Unity Game Developer Postion Interview – Part 3: Summayr

deepwaterooo

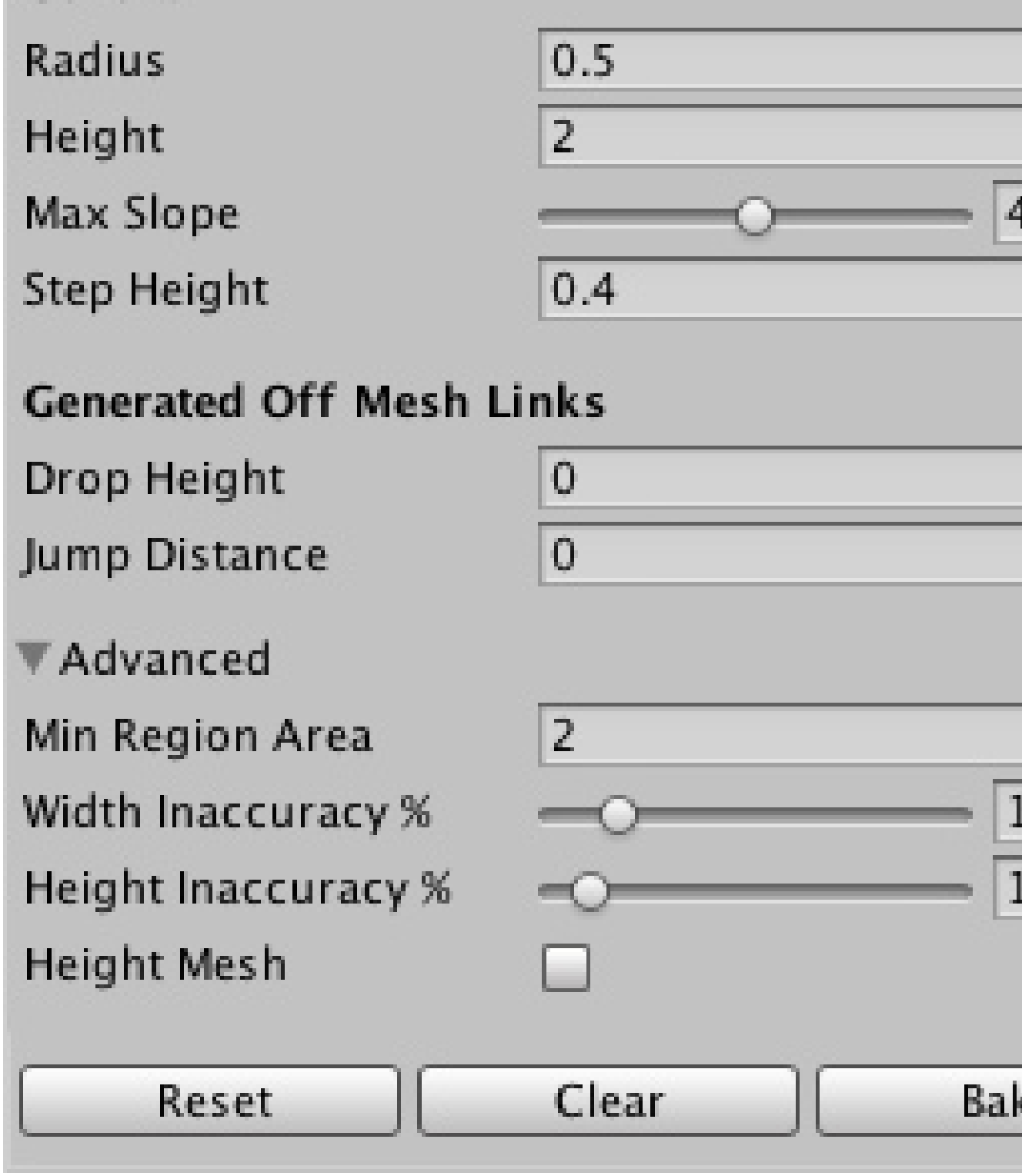
February 15, 2018

Contents

1 Nav Mesh **1**
- 1.1 Navigation . . . 1
  - 1.1.1 Object：物体参数面板 . . . 1
  - 1.1.2 Bake：烘培参数面板 . . . 2
- 1.2 Nav Mesh Agent：导航组建参数面板 . . . 3

2 基类 MonoBehaviour/自带函数以及脚本执行的生命周期 **3**
- 2.1 MonoBehaviour 的生命周期： . . . 3
- 2.2 编辑器（Editor） . . . 4
- 2.3 第一次场景加载（First Scene Load） . . . 4
- 2.4 第一帧更新之前（Before the first frame update） . . . 4
- 2.5 在帧之间（In between frames） . . . 4
- 2.6 更新顺序（Update Order） . . . 4
- 2.7 渲染（Rendering） . . . 4
- 2.8 协同程序（Coroutines） . . . 5
- 2.9 销毁（When the Object is Destroyed） . . . 5
- 2.10 退出游戏（When Quitting） . . . 5
- 2.11 官方给出的脚本中事件函数的执行顺序如下图 . . . 6

3 Unity 优化 **7**
- 3.1 Shader 着色器 . . . 8
- 3.2 光源 . . . 8
- 3.3 碰撞器 . . . 9
- 3.4 贴图纹理 . . . 9
- 3.5 脚本 . . . 10
- 3.6 组件 . . . 12
- 3.7 NGUI . . . 12
- 3.8 顶点数 . . . 12
- 3.9 材质 . . . 12
- 3.10 特效 . . . 13
- 3.11 模型物体 . . . 13
- 3.12 粒子系统 . . . 13
- 3.13 DrawCalls： . . . 14
- 3.14 Draw Call Batching . . . 14
- 3.15 Static Batching 静态批处理 . . . 14
- 3.16 Dynamic Batching 动态批处理 . . . 16
- 3.17 物理组件 . . . 19
- 3.18 处理内存，却让 CPU 受伤的 GC . . . 19
- 3.19 代码？脚本？ . . . 20
- 3.20 GPU 的优化 . . . 20
  - 3.20.1 减少绘制的数目 . . . 21
  - 3.20.2 优化显存带宽 . . . 21
- 3.21 内存的优化 . . . 21
  - 3.21.1 Unity3D 内部内存 . . . 21
  - 3.21.2 Mono 托管内存 . . . 21
  - 3.21.3 Assetbundle 的内存处理 . . . 22



- Radius：具有代表性的物体半径，半径越小生成的网格面积越大。
- Height：具有代表性的物体的高度。
- Max Slope：斜坡的坡度。
- Ste Height：台阶高度。
- Drop Height：允许最大的下落距离。
- Jump Distance：允许最大的跳跃距离。
- Min Region Area：网格面积小于该值则不生成导航网格。

- Width Inaccuracy：允许最大宽度的误差。
- Height Inaccuracy：允许最大高度的误差。
- Height Mesh：勾选后会保存高度信息，同时会消耗一些性能和存储空间。

1.2 Nav Mesh Agent：导航组建参数面板

- Radius：物体的半径
- Speed：物体的行进最大速度
- Acceleration：物体的行进加速度
- Augular Speed：行进过程中转向时的角速度。
- Stopping Distance：离目标距离还有多远时停止。
- Auto Traverse Off Mesh Link：是否采用默认方式度过链接路径。
- Auto Repath：在行进某些原因中断后是否重新开始寻路。
- Height：物体的高度。
- Base Offset：碰撞模型和实体模型之间的垂直偏移量。
- Obstacle Avoidance Type：障碍躲避的的表现登记，None 选项为不躲避障碍，另外等级越高，躲避效果越好，同时消耗的性能越多。
- Avoidance Priority：躲避优先级。
- NavMesh Walkable：该物体可以行进的网格层掩码。

2 基类 MonoBehaviour/自带函数以及脚本执行的生命周期

- Awake -> OnEnable -> Start ->-> FixedUpdate -> Update -> LateUpdate ->OnGUI -> OnDisable -> OnDestroy

2.1 MonoBehaviour 的生命周期：

- MonoBehaviour 是 Unity 中所有脚本的基类，如果你使用 JS 的话，脚本会自动继承 MonoBehaviour。如果使用 C# 的话，你需要显式继承 MonoBehaviour。
- 在我们使用 MonoBehaviour 的时候，尤其需要注意的是它有哪些可重写函数，这些可重写函数会在游戏中发生某些事件的时候被调用。我们在 Unity 中最常用到的几个可重写函数是这几个：
  - Awake：当一个脚本实例被载入时 Awake 被调用。我们大多在这个类中完成成员变量的初始化，执行一次。
  - Start：仅在 Update 函数第一次被调用前调用，只执行一次。因为它是在 Awake 之后被调用的，我们可以把一些需要依赖 Awake 的变量放在 Start 里面初始化。同时我们还大多在这个类中执行 StartCoroutine 进行一些协程的触发。要注意在用 C# 写脚本时，必须使用 StartCoroutine 开始一个协程，但是如果使用的是 JavaScript，则不需要这么做。
  - Update：当 MonoBehaviour 启用时，其 Update 在每一帧被调用。
  - FixedUpdate：当 MonoBehaviour 启用时，其 FixedUpdate 在每一固定帧被调用。生命周期中可以被执行多次。FixedUpdate 函数适合调用 Rigidbody 逻辑。
  - OnEnable：当对象变为可用或激活状态时此函数被调用。可执行多次，每次激活对象时对象上 MonoBehaviour 上脚本会调用一次。

```
gameObject.SetActive(false);// 先隐藏对象
gameObject.SetActive(true); // 显示对象
// 或
enabled = false; // 先关闭启用
enabled = true;  // 开启启用
// 都会立马会执行 OnEnable 函数
// 函数里适合放适配的逻辑
```

  - OnDisable：当对象变为不可用或非激活状态时此函数被调用。
  - OnDestroy：当 MonoBehaviour 将被销毁时，这个函数被调用。

2.2 编辑器（Editor）

- Reset：Reset 函数被调用来初始化脚本属性当脚本第一次被附到对象上，并且在 Reset 命令被使用时也会调用。
  - 编者注：Reset 是在用户点击 Inspector 面板上 Reset 按钮或者首次添加该组件时被调用。Reset 最常用于在见识面板中给定一个默认值。

2.3 第一次场景加载（First Scene Load）

- 这些函数会在一个场景开始（场景中每个物体只调用一次）时被调用。
- Awake：这个函数总是在任何 Start() 函数之前一个预设被实例化之后被调用，如果一个 GameObject 是非激活的（inactive），在启动期间 Awake 函数是不会被调用的直到它是活动的（active）。
- OnEnable：只有在对象是激活（active）状态下才会被调用，这个函数只有在 object 被启用（enable）后才会调用。这会发生在一个 MonoBehaviour 实例被创建，例如当一个关卡被加载或者一个带有脚本组件的 GameObject 被实例化。
- 注意:当一个场景被添加到场景中,所有脚本上的 Awake() 和 OnEable() 函数将会被调用在 Start()、Update() 等它们中任何函数被调用之前。自然的，当一个物体在游戏过程中被实例化时这不能被强制执行。

