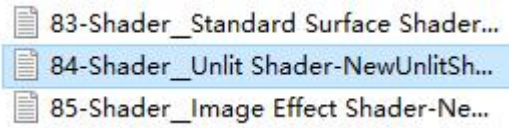

图 19. 修改 Unlit Shader 的模板内容以适应 URP 和 HDRP

首先先将 CG 语言修改为 HLSL 语言，再在标签里添加"RenderPipeline"并使其等于"UniversalRenderPipeline"。

为了使我们的着色器能够完美编译并使其过程更加高效，我们必须包含依赖项“Core.hlsl”，Core.hlsl 本身取代了“UnityCg.cginc”。

Core.hlsl 必须编写其位置的完整路径，这个路径可以自己去查找，就在当前项目的 ShaderLibrary 文件夹下。

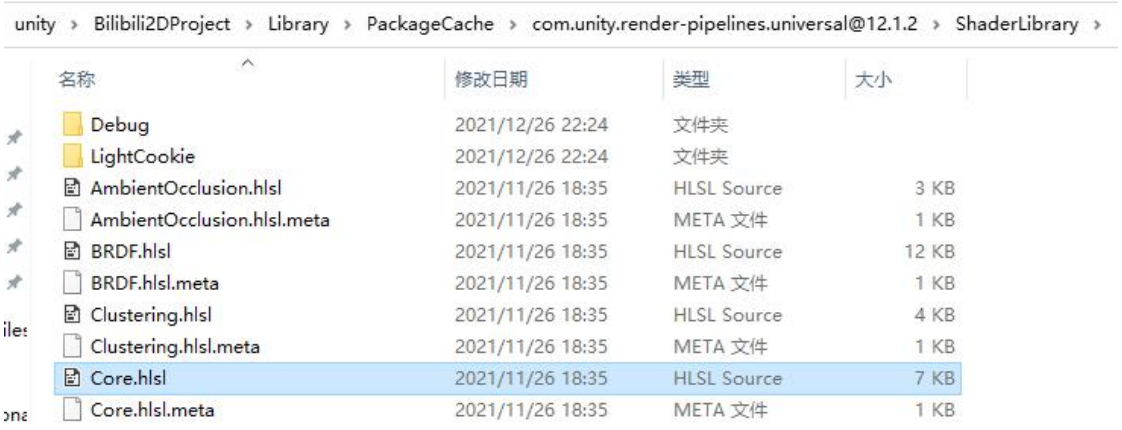

图 20. Core.hlsl 路径

一旦我们替换了这些依赖项，GPU 将无法编译雾坐标，我们需要删除这部分内容。同时也无法编译 UnityObjectToClipPos 函数，我们将不得不使用包含在“Core.hlsl”中的 TransformObjectToHClip 函数去代替。

[2] n. 模板；范本（template 的复数）；属性单元

如果此时我们保存并返回 Unity 创建一个新的 Shader，我们可以看到在控制台中生成了“无法识别的标识符”类型的错误，这是什么原因呢？请记住，默认情况下，片元着色器阶段的 col 向量都是 fixed4 类型。

Universal RP 不能编译“fixed”类型的变量或向量，我们将不得不使用“half 或 float”。

我们可以手动替换该类型的变量和向量。或者包含“HLSLSupport.cginc”依赖项，为着色器编译添加辅助宏和跨平台定义。

一旦包含了这种依赖关系，我们就可以使用 fixed 类型的变量和向量，因为现在我们的程序将根据其精度识别这种类型的数据，并自动将它们替换为 half 或 float。

十四. 封装 Debug. Log

在 Unity 中，我们是可以使用 Debug.Log 来打印信息的，但是在实际打包运行时却不需要这部分的内容。因此，我们就可以将这个方法封装一次，并为这个方法添加上一个条件编译特性，这样在打包真机调试时为了节省性能就可以快速的关闭日志。

Unity 中最常见的使用条件编译的情况就是分平台编译不同的代码片段。由于 Unity 是跨平台的，不同平台有不同的特性，我们经常能见到 #if...#endif 来做条件编译的代码块。只有在 Unity 编辑器中开启了指定的宏命令，这个宏命令所对应的代码块才会被编译。

在 Unity 的 Project Setting->Player 里，我们可以找到 Script Compliation。在 Script Compliation 里，我们是可以添加自定义的标记符号。当 Script Compliation 里有对应的预定义符号时，对应的代码块才会被编译，如果去除就不会编译对应的代码块。

