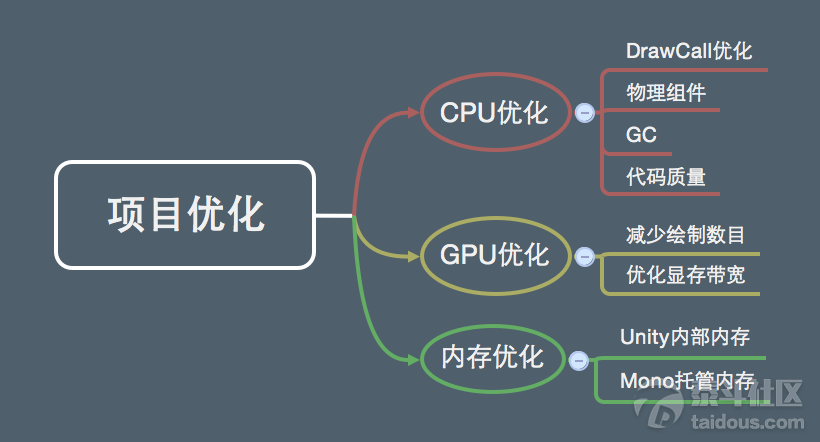
****前言****

Unity的项目优化已经是老生常谈，很多人在项目完成之后，即便创意新颖，也会觉得差强人意，原因就在于没有做详细的项目优化。众所周知，Unity是一个综合性的3D开发引擎，其中包含图像渲染，逻辑处理，数据存储，发布测试等等各方面的内容。因此Unity各个方面都存在的待优化的内容，也可以说项目优化是项目开发中必不可少的一项工作。本篇文章会从项目的各个方面分析Unity待优化的内容，并给出优化方案，全面优化你的项目。优化项目无非是减轻系统的功耗负担，故下面从CPU、GPU、内存三方面的优化来讲解。



大纲

CPU优化

DrawCalls

什么是DrawCall？

Unity每次在准备数据并通知GPU渲染的过程称为一次DrawCall。

为什么要优化DrawCall？

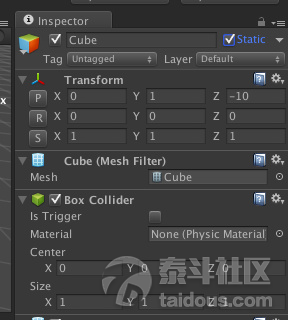
因为通知给GPU这个工作是由CPU来完成的，完成每一次DrawCall都需要CPU完成很多的内容，因此如果DrawCall很多的话CPU一定是不堪重负。但对于GPU而言，很多的工作都是一样的，也就是说很多的DrawCall是没有意义的，可以理解为很多次的DrawCall都可以合并为一次，这样减轻了CPU的负担，同样也可以通知GPU完成相应的工作。因此减少DrawCall成了优化CPU的第一要务。

降低DrawCall第一步-->Draw Call Batching(批处理)

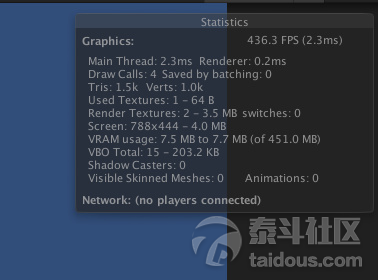
两个或多个纹理相同或材质相同的网格模型可以批量处理他们的材质，这样就可以将多个模型的材质DrawCall合并为一个，从而达到减少DrawCall的目的。批处理是系统工作范畴，我们只需要选择即可，批处理又分为静态批处理和动态批处理。

Static Batching 静态批处理

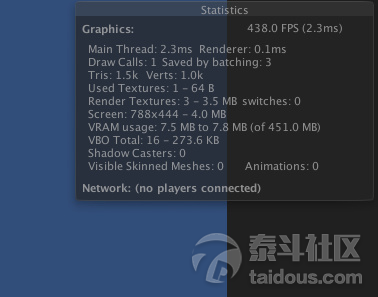
场景中有很多游戏对象，其中静态对象（Inspector勾选Static的）可以通过静态批处理来优化DrawCall。

   
静态游戏对象

下面我们通过具体实例来验证：  
在场景中创建四个游戏对象Cube，Sphere, Capsule, Cylinder，默认为非静态。此时运行游戏，DrawCall如下。