2.4 第一帧更新之前（Before the first frame update）

- Start: 只要脚本实例被启用了 Start() 函数将会在 Update() 函数第一帧之前被调用。
  - 对于那些被添加到场景中的物体，所有脚本上的 Start() 函数将会在它们中任何的 Update() 函数之前被调用，自然的，当一个物体在游戏过程中被实例化时这不能被强制执行。

2.5 在帧之间（In between frames）

- OnApplicationPause：这个函数将会被调用在暂停被检测有效的在正常的帧更新之间的一帧的结束时。在 OnApplicationPause 被调用后将会有额外的一帧用来允许游戏显示显示图像表示在暂停状态下。

2.6 更新顺序（Update Order）

- 当你在跟踪游戏逻辑和状态，动画，相机位置等的时候，有几个不同的事件函数你可以使用。常见的模式是在 Update() 函数中执行大多数任务，但是也有其它的函数你可以使用。
- FixedUpdate：FixedUpdate 函数经常会比 Update 函数更频繁的被调用。它一帧会被调用多次，如果帧率低它可能不会在帧之间被调用，就算帧率是高的。所有的图形计算和更新在 FixedUpdate 之后会立即执行。当在 FixedUpdate 里执行移动计算，你并不需要 Time.deltaTime 乘以你的值，这是因为 FixedUpdate 是按真实时间，独立于帧率被调用的。
- Update：Update 每一帧都会被调用，对于帧更新它是主要的负荷函数。
- LateUpdate：LateUpdate 会在 Update 结束之后每一帧被调用，任何计算在 Update 里执行结束当 LateUpdate 开始时。LateUpdate 常用为第三人称视角相机跟随。

2.7 渲染（Rendering）

- OnPreCull：在相机剔除场景前被调用。剔除是取决于哪些物体对于摄像机是可见的，OnPreCull 仅在剔除起作用之前被调用。
- OnBecameVisible/OnBecameInvisible：当一个物体对任意摄像机变得可见/不可见时被调用。
- OnPreRender：在摄像机开始渲染场景之前调用。
- OnRenderObject：在指定场景渲染完成之后调用，你可以使用 GL 类或者 Graphics.DrawMeshNow 来绘制自定义几何体在这里。

- OnPostRender：在摄像机完成场景渲染之后调用。
- OnRenderImage(Pro Only)：在场景徐然完成之后允许屏幕图像后期处理调用。
- OnGUI：为了响应 GUI 事件，每帧会被调用多次（一般最低两次）。布局 Layout 和 Repaint 事件会首先处理，接下来处理的是是通过 Layout 和键盘/鼠标事件对应的每个输入事件。
- OnDrawGizmos：用于可视化的绘制一些小玩意在场景视图中。

2.8 协同程序（Coroutines）

- 正常的协同程序更新是在 Update 函数返回之后运行。一个协同程序是可以暂停执行（yield）直到给出的依从指令（YieldInstruction ）完成，写成的不同运用：
- yield：在所有的 Update 函数都已经被调用的下一帧该协程将持续执行。
- yield WaitForSeconds：一段指定的时间延迟之后继续执行，在所有的 Update 函数完成调用的那一帧之后。
- yield WaitForFixedUpdate：所有脚本上的 FixedUpdate 函数已经执行调用之后持续。
- yield WWW：在 WWW 下载完成之后持续。
- yield StartCoroutine：协同程序链，将会等到 MuFunc 函数协程执行完成首先。

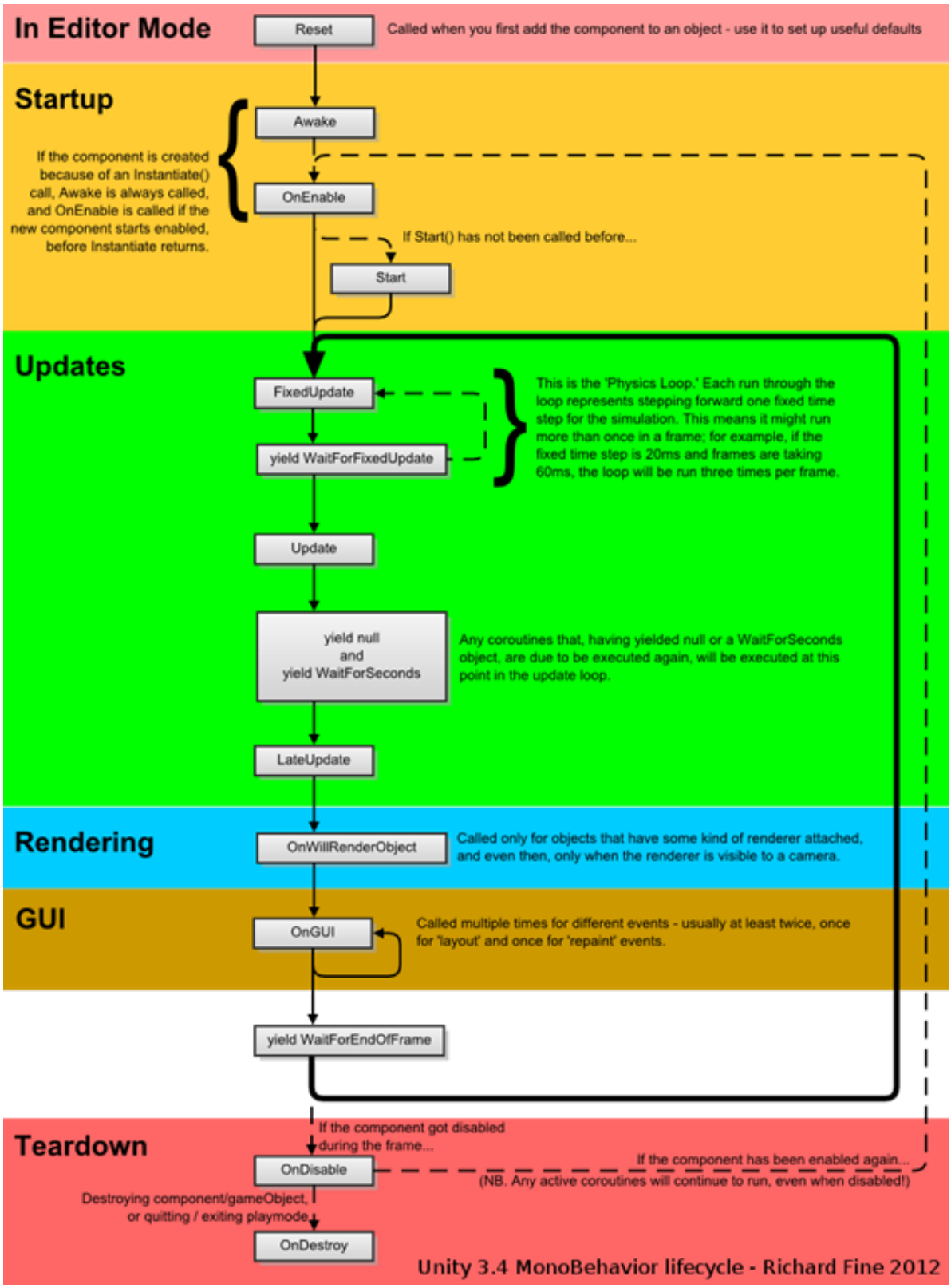
2.9 销毁（When the Object is Destroyed）

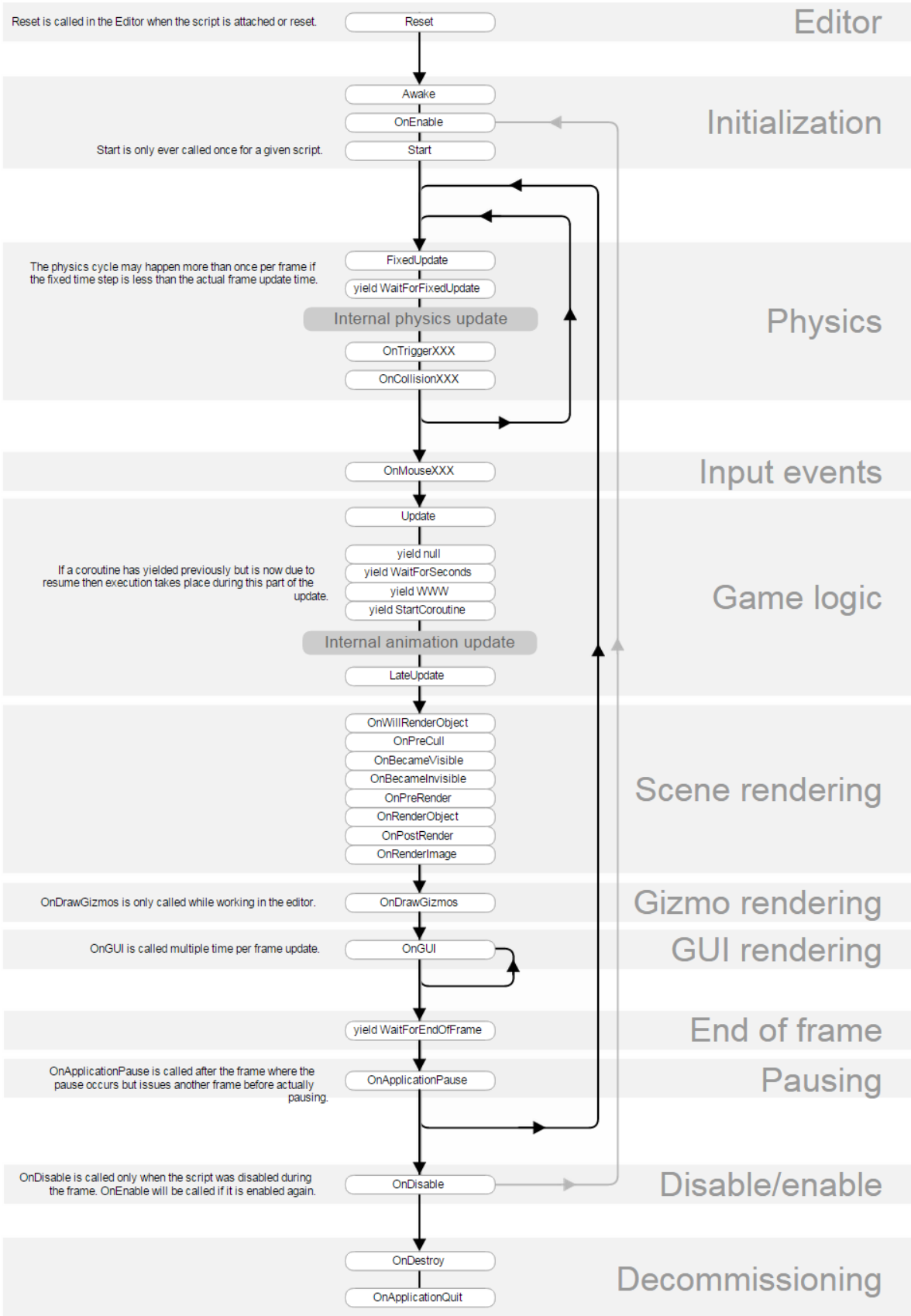
- OnDestory: 这个函数在会在一个对象销毁前一帧调用，会在所有帧更新一个对象存在的最后一帧之后执行，对象也许会响应 Object.Destroy 或一个场景关闭时被销毁。