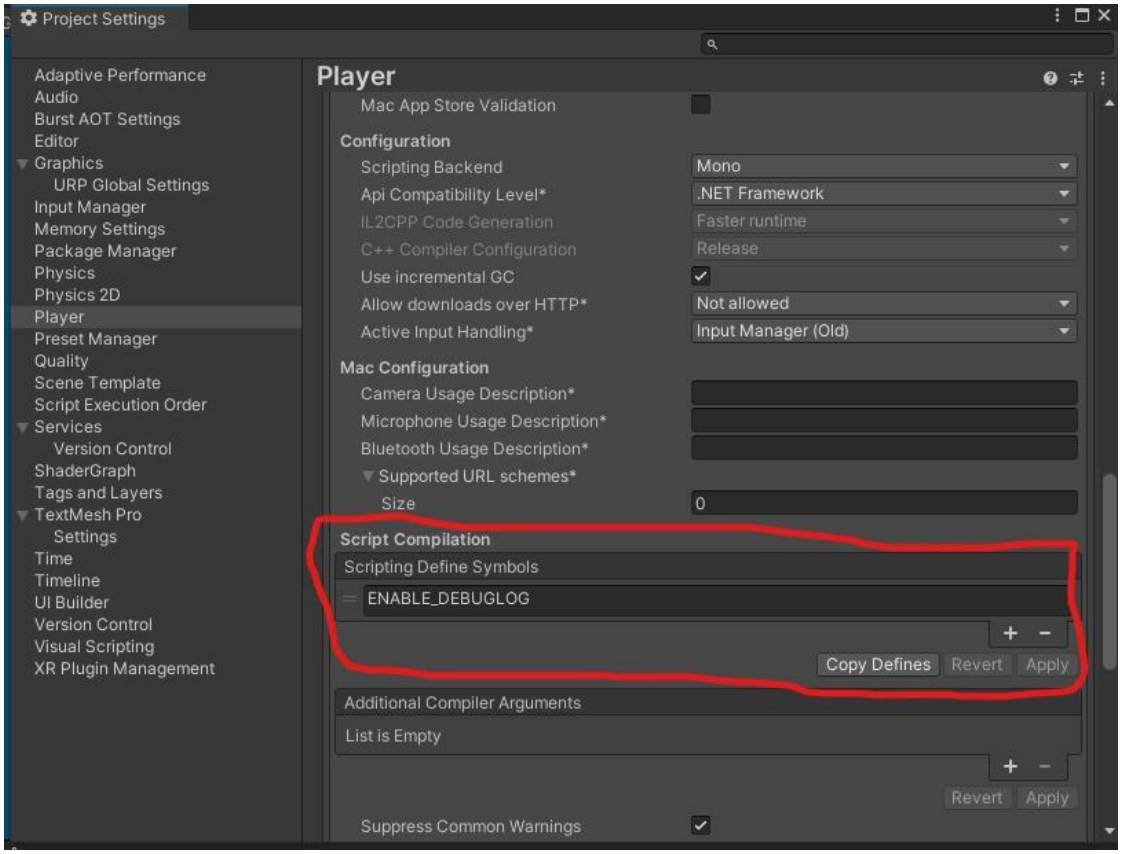

图 21. 预定义符号集

注意！Conditional 特性只能用于无返回值，无 out 形参的方法。

PS：

```
using UnityEngine;
using Debug = UnityEngine.Debug;
```

```
using System.Diagnostics;

namespace Log
{
    public static class Logger
    {
        [Conditional("ENABLE_DEBUGLOG")]
        public static void Log(string content)
        {
            Debug.Log(content);
        }
    }

    public class ConditionLog : MonoBehaviour
    {
        private void Start()
        {
            Debug.Log("这是一条打印信息");
            Logger.Log("666666666");
        }
    }
}
```

因为我们前面在 Script Compliation 里包含了 ENABLE_DEBUGLOG 这个符号，因此其对应的静态方法将可以使用。

十五. Unity 中隐藏 UI 界面解决方案

1. 通用方案

1) SetActive

优点：方便快捷。

缺点：SetActive(false)的物体上面挂载的脚本也不运行了，而很多时候我们需要那个脚本运行。这样就不能使用 SetActive(true)或者 SetActive(false)。如果把自己的引用给另外一个脚本来对自己 SetActive(true)或者 SetActive(false)，又会造成多余的耦合，并不符合 OOP 设计理念。