   
非静态四个对象的DrawCall为4

此时，DrawCall为4，Saved by batching为0。  
然后，我们将四个对象都设置成静态Static，再看DrawCall。

   
设置成静态之后，通过静批DrawCall变为1

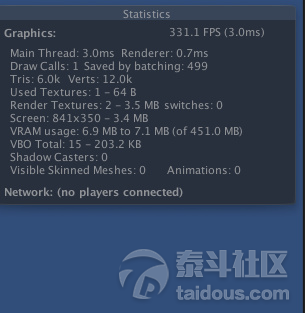
此时，DrawCall变为1，Saved by batching变为3。  
当然，目前我们只有四个对象，如果在项目中超多的相同材质静态对象或相同纹理的静态对象，DrawCall减少量更大，优化做的也就会更好。

Dynamic Batching 动态批处理

上面的静态批处理需要给对象设置成静态，而动态批处理，则不需要，非静态的对象，系统自动做批处理。  
同样，我们举例说明，通过预设体创建500个cube。

for(int i = 0; i < 500; i++){    GameObject cube = GameObject.Instantiate(cubePefab) as GameObject;}

DrawCall的数量为：

   
动态批处理后的DrawCall数量

DrawCall为1，Saved by batching变为499，是不是很爽。  
然而，事实上动态批处理有很多约束。我们同样是创建500个物体，不同的是其中的100个物体，每个物体的大小都不同，也就是Scale不同。

[AppleScript] 纯文本查看 复制代码

for(int i = 0; i < 500; i++)

{

GameObject cube = GameObject.Instantiate(cubePefab) as GameObject;