2.10 退出游戏（When Quitting）

- 这些函数会在你场景中所有的激活的物体上调用：
- OnApplicationQuit：这个函数在应用退出之前的所有游戏物体上调用，在编辑器（Editor）模式中会在用户停止 PlayMode 时调用，在网页播放器（web player）中会在网页视图关闭时调用。
- OnDisable：当行为变为非启用（disable）或非激活（inactive）时调用。

2.11 官方给出的脚本中事件函数的执行顺序如下图





3 Unity 优化

- DrawCalls：控制电脑平台上 DrawCalls 几千个之内，移动平台上 DrawCalls 几百个之内

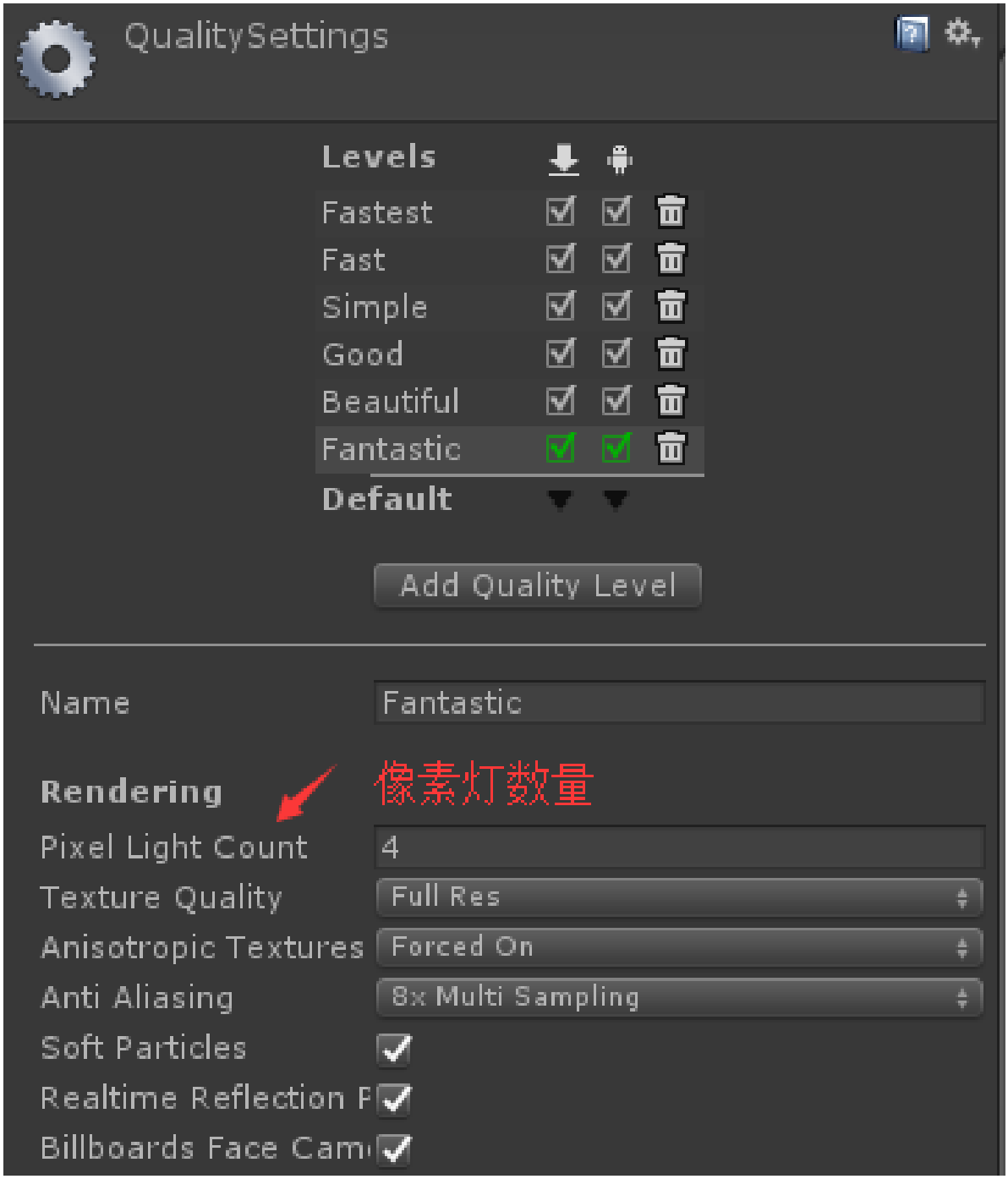
- Verts：PC 平台的话保持场景中显示的顶点数少于 300W，移动设备的话少于 10W，一切取决于你的目标 GPU 与 CPU。
- 需要注意的是:
  - 如果在 Profiler 下的 GPU 中显示的 RenderTexture.SetActive() 占用率很高的话，那么可能是因为你同时打开了编辑窗口的原因，而不是 U3D 的 BUG。

3.1 Shader 着色器

- （1）有些着色器可能是处理器密集型的，因此最好为材质指定移动设备专用的着色器。将着色器从 Diffuse 修改为 Mobile/Diffuse。
- （2）shader 中用贴图混合的方式去代替多重通道计算。
- （3）shader 中注意 float/half/fixed 的使用。
- （4）shader 中不要用复杂的计算 pow,sin,cos,tan,log 等。
- （5）shader 中越少 Fragment 越好。
- （6）自己写的 shader 请注意复杂操作符计算，类似 pow,exp,log,cos,sin,tan 等都是很耗时的计算，最多只用一次在每个像素点的计算。不推荐你自己写 normalize,dot,inversesqart 操作符，内置的肯定比你写的好。
- （7）需要警醒的是 alpha test，这个非常耗时。
- （8）浮点类型运算: 精度越低的浮点计算越快。
  - 在 CG/HLSL 中：float ： 32 位浮点格式，适合顶点变换运算，但比较慢。
    - half：16 位浮点格式，适合贴图和 UV 坐标计算，是 highp 类型计算的两倍。
    - fixed: 10 位浮点格式，适合颜色，光照，和其他。是 highp 格式计算的四倍。

3.2 光源

- （1）最好使用平行光，点光源和聚光灯消耗资源比较大
- （2）限制灯光使用数量，尽可能不用灯光。动态灯光更加不要了。
- （3）Lightmapping 烘焙灯光，为场景添加光源时要考虑一下，因为有渲染开销。如果你以前做过着色器编程，你会知道为了支持动态光源的渲染，要付出额外的代价。每个光源都需要渲染对象，根据对象使用的着色器、材质计算最终的光源效果，这个计算开销很大。尽可能的在渲染之前就将光源细节”烘焙（bake）”到对象的纹理中。”烘焙”是使用静态光源效果的渲染方式，可以实现相同的视觉效果，而无需额外的计算开销。
- （4）实时阴影技术非常棒，但消耗大量计算。为 GPU 和 CPU 都带来了昂贵的负担
- （5）灯光的 Shadow Type 只对 PC 平台有效，也就是说在移动平台是没有阴影效果的（亲测），另外软阴影更为昂贵，耗资源!!!
- （6）light 的 Render Mode 下的 Auto 是根据附近灯光的亮度和当前质量的设置（Edit ->Project Settings -> Quality ）在运行时确定，Not Important 为顶点渲染，Important 为像素渲染（更耗资源），但是像素渲染能够实现顶点渲染不能实现的效果，比如实时阴影，因此必须权衡前后照明质量和游戏速度。像素灯的实际数量可以在质量设置（Edit -> Project Settings -> Quality ）中的进行设置。



3.3 碰撞器

- （1）通常，碰撞器根据复杂度排序，对象越复杂，使用这个对象的性能开销越大。有可能的话，用盒子或者球体 (Box/Sphere) 来封装对象，这样碰撞器的计算最少。不要用网格碰撞器 (Mesh Collider)。
- （2）注意碰撞体的碰撞层，不必要的碰撞检测请舍去。

3.4 贴图纹理

- （1）可以把图像纹理或者其它资源共享使用，尽量避免透明，可以使用填充黑色
- （2）尝试用压缩贴图格式，或用 16 位代替 32 位。图片压缩将降低你的图片大小（更快地加载更小的内存跨度 (footprint)），而且大大提高渲染表现。压缩贴图比起未压缩的 32 位 RGBA 贴图占用内存带宽少得多。
- （3）之前 U3D 会议还听说过一个优化，贴图尽量都用一个大小的格式（512 * 512 ， 1024 * 1024），这样在内存之中能得到更好的排序，而不会有内存之间空隙。

- （4）MIPMAPs，跟网页上的略缩图原理一样，在 3D 游戏中我们为游戏的贴图生成多重纹理贴图，远处显示较小的物体用小的贴图，显示比较大的物体用精细的贴图。这样能更加有效的减少传输给 GPU 中的数据。但同时也会增加内存的大小，自己根据项目来权衡利弊

- （5）如果你做了一个图集是 1024X1024 的。此时你的界面上只用了图集中的一张很小的图，那么很抱歉 1024X1024 这张大图都需要载入你的内存里面，1024 就是 4M 的内存，如果你做了 10 个 1024 的图集，你的界面上刚好都只用了每个图集里面的一张小图，那么再次抱歉你的内存直接飙 40M。意思是任何一个 4096 的图片，不管是图集还是 texture，他都占用 4*4=16M?