2) Disable

优点：方便快捷。

缺点：Disable 掉的物体，重新 enable=true，会造成较大的性能消耗，如果此界面 draw call 较多，会有明显的延迟。

3) 更改 Scale

Scale 改为(0,0,0)，再改为(1,1,1)。

缺点：改回后 draw call 加倍，产生大量垃圾回收。

4) 将界面移除 Canvas 这个父物体

缺点：改回后 draw call 加倍。大量垃圾回收。而且新增父物体增加额外引用耦合。

5) 放在 Camera 的某个 culling 层上

缺点：改回后 draw call 加倍。大量垃圾回收。只对 screen space-camera 有效。

6) Canvas.enable = false

缺点：改回后延迟严重。而且不方便使用。

2. 最优解决方案

给 Panel 加一个 CanvasGroup，控制 CanvasGroup 组件的属性

```
//显示
GetComponent<CanvasGroup>().alpha = 1;
GetComponent<CanvasGroup>().interactable = true;
GetComponent<CanvasGroup>().blocksRaycasts = true;

//隐藏
GetComponent<CanvasGroup>().alpha = 0;
GetComponent<CanvasGroup>().interactable = false;
GetComponent<CanvasGroup>().blocksRaycasts = false;
```

十六. 解决脚本加载顺序矛盾

十七. Unity GC 优化

在游戏运行的时候，数据主要存储在内存中。游戏的数据在不需要的时候，当前存储数据的内存就可以被回收以再次使用。内存垃圾是指当前废弃数据所占用的内存，垃圾回收（GC）是指将废弃的内存重新回收再次使用的过程。

Unity 中将垃圾回收当作内存管理的一部分，如果游戏中废弃数据占用内存较大，则游戏的性能会受到极大影响，此时垃圾回收会成为游戏性能的一大障碍点。

1. Unity 内存管理机制简介

要想了解垃圾回收如何工作以及何时被触发，我们首先需要了解Unity的内存管理机制。Unity 主要采用自动内存管理的机制，开发时在代码中不需要详细地告诉 Unity 如何进行内存管理，Unity 内部自身会进行内存管理。这和使用 C++开发需要随时管理内存相比，有一定的优势，当然带来的劣势就是需要随时关注内存的增长，不要让游戏在手机上跑“飞”了。

Unity 的自动内存管理可以理解为以下几个部分：

① Unity 内部有两个内存管理池：堆内存和堆栈内存。堆栈内存(stack)主要用来存储较小的和短暂的数据，堆内存(heap)主要用来存储较大的和存储时间较长的数据。

② Unity 中的变量只会在堆栈或者堆内存上进行内存分配，变量要么存储在堆栈内存上，要么处于堆内存上。

③ 只要变量处于激活状态，则其占用的内存会被标记为使用状态，则该部分的内存处于被分配的状态。

④ 一旦变量不再激活，则其所占用的内存不再需要，该部分内存可以被回收到内存池中被再次使用，这样的操作就是内存回收。处于堆栈上的内存回收及其快速，处于堆上的内存并不是及时回收的，此时其对应的内存依然会被标记为使用状态。

⑤ 垃圾回收主要是指堆上的内存分配和回收，Unity 中会定时对堆内存进行 GC 操作。

2. 堆栈内存分配和回收机制

堆栈上的内存分配和回收十分快捷简单，因为堆栈上只会存储短暂的或者较小的变量。内存分配和回收都会以一种顺序和大小可控制的形式进行。

堆栈的运行方式就像 stack：其本质只是一个数据的集合，数据的进出都以一种固定的

方式运行。正是这种简洁性和固定性使得堆栈的操作十分快捷。当数据被存储在堆栈上的时候，只需要简单地在其后进行扩展。当数据失效的时候，只需要将其从堆栈上移除。

3. 堆内存分配和回收机制

堆内存上的内存分配和存储相对而言更加复杂，主要是堆内存上可以存储短期较小的数据，也可以存储各种类型和大小的数据。其上的内存分配和回收顺序并不可控，可能会要求分配不同大小的内存单元来存储数据。