}

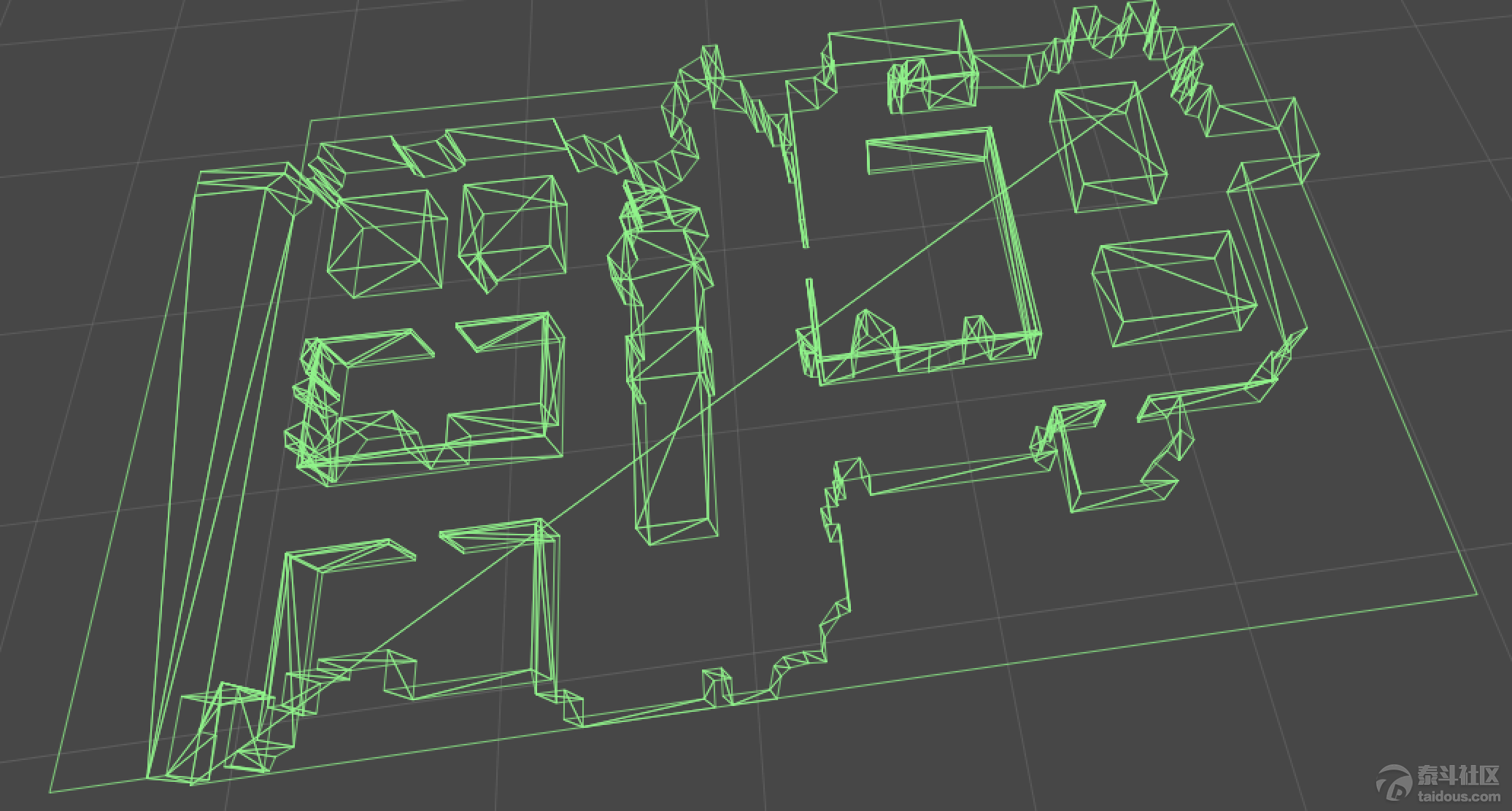
此时，DrawCall的数量：

   
变换缩放后的动批

DrawCall变为101，Saved by batching变为399，OhMyGod，小小的变换一下缩放，就不能批处理了，简直有点不智能。所以，在使用动批的时候，需要注意一下，如果你的动批不起作用可能是一下原因。

* 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。  
  所以动态批处理还是约束蛮多的，尽量使用静态批处理。
* 物理组件

物理组件是我们在游戏开发中经常用到的组件，比起设计超级复杂的伤害计算算法，一个Trigger就能解决很多问题，还有那可以模拟一切物理效果的Rigidbody。但如果物理组件使用过多，计算量过大，也会造成CPU过载。对于物理组件的优化，有以下两点。

* + 设置一个合适的Fixed Timestep。  
    我们都知道在计算物理逻辑的时候通常会将代码放到FixedUpdate里面，然而FixedUpdate的执行频率，就由Fixed Timestep决定，并不是所有的游戏中物理计算都需要0.02秒执行一次。因此这个值，可以针对项目慢慢调试，设置出一个比较合适的值，这样即完成了物理计算，也可以减轻CPU的负担。所以，想想自己上高中物理的时候做的物理大题，让CPU少做几道，确实可以轻松很多。换位思考也可以这样对吧，换到CPU的角度考虑。
  + 不要使用MeshCollider  
    原因很简单，因为MeshCollider太复杂了，想起我当年做秘密行动的时候一次次的崩溃，也是醉了，就是因为加了一个超大的MeshCollider，不夸张的说真是要一分钟崩溃一次，更严重的是崩溃完之后，场景文件都没有了，可见其威力巨大。  
      
    秘密行动中的MeshCollider  
      
    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。
* GC(Garbage Collection垃圾回收)

GC是用来处理内存的，为什么会影响到CPU的开销呢？因为GC是CPU调度的。大量的调用GC确实可以回收内存，但如果内存占用量不是很大的情况下，调用GC的性价比就很低，因为GC对CPU的开销所造成的代价更大。所以优化GC，就是减少对GC的调用。

  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。  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。

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。
* 代码质量

代码质量优化老生常谈，这里简单提几点：

* + 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:  
    1.GetComponent = 619ms  
    2.Monobehaviour = 60ms  
    3.CachedMB = 8ms  
    4.Manual Cache = 3ms  
    所以最好不要频繁使用GetComponent，尤其是在循环中。
  + 善于使用OnBecameVisible()和OnBecameVisible(),来控制物体的update()函数的执行以减少开销。
  + 使用内建的数组，比如用Vector3.zero而不是new Vector(0, 0, 0);
  + 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。

GPU优化

GPU主要处理图像渲染，与CPU不同，侧重点自然也不同。GPU需要优化的点主要有以下几点：

* + 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）
  + GPU的显存带宽

  针对上面的汇总，可以得知GPU的优化无非两点。 减少绘制的数目和优化显存带宽。

* + 减少绘制的数目

优化方案很简单，减少绘制的数目，无非是减少顶点数量，简化复杂度，举措如下。

* + - 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，简单。
  + 优化显存带宽

再有就是压缩图片，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  
        
      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。

内存优化

谈到内存优化，首先要看一下，Unity开发中需要用到哪些内存。

* + 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。（非重点）
  + Unity内部内存

Unity3D的内部内存都会存放一些什么呢？

* + - 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
  + 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。

****结束语****

项目优化至关重要，因为现在的产品大同小异的很多，游戏也是如此，无非就是那么几种类型，在保持创新精神的基础上，还要着重关注的就是用户体验。用户体验这个词不是新词了，所以怎样提高用户体验呢，流畅、明朗、便捷。当然，这些都取决与项目优化。所以，是你的产品就好好优化它，因为每个产品都可以变得更好。