- （6）IOS 平台使用 PVRTC 压缩纹理。Adroid 平台使用 ETC1 格式压缩。均可以减至 1/4 的内存大小，优化非常明显！目前主流的 Android 机型基本都支持 ETC1 格式压缩。但 ETC1 只能支持非 Alpha 通道的图片压缩。所以一般把 Alpha 通道图分离出来，绘制到 GPU 显存时，a 值从 Alpha 图里获取，无 Alpha 通道的图就可以使用 ETC1 压缩。

- （7）设置不透明贴图的压缩格式为 ETC 4bit，因为 android 市场的手机中的 GPU 有多种，每家的 GPU 支持不同的压缩格式，但他们都兼容 ETC 格式
  - 对于透明贴图，我们只能选择 RGBA 16bit 或者 RGBA 32bit。

- （8）减少 FPS，在 ProjectSetting -> Quality 中的 VSync Count 参数会影响你的 FPS，EveryVBlank 相当于 FPS=60，EverySecondVBlank = 30；
  - 如果这两种情况都不符合游戏的 FPS 的话，我们需要手动调整 FPS，首先关闭垂直同步（VSync = Vertical Sync）这个功能，即设置 VSync 为 Don't Sync
  - 然后在代码的 Awake 方法里手动设置 FPS

```
Application.targetFrameRate = 45;
```

  - 降低 FPS 的好处：
    * 1）省电，减少手机发热的情况；
    * 2）能都稳定游戏 FPS，减少出现卡顿的情况。

- （9）当我们设置了 FPS 后，再调整下 Fixed timestep 这个参数，这个参数在 ProjectSetting->Time 中，目的是减少物理计算的次数，来提高游戏性能。

- （10）尽量少使用 Update LateUpdate FixedUpdate，这样也可以提升性能和节省电量。多使用事件（不是 SendMessage，使用自己写的，或者 C# 中的事件委托）。

- （11）待机时，调整游戏的 FPS 为 1，节省电量。

- （12）图集大小最好不要高于 1024，否则游戏安装之后、低端机直接崩溃、原因是手机系统版本低于 2.2、超过 1000 的图集无法读取导致。2.2 以上没有遇见这个情况。注意手机的 RAM 与 ROM、小于 512M 的手机、直接放弃机型适配。

- （13）不同设备要使用不同的纹理大小，尤其是 UI 和大型背景中的纹理。《Shadow Blade》使用的是通用型模板，但如果在启动时检测到设备大小和分辨率，就会载入不同资产。

- （14）远处的物体绘制在 skybox 上

3.5 脚本

- （1）如果你不需要运行某一个脚本，那么就禁用它。不管它多少的小，或者出现的很少，但每一个处理都需要占用时间。

- （2）不要留着未实现的 Update，FixedUpdate 等方法，用不到就删除，不然会执行，消耗时间！

- （3）移除代码中的任何字符串连接，因为这会给 GC 留下大量垃圾。使用 StringBuilder 链接字符串

- （4）用简单的"for" 循环代替"foreach" 循环。由于某些原因，每个"foreach" 循环的每次迭代会生成 24 字节的垃圾内存。一个简单的循环迭代 10 次就可以留下 240 字节的垃圾内存。

- （5）更改我们检查游戏对象标签的方法。用"if (go.CompareTag ("Enemy")" 来代替"if (go.tag == "Enemy")" 。

```
if (go.CompareTag ("Enemy")) {}
// 来代替
// if (go.tag == "Enemy")
```

  - 在一个内部循环调用对象分配的标签属性以及拷贝额外内存，这是一个非常糟糕的做法。

- （6）不使用 LINQ 命令，因为它们一般会分配中间缓器，而这很容易生成垃圾内存。
- （7）修改代码以免依赖”ControllerColliderHit” 回调函数。这证明这些回调函数处理得并不十分迅速。
- （8）要谨慎评估触发器的”onInside” 回调函数，在我们的项目中，我们尽量在不依赖它们的情况下模拟逻辑。
- （9）注意是否有多余的动画脚本，模型自动导入到 U3D 会有动画脚本，大量的话会严重影响消耗 CPU 计算。
- （10）尽量避免每帧处理, 可以每隔几帧处理一次

```
void Update() {
    if (Time.frameCount % 5 == 0) {
        DoSomeThing();
    }
}
```

- （11）尽量避免使用 float，而使用 int，特别是在手机游戏中，尽量少用复杂的数学函数，比如 sin,cos 等函数。改除法/为乘法, 例如：使用 x*0.5f 而不是 x/2.0f 。
- （12）避免使用

```
for (int i = 0; i < myArray.Length; i++)
```

  - 而应该这样

```
int length = myArray.Length;
for (int i = 0; i < length; i++)
```

- （13）少使用临时变量，特别是在 Update OnGUI 等实时调用的函数中定义临时变量。
- （14）协同是一个好方法。可以使用协同程序来代替不必每帧都执行的方法。（还有 InvokeRepeating 方法也是一个好的取代 Update 的方法）。
- （15）不要使用 SendMessage 之类的方法，他比直接调用方法慢了 100 倍，你可以直接调用或通过 C# 的委托来实现。
- （16）操作 transform.localPosition 的时候请小心，移动 GameObject 是非常平常的一件事情，以下代码看起来很简单:

```
    transform.localPosition += new Vector3 ( 10.0f * Time.deltaTime, 0.0f, 0.0f );
```

  - 但是小心了，假设上面这个 GameObject 有一个 parent, 并且这个 parent GameObject 的 localScale 是 (2.0f, 2.0f, 2.0f)。你的 GameObject 将会移动 20.0 个单位/秒。
  - 因为该 GameObject 的 world position 等于:

```
Vector3 offset = new Vector3( my.localPosition.x * parent.lossyScale.x,
                              my.localPosition.y * parent.lossyScale.y,
                              my.localPosition.z * parent.lossyScale.z );
Vector3 worldPosition = parent.position + parent.rotation * offset;
```

  - 换句话说，上面这种直接操作 localPosition 的方式是在没有考虑 scale 计算的时候进行的，为了解决这个问题，unity3d 提供了 Translate 函数, 所以正确的做法应该是:

```
transform.Translate ( 10.0f * Time.deltaTime, 0.0f, 0.0f );
```

- （17）减少固定增量时间，将固定增量时间值设定在 0.04-0.067 区间（即，每秒 15-25 帧）。您可以通过 Edit -> Project Settings -> Time 来改变这个值。这样做降低了 FixedUpdate 函数被调用的频率以及物理引擎执行碰撞检测与刚体更新的频率。如果您使用了较低的固定增量时间，并且在主角身上使用了刚体部件，那么您可以启用插值办法来平滑刚体组件。

- （18）减少 GetComponent 的调用使用，GetComponent 或内置组件访问器（transform）会产生明显的开销。您可以通过一次获取组件的引用来避免开销，并将该引用分配给一个变量（有时称为” 缓存” 的引用）。

```
Transform myTransform ;
void Awake () {
    myTransform = transform;
}
```

- （19）同时，在某些可能的情况下，您也可以使用结构（struct）来代替类（class）。这是因为，结构变量主要存放在栈区而非堆区。因为栈的分配较快，并且不调用垃圾回收操作，所以当结构变量比较小时可以提升程序的运行性能。但是当结构体较大时，虽然它仍可避免分配/回收的开销，而它由于” 传值” 操作也会导致单独的开销，实际上它可能比等效对象类的效率还要低。

- （20）使用 GUILayout 函数可以很方便地将 GUI 元素进行自动布局。然而，这种自动化自然也附带着一定的处理开销。您可以通过手动的 GUI 功能布局来避免这种开销。此外，您也可以设置一个脚本的 useGUILayout 变量为 false 来完全禁用 GUI 布局：

```
void Awake () {
   useGUILayout = false;
}
```

- （21）最小化碰撞检测请求（例如 ray casts 和 sphere checks），尽量从每次检查中获得更多信息。
- （22）在 edit->project setting->time 中调大 FixedTimestep（真实物理的帧率）来减少 cpu 损耗
- （23）尽量不要动态的 instantiate 和 destroy object，使用 object pool
- （24）尽量不要再 update 函数中做复杂计算，如有需要，可以隔 N 帧计算一次
- （25）不要使用内置的 onGUii 函数处理 gui，使用其他方案，如 NGUI

3.6 组件

- （1）尽可能的使用简单组件—如果你不需求功能较多的组件，那么就自己去实现它避免一起使用大量系统组件。比如，CharacterController 是一个很废资源的组件，那么最好使用刚体来定义自己的解决方案。
- （2）面对性能更弱的设备，要用 skinned mesh 代替 physics cloth。cloth 参数在运行表现中发挥重要作用，如果你肯花些时间找到美学与运行表现之间的平衡点，就可以获得理想的结果。
- （3）在物理模拟过程中不要使用 ragdolls（布娃娃系统），只有在必要时才让它生效。
- （4）真实的物理（刚体）很消耗，不要轻易使用，尽量使用自己的代码模仿假的物理

3.7 NGUI

- （1）NGUI 中所有 Panel 都有一个 Depth 值影响着他下面的所有挂件。如果你正在创建一个使用多个窗口的复杂 UI，通常最好的做法是每个窗口有一个 UIPanel。请确认你的 panel 不会拥有相同的 depth 值。如果这个值是一样的，为了保证绘制顺序，draw call 将会开始频繁分割，这将导致产生比平常更多的 draw call。

3.8 顶点数

- （1）尽量减少顶点数

3.9 材质

- （1）尽可能共用材质。这样便可以减少 DrawCall，引擎可以进行其批处理!
- （2）如果你需要通过脚本来控制单个材质属性，需要注意改变 Renderer.material 将会造成一份材质的拷贝。因此，你应该使用 Renderer.sharedMaterial 来保证材质的共享状态。
- （3）有一个合并模型材质不错的插件叫 Mesh Baker

3.10 特效

- （1）如果不需要别用雾效 (fog)
- （2）要找到美学/性能之间的平衡，就免不了许多粒子效果的迭代。减少发射器数量并尽量减少透明度需求也是一大挑战。

3.11 模型物体

- （1）不要有不必要的三角面。面片数最好控制在 300~2000 面片
- （2）UV 贴图中的接缝和硬边越少越好。
  - 需要注意的是，图形硬件需要处理顶点数和硬件报告说的并不一样。不是硬件说能渲染几个点就是几个点。模型处理应用通常展示的是几何顶点数量。例如，一个由一些不同顶点构成的模型。在显卡中，一些集合顶点将会被分离 (split) 成两个或者更多逻辑顶点用作渲染。如果有法线、UV 坐标、顶点色的话，这个顶点必须会被分离。所以在游戏中处理的实际数量显然要多很多。
- （3）LOD (Level Of Detail) 是很常用的 3D 游戏技术了，其功能理解起来则是相当于多重纹理贴图。在以在屏幕中显示模型大小的比例来判断使用高或低层次的模型来减少对 GPU 的传输数据，和减少 GPU 所需要的顶点计算。
- （4）摄像机分层距离剔除 (Per-Layer Cull Distances)：为小物体标识层次，然后根据其距离主摄像机的距离判断是否需要显示。
- （5）遮挡剔除（Occlusion Culling）其实就是当某个物体在摄像机前被另外一个物体完全挡住的情况，挡住就不发送给 GPU 渲染，从而直接降低 DRAW CALL。不过有些时候在 CPU 中计算其是否被挡住则会很耗计算，反而得不偿失。
- （6）将不需要移动的物体设为 Static，让引擎可以进行其批处理。
- （7）用单个蒙皮渲染、尽量少用材质、少用骨骼节点、移动设备上角色多边形保持在 300~1500 内 (当然还要看具体的需求)、PC 平台上 1500~4000 内 (当然还要看具体的需求)。
  - 角色的面数一般不要超过 1500，骨骼数量少于 30 就好，越多的骨骼就会越多的带来 CPU 消耗, 角色 Material 数量一般 1~2 个为最佳。
- （8）导入 3D 模型之后，在不影响显示效果的前提下，最好打开 Mesh Compression。Off, Low, Medium, High 这几个选项，可酌情选取。
- （9）避免大量使用 unity 自带的 Sphere 等内建 Mesh，Unity 内建的 Mesh，多边形的数量比较大，如果物体不要求特别圆滑，可导入其他的简单 3D 模型代替。
- （10）每个角色尽量使用一个 Skinned Mesh Renderer, 这是因为当角色仅有一个 Skinned Mesh Renderer 时，Unity 会使用视锥型可见性裁剪和多边形网格包围体更新的方法来优化角色的运动，而这种优化只有在角色仅含有一个 Skinned Mesh Renderer 时才会启动。
- （11）对于静态物体顶点数要求少于 500，UV 的取值范围不要超过（0,1）区间，这对于纹理的拼合优化很有帮助。
- （12）不需要的 Animation 组件就删掉