堆上的变量在存储的时候，主要分为以下几步：

① 首先，Unity 检测是否有足够的闲置内存单元用来存储数据，如果有，则分配对应大小的内存单元。

② 如果没有足够的存储单元，Unity 会触发垃圾回收来释放不再被使用的堆内存。这步操作是一步缓慢的操作，如果垃圾回收后有足够大小的内存单元，则进行内存分配。

③ 如果垃圾回收后并没有足够的内存单元，则 Unity 会扩展堆内存的大小，这步操作会很缓慢，然后分配对应大小的内存单元给变量。

④ 堆内存的分配有可能会变得十分缓慢，特别是在需要垃圾回收和堆内存需要扩展的情况下，通常需要减少这样的操作次数。

4. 垃圾回收时的操作

当堆内存上一个变量不再处于激活状态的时候，其所占用的内存并不会立刻被回收，不再使用的内存只会在 GC 的时候才会被回收。

每次运行 GC 的时候，主要进行下面的操作：

① GC 会检查堆内存上的每个存储变量。

② 对每个变量会检测其引用是否处于激活状态。

③ 如果变量的引用不再处于激活状态，则会被标记为可回收。

④ 被标记的变量会被移除，其所占有的内存会被回收到堆内存上。

GC 操作是一个极其耗费的操作，堆内存上的变量或者引用越多则其运行的操作会更多，耗费的时间越长。

5. 何时会触发垃圾回收

主要有三个操作会触发垃圾回收：

① 在堆内存上进行内存分配操作而内存不够的时候都会触发垃圾回收来利用闲置的内存。

② GC 会自动的触发，不同平台运行频率不一样。

③ GC 可以被强制执行。

特别是在堆内存上进行内存分配时，而内存单元不足够的时候，GC 会被频繁触发。这就意味着频繁在堆内存上进行内存分配和回收会触发频繁的 GC 操作。

6. GC 操作带来的问题

了解了 GC 在 Unity 内存管理中的作用后，我们需要考虑其带来的问题。最明显的问题是 GC 操作会需要大量的时间来运行，如果堆内存上有大量的变量或者引用需要检查，则检查的操作会十分缓慢，这就会使得游戏运行缓慢。其次 GC 可能会在关键时候运行，例如在 CPU 处于游戏的性能运行关键时刻，此时任何一个额外的操作都可能会带来极大的影响，使得游戏帧率下降。

另外一个 GC 带来的问题是堆内存的碎片化。当一个内存单元从堆内存上分配出来，其大小取决于其存储的变量的大小。当该内存被回收到堆内存上的时候，有可能使得堆内存被分割成碎片化的单元。也就是说堆内存总体可以使用的内存单元较大，但是单独的内存单元

较小，在下次内存分配的时候不能找到合适大小的存储单元，这也会触发 GC 操作或者堆内存扩展操作。

堆内存碎片会造成两个结果，一个是游戏占用的内存会越来越大，一个是 GC 会更加频繁地被触发。

7. 分析堆内存的分配

如果 GC 造成游戏的性能问题，我们需要知道游戏中的哪部分代码会造成 GC，内存垃圾会在变量不再激活的时候产生，所以首先我们需要知道堆内存上分配的是什么变量。

在 Unity 中，值类型变量都在堆栈上进行内存分配，其他类型的变量都在堆内存上分配。

下面的代码可以用来理解值类型的分配和释放,其对应的变量在函数调用完后会立即回收。

PS：

```
void ExampleFunciton()
{
    int localInt = 5;
}
```

对应的引用类型的参考代码如下，其对应的变量在 GC 的时候才回收。

PS：

```
void ExampleFunction()
{
    List localList = new List();
}
```

8. 降低 GC 的影响的方法

大体上来说，我们可以通过三种方法来降低 GC 的影响：

① 减少 GC 的运行次数。

② 减少单次 GC 的运行时间。

③ 将 GC 的运行时间延迟，避免在关键时候触发，比如可以在场景加载的时候调用 GC。

似乎看起来很简单，基于此，我们可以采用三种策略：

① 对游戏进行重构，减少堆内存的分配和引用的分配。更少的变量和引用会减少 GC 操作中的检测个数从而提高 GC 的运行效率。

② 降低堆内存分配和回收的频率，尤其是在关键时刻。也就是说更少的事件触发 GC 操作，同时也降低堆内存的碎片化。

③ 我们可以试着测量 GC 和堆内存扩展的时间，使其按照可预测的顺序执行。当然这样操作的难度极大，但是这会大大降低 GC 的影响。

9. 减少内存垃圾的数量

减少内存垃圾主要可以通过一些方法来减少。

1) 缓存数据

如果在代码中反复调用某些造成堆内存分配的函数但是其返回结果并没有使用，这就会造成不必要的内存垃圾，我们可以缓存这些变量来重复利用，这就是缓存。

例如下面的代码每次调用的时候就会造成堆内存分配，主要是每次都会分配一个新的数组。

PS：

```
void OnTriggerEnter(Collider other)
```

```
{
    Renderer[] allRenderers = FindObjectsOfType<Renderer>();
    ExampleFunction(allRenderers);
}
```

而下面的代码只会生产一个数组用来缓存数据，实现反复利用而不需要造成更多的内存垃圾。

PS：

```
private Renderer[] allRenderers;

void Start()
{
    allRenderers = FindObjectsOfType<Renderer>();
}

void OnTriggerEnter(Collider other)
{
    ExampleFunction(allRenderers);
}
```

2) 不要在频繁调用的函数中反复进行堆内存分配

在 MonoBehaviour 中，如果我们需要进行堆内存分配，最坏的情况就是在其反复调用的函数中进行堆内存分配，例如 Update() 和 LateUpdate() 函数这种每帧都调用的函数中，这会产生大量的内存垃圾。我们可以考虑在 Start()或者 Awake()函数中进行内存分配，这样可以减少内存垃圾。

下面的例子中，Update 函数会多次触发内存垃圾的产生。

PS：

```
void Update()
{
    ExampleGarbageGenerationFunction(transform.position.x);
}
```

通过一个简单的改变，我们可以确保每次在 x 坐标发生改变的时候才触发函数调用，避免每帧都进行堆内存分配。

PS：

```
private float previousTransformPositionX;
void Update()
{
    float transformPositionX = transform.position.x;
    if(transfromPositionX != previousTransformPositionX)
    {
        ExampleGarbageGenerationFunction(transformPositionX);
        previousTransformPositionX = trasnformPositionX;
    }
}
```

另外的一种方法是在 Update 中采用计时器，特别是在运行有规律但是不需要每帧都运行的代码中，例如。

PS：

```
void Update()
{
    ExampleGarbageGeneratiingFunction()
}
```

通过添加一个计时器，我们可以确保每隔 1s 才触发该函数一次：

PS：

```
private float timeSinceLastCalled;
private float delay = 1f;
void Update()
{
    timSinceLastCalled += Time.deltaTime;
    if(timeSinceLastCalled > delay)
    {
        ExampleGarbageGenerationFunction();
        timeSinceLastCalled = 0f;
    }
}
```

通过这样细小的改变，我们可以使得代码运行的更快同时减少内存垃圾的产生。

不要忽略这一个方法，对于很多固定时间的事件回调函数，如果每次都分配新的缓存，但是在操作完后又不进行释放，这样就会造成大量的内存垃圾。对于这样的缓存，最好的办法就是当前周期回调后执行清除或者标志为废弃。

3) 清除链表

在堆内存上进行链表的分配的时候，如果该链表需要多次反复的分配，我们可以采用链表的 clear 函数来清空链表从而替代反复多次的创建分配链表。

PS：

```
void Update()
{
    List myList = new List();
    PopulateList(myList);
}
```

通过改进，我们可以将该链表只在第一次创建或者该链表必须重新设置的时候才进行堆内存分配，从而大大减少内存垃圾的产生。

PS：

```
    private List myList = new List();
    void Update()
    {
        myList.Clear();
        PopulateList(myList);
    }
```

4) 对象池

即便我们在代码中尽可能地减少堆内存的分配行为，但是如果游戏有大量的对象需要产生和销毁依然会造成 GC。对象池技术可以通过重复使用对象来降低堆内存的分配和回收频率。对象池在游戏中广泛的使用，特别是在游戏中需要频繁的创建和销毁相同的游戏对象的

时候，例如枪的子弹这种会频繁生成和销毁的对象。

10. 造成不必要的堆内存分配的因素

我们已经知道值类型变量在堆栈上分配，其他的变量在堆内存上分配，但是任然有一些情况下的堆内存分配会让我们感到吃惊。下面让我们分析一些常见的不必要的堆内存分配行为并对其进行优化。

1) 字符串-String

在 c#中，字符串是引用类型变量而不是值类型变量，即使它看起来是存储字符串的，这就意味着字符串会造成一定的内存垃圾。然而在代码中字符串是我们经常使用的，所以我们需要对其格外小心。

c#中的字符串是不可变更的，也就是说其内部的值在创建后是不可被变更的。每次对字符串进行操作的时候（例如运用字符串的“+”操作），Unity 会新建一个字符串用来存储新的字符串，使得旧的字符串被废弃，这样就会造成内存垃圾。