3.12 粒子系统

- 粒子系统运行在 iPhone 上时很慢，怎么办？因为 iPhone 拥有相对较低的 fillrate 。如果您的粒子效果覆盖大部分的屏幕，而且是 multiple layers 的，这样即使最简单的 shader，也能让 iPhone 傻眼。我们建议把您的粒子效果 baking 成纹理序列图。然后在运行时可以使用 1-2 个粒子，通过动画纹理来显示它们。这种方式可以取得很好的效果，以最小的代价。
- 自带地形：地形高度图尺寸小于 257，尽量使用少的混合纹理数目，尽量不要超过 4 个，Unity 自带的地形时十分占资源的，强烈建议不要使用，自己制作地形，尽量一张贴图搞定
  - drawcall 是啥？draw：绘制，call：调用，其实就是对底层图形程序（比如：OpenGL ES) 接口的调用，以在屏幕上画出东西。那么，是谁去调用这些接口呢？CPU。

- fragment 是啥？经常有人说 vf 啥的，vertex 我们都知道是顶点，那 fragment 是啥呢？说它之前需要先说一下像素，像素各位应该都知道吧？像素是构成数码影像的基本单元呀。那 fragment 呢？是有可能成为像素的东西。啥叫有可能？就是最终会不会被画出来不一定，是潜在的像素。这会涉及到谁呢？GPU。
- batching 是啥？都知道批处理是干嘛的吧？没错，将批处理之前需要很多次调用（drawcall）的物体合并，之后只需要调用一次底层图形程序的接口就行。听上去这简直就是优化的终极方案啊！但是，理想是美好的，世界是残酷的，一些不足之后我们再细聊。
- 内存的分配：记住，除了 Unity3D 自己的内存损耗。我们可是还带着 Mono 呢啊，还有托管的那一套东西呢。更别说你一激动，又引入了自己的几个 dll。这些都是内存开销上需要考虑到的。
- CPU 方面：
  * 上文中说了，drawcall 影响的是 CPU 的效率，而且也是最知名的一个优化点。但是除了 drawcall 之外，还有哪些因素也会影响到 CPU 的效率呢？让我们一一列出暂时能想得到的：
    · （1）DrawCalls
    · （2）物理组件（Physics）
    · （3）GC（什么？GC 不是处理内存问题的嘛？匹夫你不要骗我啊！不过，匹夫也要提醒一句，GC 是用来处理内存的，但是是谁使用 GC 去处理内存的呢？）
    · （4）当然，还有代码质量

3.13 DrawCalls：

- 前面说过了，DrawCall 是 CPU 调用底层图形接口。比如有上千个物体，每一个的渲染都需要去调用一次底层接口，而每一次的调用 CPU 都需要做很多工作，那么 CPU 必然不堪重负。但是对于 GPU 来说，图形处理的工作量是一样的。所以对 DrawCall 的优化，主要就是为了尽量解放 CPU 在调用图形接口上的开销。所以针对 drawcall 我们主要的思路就是每个物体尽量减少渲染次数，多个物体最好一起渲染。所以，按照这个思路就有了以下几个方案：
  - 使用 Draw Call Batching，也就是描绘调用批处理。Unity 在运行时可以将一些物体进行合并，从而用一个描绘调用来渲染他们。具体下面会介绍。
  - 通过把纹理打包成图集来尽量减少材质的使用。
  - 尽量少的使用反光啦，阴影啦之类的，因为那会使物体多次渲染。

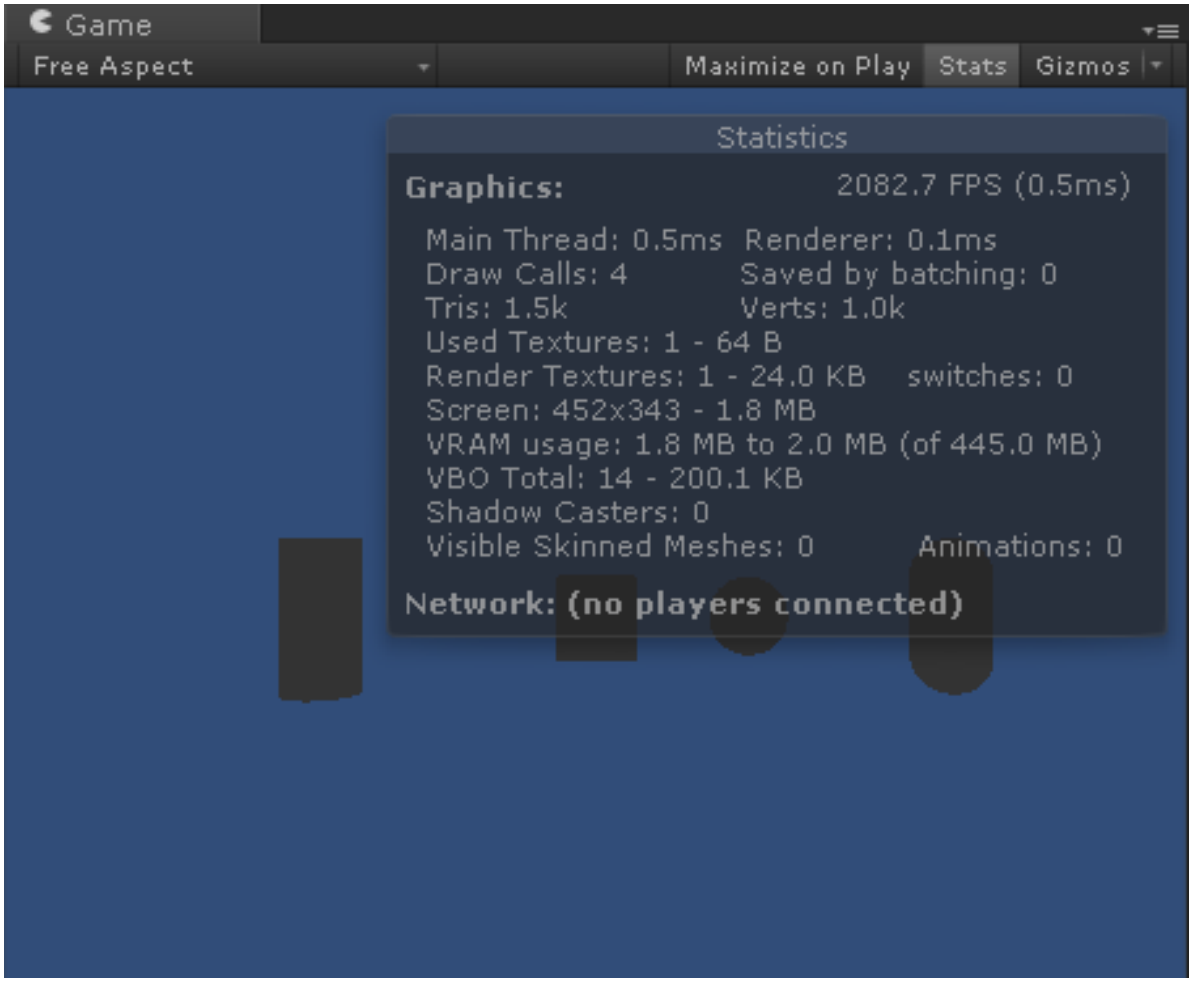
3.14 Draw Call Batching

- 首先我们要先理解为何 2 个没有使用相同材质的物体即使使用批处理，也无法实现 Draw Call 数量的下降和性能上的提升。
- 因为被” 批处理” 的 2 个物体的网格模型需要使用相同材质的目的，在于其纹理是相同的，这样才可以实现同时渲染的目的。因而保证材质相同，是为了保证被渲染的纹理相同。
- 因此，为了将 2 个纹理不同的材质合二为一，我们就需要进行上面列出的第二步，将纹理打包成图集。具体到合二为一这种情况，就是将 2 个纹理合成一个纹理。这样我们就可以只用一个材质来代替之前的 2 个材质了。
- 而 Draw Call Batching 本身，也还会细分为 2 种。