我们可以采用以下的一些方法来最小化字符串的影响：

① 减少不必要的字符串的创建，如果一个字符串被多次利用，我们可以创建并缓存该字符串。

② 减少不必要的字符串操作，例如如果在 Text 组件中，有一部分字符串需要经常改变，但是其他部分不会，则我们可以将其分为两个部分的组件，对于不变的部分就设置为类似常量字符串即可。

③ 如果我们需要实时的创建字符串，我们可以采用 StringBuilder 来代替，StringBuilder 专为不需要进行内存分配而设计，从而减少字符串产生的内存垃圾。

④ 移除游戏中的 Debug.Log() 函数的代码，虽然该函数输出可能为空，但是该函数的调用依然会执行。Debug.Log()函数会创建至少一个字符（空字符）的字符串。如果游戏中有大量的该函数的调用，这会造成内存垃圾的增加。

在下面的代码中， Update 函数中会进行一个 string 的操作，这样的操作就会造成不必要的内存垃圾。

PS：

```
public Text timerText;
private float timer;
void Update()
{
    timer += Time.deltaTime;
    timerText.text = "Time:"\+ timer.ToString();
}
```

通过将字符串进行分隔，我们可以剔除字符串的加操作，从而减少不必要的内存垃圾。

PS：

```
public Text timerHeaderText;
public Text timerValueText;
private float timer;
void Start()
{
    timerHeaderText.text = "TIME:";
}

void Update()
```

```
{
    timerValueText.text = timer.ToString();
}
```

2) Unity 函数调用

在编程中，我们调用不是自己编写的代码时，无论是 Unity 自带的还是插件中的，我们都可能会产生内存垃圾。Unity 的某些函数调用会产生内存垃圾，我们在使用的时候需要注意它的使用。

这儿没有明确的列表指出哪些函数需要注意，每个函数在不同的情况下有不同的使用，所以最好仔细地分析游戏，定位内存垃圾的产生原因以及如何解决问题。有时候缓存是一种有效的办法，有时候尽量降低函数的调用频率是一种办法，有时候用其他函数来重构代码是一种办法。

在 Unity 中如果函数需要返回一个数组，则一个新的数组会被分配出来用作结果返回，这不容易被注意到，特别是如果该函数含有迭代器，下面的代码中对于每个迭代器都会产生一个新的数组。

PS：

```
void ExampleFunction()
{
    for(int i=0; i < myMesh.normals.Length;i++)
    {
        Vector3 normal = myMesh.normals[i];
    }
}
```

对于这样的问题，我们可以缓存一个数组的引用，这样只需要分配一个数组就可以实现相同的功能，从而减少内存垃圾的产生。

PS：

```
void ExampleFunction()
{
    Vector3[] meshNormals = myMesh.normals;
    for(int i=0; i < meshNormals.Length;i++)
    {
        Vector3 normal = meshNormals[i];
    }
}
```

另外的一个函数调用 GameObject.name 或者 GameObject.tag 也会造成预想不到的堆内存分配，这两个函数都会将结果存为新的字符串返回，这就会造成不必要的内存垃圾。对结果进行缓存是一种有效的办法，但是在 Unity 中都对应的有相关的函数来替代。对于比较 gameObject 的 tag，可以采用 GameObject.CompareTag()来替代。

在下面的代码中，调用 gameobject.tag 就会产生内存垃圾。

PS：

```
private string playerTag="Player";
void OnTriggerEnter(Collider other)
{
    bool isPlayer = other.gameObject.tag == playerTag;
}
```

采用 GameObject.CompareTag()可以避免内存垃圾的产生。

PS：

```
private string playerTag = "Player";
void OnTriggerEnter(Collider other)
{
    bool isPlayer = other.gameObject.CompareTag(playerTag);
}
```

不只是 GameObject.CompareTag ，Unity 中许多其他的函数也可以避免内存垃圾的生成。比如我们可以用 Input.GetTouch()和 Input.touchCount 来代替 Input.touches，或者用 Physics.SphereCastNonAlloc()来代替 Physics.SphereCastAll()。

3) 装箱操作

装箱操作是指一个值类型变量被用作引用类型变量时的内部变换过程，如果我们向带有对象类型参数的函数传入值类型，这就会触发装箱操作。比如 String.Format()函数需要传入字符串和对象类型参数，如果传入字符串和 int 类型数据，就会触发装箱操作。如下面代码所示。

PS：

```
void ExampleFunction()
{
    int cost = 5;
    string displayString = String.Format("Price:{0} gold",cost);
}
```