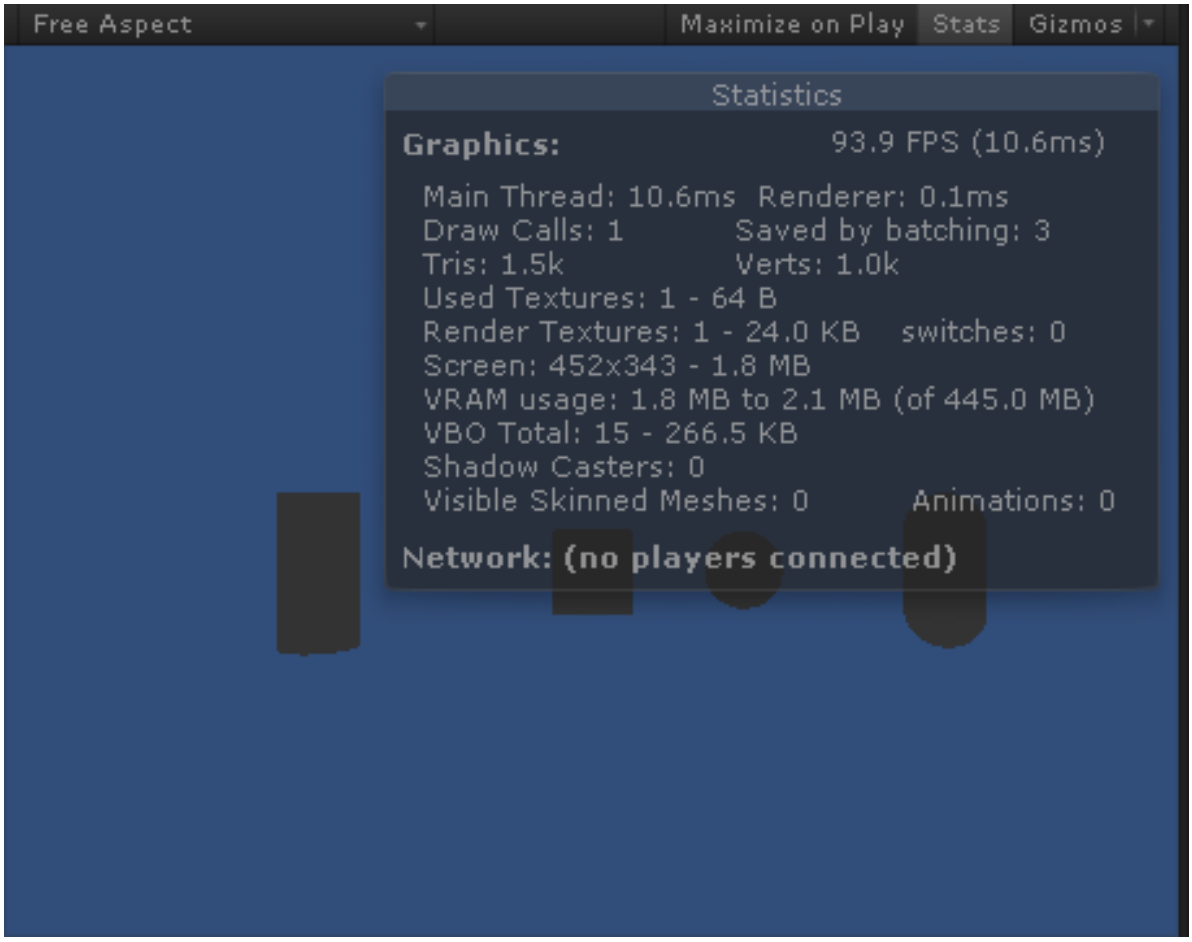
3.15 Static Batching 静态批处理

- 看名字，猜使用的情景。
- 静态？那就是不动的咯。还有呢？额，听上去状态也不会改变，没有” 生命”，比如山山石石，楼房校舍啥的。那和什么比较类似呢？嗯，聪明的各位一定觉得和场景的属性很像吧！所以我们的场景似乎就可以采用这种方式来减少 draw call 了。
- 那么写个定义：只要这些物体不移动，并且拥有相同的材质，静态批处理就允许引擎对任意大小的几何物体进行批处理操作来降低描绘调用。
- 那要如何使用静态批来减少 Draw Call 呢？你只需要明确指出哪些物体是静止的，并且在游戏中永远不会移动、旋转和缩放。想完成这一步，你只需要在检测器（Inspector）中将 Static 复选框打勾即可！
- 至于效果如何呢？

- 举个例子：新建 4 个物体，分别是 Cube，Sphere, Capsule, Cylinder, 它们有不同的网格模型，但是也有相同的材质（Default-Diffuse）。
- 首先，我们不指定它们是 static 的。Draw Call 的次数是 4 次，如图：



- 我们现在将它们 4 个物体都设为 static，在来运行一下：



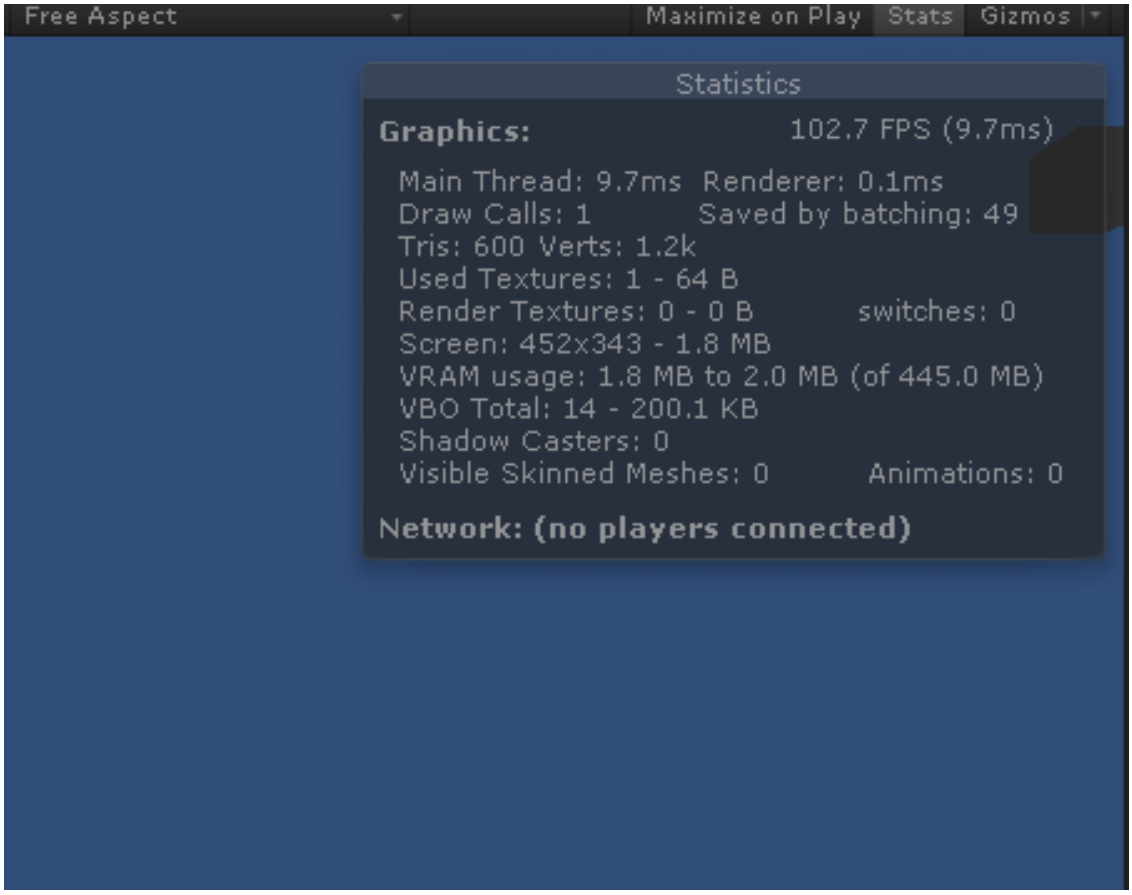
- 如图，Draw Call 的次数变成了 1，而 Saved by batching 的次数变成了 3。
- 静态批处理的好处很多，其中之一就是与下面要说的动态批处理相比，约束要少很多。所以一般推荐的是 draw call 的静态批处理来减少 draw call 的次数。那么接下来，我们就继续聊聊 draw call 的动态批处理。

3.16 Dynamic Batching 动态批处理

- 有阴就有阳，有静就有动，所以聊完了静态批处理，肯定跟着就要说说动态批处理了。首先要明确一点，Unity3D 的 draw call 动态批处理机制是引擎自动进行的，无需像静态批处理那样手动设置 static。我们举一个动态实例化 prefab 的例子，如果动态物体共享相同的材质，则引擎会自动对 draw call 优化，也就是使用批处理。首先，我们将一个 cube 做成 prefab，然后再实例化 50 次，看看 draw call 的数量。

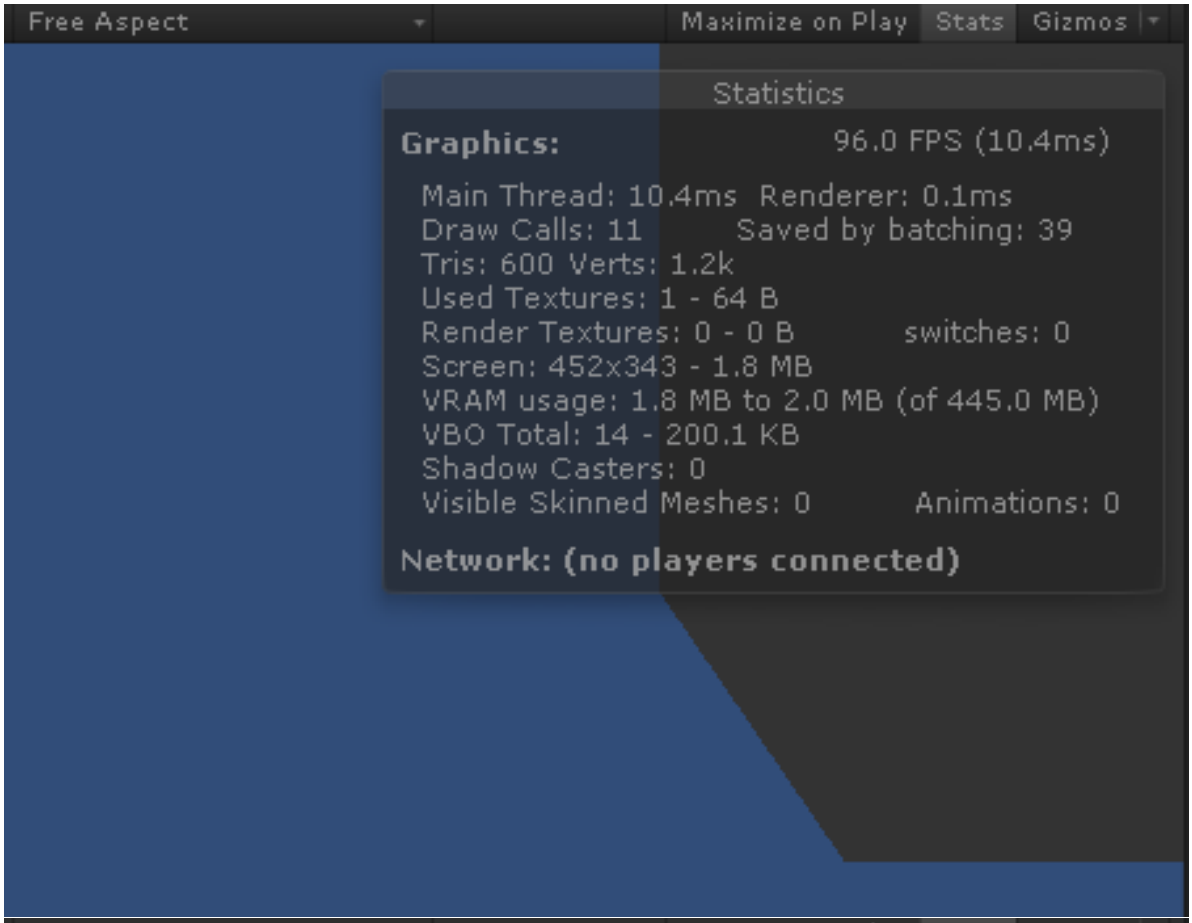
```
for (int i = 0; i < 50; i++) {
    GameObject cube;
    cube = GameObject.Instantiate(prefab) as GameObject;
}
```

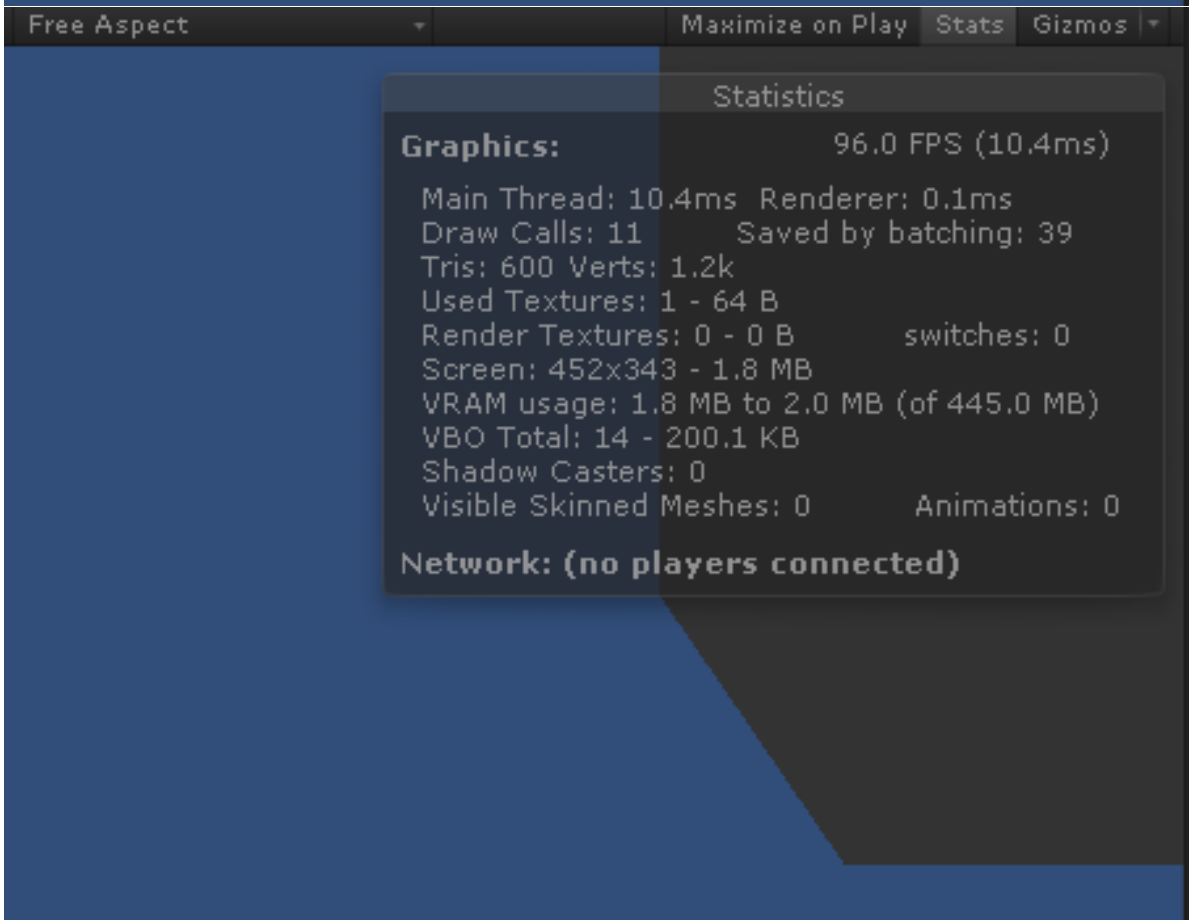
- draw call 的数量：



- 可以看到 draw call 的数量为 1，而 saved by batching 的数量是 49。而这个过程中，我们除了实例化创建物体之外什么都没做。不错，unity3d 引擎为我们自动处理了这种情况。
- 但是有很多童靴也遇到这种情况，就是我也是从 prefab 实例化创建的物体，为何我的 draw call 依然很高呢？这就是匹夫上文说的，draw call 的动态批处理存在着很多约束。下面匹夫就演示一下，针对 cube 这样一个简单的物体的创建，如果稍有不慎就会造成 draw call 飞涨的情况吧。
- 我们同样是创建 50 个物体，不同的是其中的 10 个物体，每个物体的大小都不同，也就是 Scale 不同。

```
for (int i = 0; i < 50; i++) {
    GameObject cube;
    cube = GameObject.Instantiate(prefab) as GameObject;
    if (i / 10 == 0) {
        cube.transform.localScale = new Vector3(2 + i, 2 + i, 2 + i);
    }
}
```





- 我们看到 draw call 的数量上升到了 11 次，而 saved by batching 的数量也下降到了 39。各位看官可以看到，

仅仅是一个简单的 cube 的创建，如果 scale 不同，竟然也不会去做批处理优化。这仅仅是动态批处理机制的一种约束，那我们总结一下动态批处理的约束，各位也许也能从中找到为何动态批处理在自己的项目中不起作用的原因：

  - 批处理动态物体需要在每个顶点上进行一定的开销，所以动态批处理仅支持小于 900 顶点的网格物体。
  - 如果你的着色器使用顶点位置，法线和 UV 值三种属性，那么你只能批处理 300 顶点以下的物体；如果你的着色器需要使用顶点位置，法线，UV0，UV1 和切向量，那你只能批处理 180 顶点以下的物体。
  - 不要使用缩放。分别拥有缩放大小 (1,1,1) 和 (2,2,2) 的两个物体将不会进行批处理。
  - 统一缩放的物体不会与非统一缩放的物体进行批处理。
  - 使用缩放尺度 (1,1,1) 和 (1,2,1) 的两个物体将不会进行批处理，但是使用缩放尺度 (1,2,1) 和 (1,3,1) 的两个物体将可以进行批处理。
  - 使用不同材质的实例化物体（instance）将会导致批处理失败。
  - 拥有 lightmap 的物体含有额外（隐藏）的材质属性，比如：lightmap 的偏移和缩放系数等。所以，拥有 lightmap 的物体将不会进行批处理（除非他们指向 lightmap 的同一部分）。
  - 多通道的 shader 会妨碍批处理操作。比如，几乎 unity 中所有的着色器在前向渲染中都支持多个光源，并为它们有效地开辟多个通道。
  - 预设体的实例会自动地使用相同的网格模型和材质。

- 所以，尽量使用静态的批处理。

3.17 物理组件

- 1. 设置一个合适的 Fixed Timestep。设置的位置：Edit → Project Settings → Time

  - 那何谓” 合适” 呢？首先我们要搞明白 Fixed Timestep 和物理组件的关系。物理组件，或者说游戏中模拟各种物理效果的组件，最重要的是什么呢？计算啊。对，需要通过计算才能将真实的物理效果展现在虚拟的游戏中。那么 Fixed Timestep 这货就是和物理计算有关的啦。所以，若计算的频率太高，自然会影响到 CPU 的开销。同时，若计算频率达不到游戏设计时的要求，有会影响到功能的实现，所以如何抉择需要各位具体分析，选择一个合适的值。

- 2. 就是不要使用网格碰撞器（mesh collider）：为啥？因为实在是太复杂了。网格碰撞器利用一个网格资源并在其上构建碰撞器。对于复杂网状模型上的碰撞检测，它要比应用原型碰撞器精确的多。标记为凸起的（Convex ）的网格碰撞器才能够和其他网格碰撞器发生碰撞。各位上网搜一下 mesh collider 的图片，自然就会明白了。我们的手机游戏自然无需这种性价比不高的东西。

  - 当然，从性能优化的角度考虑，物理组件能少用还是少用为好。

3.18 处理内存，却让 CPU 受伤的 GC

- 在 CPU 的部分聊 GC，感觉是不是怪怪的？其实小匹夫不这么觉得，虽然 GC 是用来处理内存的，但的确增加的是 CPU 的开销。因此它的确能达到释放内存的效果，但代价更加沉重，会加重 CPU 的负担，因此对于 GC 的优化目标就是尽量少的触发 GC。

- 首先我们要明确所谓的 GC 是 Mono 运行时的机制，而非 Unity3D 游戏引擎的机制，所以 GC 也主要是针对 Mono 的对象来说的，而它管理的也是 Mono 的托管堆。搞清楚这一点，你也就明白了 GC 不是用来处理引擎的 assets（纹理啦，音效啦等等）的内存释放的，因为 U3D 引擎也有自己的内存堆而不是和 Mono 一起使用所谓的托管堆。

- 其次我们要搞清楚什么东西会被分配到托管堆上？不错咯，就是引用类型咯。比如类的实例，字符串，数组等等。而作为 int，float，包括结构体 struct 其实都是值类型，它们会被分配在堆栈上而非堆上。所以我们关注的对象无外乎就是类实例，字符串，数组这些了。

- 那么 GC 什么时候会触发呢？两种情况：

  - 首先当然是我们的堆的内存不足时，会自动调用 GC。
  - 其次呢，作为编程人员，我们自己也可以手动的调用 GC。

- 所以为了达到优化 CPU 的目的，我们就不能频繁的触发 GC。而上文也说了 GC 处理的是托管堆，而不是 Unity3D 引擎的那些资源，所以 GC 的优化说白了也就是代码的优化。那么匹夫觉得有以下几点是需要注意的：

  - 字符串连接的处理。因为将两个字符串连接的过程，其实是生成一个新的字符串的过程。而之前的旧的字符串自然而然就成为了垃圾。而作为引用类型的字符串，其空间是在堆上分配的，被弃置的旧的字符串的空间会被 GC 当做垃圾回收。
  - 尽量不要使用 foreach，而是使用 for。foreach 其实会涉及到迭代器的使用，而据传说每一次循环所产生的迭代器会带来 24 Bytes 的垃圾。那么循环 10 次就是 240Bytes。
  - 不要直接访问 gameobject 的 tag 属性。比如 if (go.tag == "human") 最好换成 if (go.CompareTag ("human"))。因为访问物体的 tag 属性会在堆上额外的分配空间。如果在循环中这么处理，留下的垃圾就可想而知了。
  - 使用" 池"，以实现空间的重复利用。
  - 最好不用 LINQ 的命令，因为它们会分配临时的空间，同样也是 GC 收集的目标。而且我很讨厌 LINQ 的一点就是它有可能在某些情况下无法很好的进行 AOT 编译。比如"OrderBy" 会生成内部的泛型类"OrderedEnumerable"。这在 AOT 编译时是无法进行的，因为它只是在 OrderBy 的方法中才使用。所以如果你使用了 OrderBy，那么在 IOS 平台上也许会报错。

3.19 代码? 脚本?

- 聊到代码这个话题，也许有人会觉得匹夫多此一举。因为代码质量因人而异，很难像上面提到的几点，有一个明确的评判标准。也是，公写公有理，婆写婆有理。但是匹夫这里要提到的所谓代码质量是基于一个前提的：Unity3D 是用 C++ 写的，而我们的代码是用 C# 作为脚本来写的，那么问题就来了 ~ 脚本和底层的交互开销是否需要考虑呢? 也就是说，我们用 Unity3D 写游戏的" 游戏脚本语言"，也就是 C# 是由 mono 运行时托管的。而功能是底层引擎的 C++ 实现的，" 游戏脚本" 中的功能实现都离不开对底层代码的调用。那么这部分的开销，我们应该如何优化呢?