在 Unity 的装箱操作中，对于值类型会在堆内存上分配一个 System.Object 类型的引用来封装该值类型变量，其对应的缓存就会产生内存垃圾。装箱操作是非常普遍的一种产生内存垃圾的行为，即使代码中没有直接的对变量进行装箱操作，在插件或者其他的函数中也有可能会产生。最好的解决办法是尽可能的避免或者移除造成装箱操作的代码。

4) 协程

调用 StartCoroutine()会产生少量的内存垃圾，因为 Unity 会生成实体来管理协程。所以在游戏的关键时刻应该限制该函数的调用。基于此，任何在游戏关键时刻调用的协程都需要特别的注意，特别是包含延迟回调的协程。

yield 在协程中不会产生堆内存分配，但是如果 yield 带有参数返回，则会造成不必要的内存垃圾。

PS：

```
yield return 0;
```

由于上面的代码需要返回 0，触发了装箱操作，所以会产生内存垃圾。这种情况下，为了避免内存垃圾，我们可以这样返回。

PS：

```
yield return null;
```

另外一种对协程的错误使用是每次返回的时候都 new 同一个变量。

PS：

```
while(!isComplete)
{
    yield return new WaitForSeconds(1f);
}
```

我们可以采用缓存来避免这样的内存垃圾产生。

PS：

```
WaitForSeconds delay = new WaiForSeconds(1f);
while(!isComplete)
{
    yield return delay;
}
```

如果游戏中的协程产生了内存垃圾，我们可以考虑用其他的方式来替代协程。重构代码对于游戏而言十分复杂，但是对于协程而言我们也可以注意一些常见的操作。如果用协程来管理时间，最好在 update 函数中保持对时间的记录。如果用协程来控制游戏中事件的发生顺序，最好对于不同事件之间有一定的信息通信的方式。对于协程而言没有适合各种情况的方法，只有根据具体的代码来选择最好的解决办法。

5) 函数引用

函数的引用，无论是指向匿名函数还是显式函数，在 Unity 中都是引用类型变量，这都会在堆内存上进行分配。匿名函数的调用完成后都会增加内存的使用和堆内存的分配。具体函数的引用和终止都取决于操作平台和编译器设置，但是如果想减少 GC 最好减少函数的引用。

6) LINQ 和常量表达式

由于 LINQ 和常量表达式以装箱的方式实现，所以在使用的时候最好进行性能测试。

11. 重构代码来减小 GC 的影响

即使我们减小了代码在堆内存上的分配操作，代码也会增加 GC 的工作量。最常见的增加 GC 工作量的方式是让其检查它不必检查的对象。struct 是值类型的变量，但是如果 struct 中包含有引用类型的变量，那么 GC 就必须检测整个 struct。如果这样的操作很多，那么 GC 的工作量就大大增加。在下面的例子中 struct 包含一个 string，那么整个 struct 都必须在 GC 中被检查。

PS：

```
public struct ItemData
{
    public string name;
    public int cost;
    public Vector3 position;
}

private ItemData[] itemData;
```

我们可以将该 struct 拆分为多个数组的形式，从而减小 GC 的工作量。

PS：

```
private string[] itemNames;
private int[] itemCosts;
private Vector3[] itemPositions;
```

另外一种在代码中增加 GC 工作量的方式是保存不必要的 Object 引用，在进行 GC 操作的时候会对堆内存上的 object 引用进行检查，越少的引用就意味着越少的检查工作量。在下面的例子中，当前的对话框中包含一个对下一个对话框引用，这就使得 GC 的时候会去检查下一个对象框。

PS：

```
public class DialogData
```

```
{
    private DialogData nextDialog;
    public DialogData GetNextDialog()
    {
        return nextDialog;
    }
}
```

通过重构代码，我们可以返回下一个对话框实体的标记，而不是对话框实体本身，这样就没有多余的 object 引用，从而减少 GC 的工作量。

PS：

```
public class DialogData
{
    private int nextDialogID;
    public int GetNextDialogID()
    {
        return nextDialogID;
    }
}
```

如果我们的游戏中包含大量的含有对其他 Object 引用的 object，我们可以考虑通过重构代码来减少 GC 的工作量。

12. 主动调用 GC 操作

如果我们知道堆内存在被分配后并没有被使用，我们希望可以主动地调用 GC 操作，或者在 GC 操作并不影响游戏体验的时候（例如场景切换的时候），我们可以主动的调用 GC 操作：

PS：

```
System.GC.Collect();
```

通过主动的调用，我们可以主动驱使 GC 操作来回收堆内存。

十八. 地形 5 种常见接缝的修复方案

地形渲染时总是会遇到接缝问题，但这并不是单一的原因导致.有很多情况都会产生接缝，这里列举我开发过程中遇到的 5 种情况。

1. 图集问题

这是手机开发最常见的一种情况，因为把 repeat 采样的小图都拼到一个图集里，采样单个图片边缘时，原本的左右、上下 2 像素硬件双线性插值采样变成了与相邻像素插值了。最后输出的结果当然就是错误的了。

图 22.　图集边缘显示问题

一般做法是把单个图块最左边像素拷贝到右边，右边拷贝到左边，上下同样操作。采样的时候就会和 repeat 单图时插值内容与结果一致了。在端游直接用 Texture2DArray 代替图集，完美绕过这个问题。

这种拼接拷贝时如果增加了图片的尺寸，那么整个图集大小显存可能翻倍，比如原来 4 张 1024 用 4096 尺寸的图集，现在 4096 不够了直接 8192 肯定不划算。所以一般拼 3x3 或 7x7 块，而不是 4x4 充分利用空间这种。

还有如果只复制一行一列，那么 mipmap 又错误了，所以要么选择复制多行多列，要么选择放弃默认 mipmaps 创建方式，自己生成 mipmap 并每一 level 都做左右上下拼接。此外 uv 坐标也需要做调整，不再是默认规则的 1/4 等数据。总之比较麻烦，能用 Texture2DArray 尽量用 Texture2DArray 完全避开这问题。

2. Mipmap 跳变问题

一般多层地形都不采用 unity 每层 ID 和权重都记录的方式，而是只记录每个纹素最重要几个 ID，这种模式记录与采样方式有多种技术变种，但都会遇到不同图层 ID 的 tilingScale 不同，导致底层自动计算 ddx ddy 出现跳变。因为 worldPos.xz 是连续的，但由于 id 不同，所以 worldPos.xz * tilingScale 就变得不连续了，假设屏幕空间当前像素对应 worldPos.xz 为 1，屏幕空间右边像素对应 worldPos.xz 为 1.01，这个作为 uv 进行 ddx ddy 是真实变化量就是变化了 0.01。但如果当前 tilingScale 是 1，右边像素是 2，那么 ddx ddy 的内容变化量为 1.02，显然与当前点真实 uv 变化量不同了，且差了 100 倍。所以变化率也不同，导致 mipmaps 计算出错，而且错的离谱。所以会出现完全不同的颜色比如取到了完全不代表当前颜色的 miplevel10 的像素，整个图片都平均成一个颜色了。所以就出现了纯色的虚线。

图 23.　mipmaps 计算出错导致的错误结果

对应的方法就是手动计算出正确 lod，因为知道了原因，那么只要计算 ddx ddy 时，把 ddx(worldPos*currentIDTiling) 改成 ddx(worldPos)*currentIDTiling，就可以保证比例又不会让偏导内容不连续了。

十九.　线程池技术

线程池是一种线程管理技术，它可以事先创建好一定数量的线程，这些线程被放在一个池子里，等待任务的到来。一旦有任务到来，就从线程池中取出一条空闲的线程，执行该任务。

线程池的优点在于：

① 线程池可以避免频繁创建和销毁线程的开销，在线程数量较多、任务执行效率较高的情况下，能够提升系统的性能。

② 线程池可以控制线程的数量，避免过多的线程占用系统资源，从而导致系统性能下降或者崩溃。

③ 线程池可以实现任务调度和任务执行的分离，可以将任务放到队列中去，供线程池按照优先级执行。

④ 线程池中的线程可以被复用，可以接受多个任务的调度，减少了线程创建和销毁的开销。

在使用线程池时，需要注意以下几点：

① 确定线程池的大小，这个数值需要根据实际应用场景和系统硬件资源等因素来确定。

② 对于 CPU 密集型和 I/O 密集型任务，线程池的大小应该有所不同，可以根据实际情况进行调整。

③ 对于任务执行过程中的异常，需要进行合理的处理，避免影响线程池的正常调度。

④ 线程池可能会出现死锁、饥饿等问题，需要进行合理的架构设计和优化。

二十.　还原死机前场景

当我们在 Unity3D 中编辑场景，突然死机时，可以在项目文件目录中找到 Temp 文件夹，双击文件夹，找到_Backupscenes 文件夹，把后缀为 .backup 的文件后缀改为 .unity ，然后拖进 Unity3D 的 Project 界面里面，这样就可以还原死机前场景最后情况。