- 1. 以物体的 Transform 组件为例，我们应该只访问一次，之后就将它的引用保留，而非每次使用都去访问。这里有人做过一个小实验，就是对比通过方法 GetComponent<Transform>() 获取 Transform 组件, 通过 MonoBehavor 的 transform 属性去取，以及保留引用之后再去访问所需要的时间：

```
GetComponent = 619ms
Monobehaviour = 60ms
CachedMB = 8ms
Manual Cache = 3ms
```

- 2. 如上所述，最好不要频繁使用 GetComponent，尤其是在循环中。

- 3. 善于使用 OnBecameVisible() 和 OnBecameVisible(), 来控制物体的 update() 函数的执行以减少开销。

- 4. 使用内建的数组，比如用 Vector3.zero 而不是 new Vector(0, 0, 0);

- 5. 对于方法的参数的优化：善于使用 ref 关键字。值类型的参数，是通过将实参的值复制到形参，来实现按值传递到方法，也就是我们通常说的按值传递。复制嘛，总会让人感觉很笨重。比如 Matrix4x4 这样比较复杂的值类型，如果直接复制一份新的，反而不如将值类型的引用传递给方法作为参数。

好啦，CPU 的部分匹夫觉得到此就介绍的差不多了。下面就简单聊聊其实匹夫并不是十分熟悉的部分，GPU 的优化。

3.20 GPU 的优化

- GPU 与 CPU 不同，所以侧重点自然也不一样。GPU 的瓶颈主要存在在如下的方面：
  - 填充率，可以简单的理解为图形处理单元每秒渲染的像素数量。
  - 像素的复杂度，比如动态阴影，光照，复杂的 shader 等等
  - 几何体的复杂度（顶点数量）
  - 当然还有 GPU 的显存带宽

- 那么针对以上 4 点，其实仔细分析我们就可以发现，影响的 GPU 性能的无非就是 2 大方面，一方面是顶点数量过多，像素计算过于复杂。另一方面就是 GPU 的显存带宽。那么针锋相对的两方面举措也就十分明显了。
  - 减少顶点数量，简化计算复杂度。
  - 压缩图片，以适应显存带宽。

3.20.1 减少绘制的数目

- 那么第一个方面的优化也就是减少顶点数量，简化复杂度，具体的举措就总结如下了：
  - 保持材质的数目尽可能少。这使得 Unity 更容易进行批处理。
  - 使用纹理图集（一张大贴图里包含了很多子贴图）来代替一系列单独的小贴图。它们可以更快地被加载，具有很少的状态转换，而且批处理更友好。
  - 如果使用了纹理图集和共享材质，使用 Renderer.sharedMaterial 来代替 Renderer.material 。
  - 使用光照纹理 (lightmap) 而非实时灯光。
  - 使用 LOD，好处就是对那些离得远，看不清的物体的细节可以忽略。
  - 遮挡剔除（Occlusion culling）
  - 使用 mobile 版的 shader。因为简单。

3.20.2 优化显存带宽

- 第二个方向呢？压缩图片，减小显存带宽的压力。
  - OpenGL ES 2.0 使用 ETC1 格式压缩等等，在打包设置那里都有。
  - 使用 mipmap。
- 上面是一个 mipmap 如何储存的例子，左边的主图伴有一系列逐层缩小的备份小图
- 是不是很一目了然呢？Mipmap 中每一个层级的小图都是主图的一个特定比例的缩小细节的复制品。因为存了主图和它的那些缩小的复制品，所以内存占用会比之前大。但是为何又优化了显存带宽呢？因为可以根据实际情况，选择适合的小图来渲染。所以，虽然会消耗一些内存，但是为了图片渲染的质量（比压缩要好），这种方式也是推荐的。

3.21 内存的优化

- 既然要聊 Unity3D 运行时候的内存优化，那我们自然首先要知道 Unity3D 游戏引擎是如何分配内存的。大概可以分成三大部分：
  - Unity3D 内部的内存
  - Mono 的托管内存
  - 若干我们自己引入的 DLL 或者第三方 DLL 所需要的内存。
- 第 3 类不是我们关注的重点，所以接下来我们会分别来看一下 Unity3D 内部内存和 Mono 托管内存，最后还将分析一个官网上 Assetbundle 的案例来说明内存的管理。

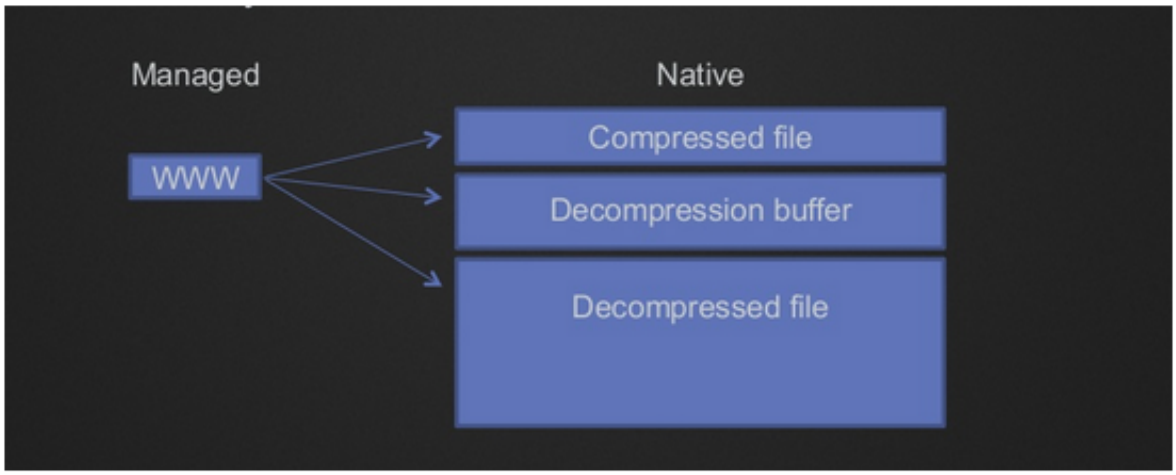
3.21.1 Unity3D 内部内存

- Unity3D 的内部内存都会存放一些什么呢？各位想一想，除了用代码来驱动逻辑，一个游戏还需要什么呢？对，各种资源。所以简单总结一下 Unity3D 内部内存存放的东西吧：
  - 资源：纹理、网格、音频等等
  - GameObject 和各种组件。
  - 引擎内部逻辑需要的内存：渲染器，物理系统，粒子系统等等

3.21.2 Mono 托管内存

- 因为我们的游戏脚本是用 C# 写的，同时还要跨平台，所以带着一个 Mono 的托管环境显然必须的。那么 Mono 的托管内存自然就不得不放到内存的优化范畴中进行考虑。那么我们所说的 Mono 托管内存中存放的东西和 Unity3D 内部内存中存放的东西究竟有何不同呢？其实 Mono 的内存分配就是很传统的运行时内存的分配了：
  - 值类型：int 型啦，float 型啦，结构体 struct 啦，bool 啦之类的。它们都存放在堆栈上（注意额，不是堆所以不涉及 GC）。
  - 引用类型：其实可以狭义的理解为各种类的实例。比如游戏脚本中对游戏引擎各种控件的封装。其实很好理解，C# 中肯定要有对应的类去对应游戏引擎中的控件。那么这部分就是 C# 中的封装。由于是在堆上分配，所以会涉及到 GC。

- 而 Mono 托管堆中的那些封装的对象，除了在在 Mono 托管堆上分配封装类实例化之后所需要的内存之外，还会牵扯到其背后对应的游戏引擎内部控件在 Unity3D 内部内存上的分配。
- 举一个例子：一个在.cs 脚本中声明的 WWW 类型的对象 www，Mono 会在 Mono 托管堆上为 www 分配它所需要的内存。同时，这个实例对象背后的所代表的引擎资源所需要的内存也需要被分配。
- 一个 WWW 实例背后的资源：
  - 压缩的文件
  - 解压缩所需的缓存
  - 解压缩之后的文件
- 如图：



- 那么下面就举一个 AssetBundle 的例子：

3.21.3 Assetbundle 的内存处理

- 以下载 Assetbundle 为例子，聊一下内存的分配。匹夫从官网的手册上找到了一个使用 Assetbundle 的情景如下：

```
IEnumerator DownloadAndCache (){
        // Wait for the Caching system to be ready
        while (!Caching.ready)
            yield return null;
        // Load the AssetBundle file from Cache if it exists with the same version or download
        using(WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)){
            yield return www; //WWW 是第 1 部分
            if (www.error != null)
                throw new Exception("WWW download had an error:" + www.error);
            AssetBundle bundle = www.assetBundle;//AssetBundle 是第 2 部分
            if (AssetName == "")
                Instantiate(bundle.mainAsset);//实例化是第 3 部分
            else
                Instantiate(bundle.Load(AssetName));
                    // Unload the AssetBundles compressed contents to conserve memory
                    bundle.Unload(false);
        } // memory is freed from the web stream (www.Dispose() gets called implicitly)
    }
}
```

- 内存分配的三个部分匹夫已经在代码中标识了出来：
  - Web Stream：包括了压缩的文件，解压所需的缓存，以及解压后的文件。

  - AssetBundle：Web Stream 中的文件的映射，或者说引用。
  - 实例化之后的对象：就是引擎的各种资源文件了，会在内存中创建出来。

- 那就分别解析一下：

```
WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)
```

- 将压缩的文件读入内存中
- 创建解压所需的缓存
- 将文件解压，解压后的文件进入内存
- 关闭掉为解压创建的缓存

```
AssetBundle bundle = www.assetBundle;
```

- AssetBundle 此时相当于一个桥梁，从 Web Stream 解压后的文件到最后实例化创建的对象之间的桥梁。
- 所以 AssetBundle 实质上是 Web Stream 解压后的文件中各个对象的映射。而非真实的对象。
- 实际的资源还存在 Web Stream 中，所以此时要保留 Web Stream。

```
Instantiate(bundle.mainAsset);
```

- 通过 AssetBundle 获取资源，实例化对象
- 最后各位可能看到了官网中的这个例子使用了：

```
using(WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)){ }
```

- 这种 using 的用法。这种用法其实就是为了在使用完 Web Stream 之后，将内存释放掉的。因为 WWW 也继承了 idispose 的接口，所以可以使用 using 的这种用法。其实相当于最后执行了：

```
//删除 Web Stream
www.Dispose();
```

- OK,Web Stream 被删除掉了。那还有谁呢？对 Assetbundle。那么使用

```
//删除 AssetBundle
bundle.Unload(false);
```

- 尽可能地减少 Drawcall 的数量。
- 减少的方法主要有如下几种：Frustum Culling ， Occlusion Culling ， Texture Packing 。
  - Frustum Culling 是 Unity 内建的，我们需要做的就是寻求一个合适的远裁剪平面；
  - Occlusion Culling ， 遮挡剔除，Unity 内嵌了 Umbra ， 一个非常好 OC 库。
  - 但 Occlusion Culling 也并不是放之四海而皆准的，有时候进行 OC 反而比不进行还要慢，
  - 建议在 OC 之前先确定自己的场景是否适合利用 OC 来优化；Texture Packing ， 或者叫 Texture Atlasing ，
  - 是将同种 shader 的纹理进行拼合，根据 Unity 的 static batching 的特性来减少 draw call 。
  - 建议使用，但也有弊端，那就是一定要将场景中距离相近的实体纹理进行拼合，否则，拼合后很可能会增加每帧渲染所需的纹理大小，加大内存带宽的负担。这也就是为什么会出现” DrawCall 降了，渲染速度也变慢了” 的原因。