# draw call 理解和优化

　　draw call是openGL的描绘次数（directX没怎么研究，但原理应该差不多）

　　一个简单的openGL的绘图次序是：设置颜色→绘图方式→顶点座标→绘制→结束。

　　每帧都会重复以上的步骤。这就是一次draw call

　　如果有两个model，那么需要

　　设置颜色→绘图方式→顶点座标A→绘制→结束。

　　设置颜色→绘图方式→顶点座标B→绘制→结束。

　　两次draw calls;

　　也就是说在openGl绘制前，如果色彩通道(color filter)，绘图方式(shader)，顶点座标（model）不同的情况下draw calls就会增加。

　　对openGl来说绘制参数（状态值）的变更要比绘制大量的顶点更耗费cpu。

　　所谓高速绘图就是，在尽量不改变openGl状态值的情况下，用一次draw call完成所有绘制。

　　比如上面的例子：

　　设置颜色→绘图方式→顶点座标A＋顶点座标B→绘制→结束。

　　就要更加有效率。

　　个人估计unity3d的dynamic batch,static batch都是通过一定的方法使不同的object的顶点座标能够结合成一个整体，达到减少draw calls的效果。

　　但是有一定的要求限制，比如material要相同，mesh要相同并在300个面以内等等，这些都是为了保证openGl的状态值不改变。

　　Unity在 Player Setting 里的两个功能选项 Static Batching 与 Dynamic Batching。功能描述如下：

　　Static Batching 是将标明为 Static 的静态物件，如果在使用相同材质球的条件下，Unity 会自动帮你把这两个物件合并成一个 Batch，送往 GPU 来处理。这功能对效能上非常的有帮助，所以是需要付费才有的。

　　Dynamic Batching 是在物件小于300面的条件下(不论物件是否为静态或动态)，在使用相同材质球下，Unity就会自动帮你合合并成一个 Batch 送往 GPU 来处理。

**Unity3D项目优化--绘制调用批处理unity3D Draw**

　　在屏幕上渲染物体，引擎需要发出一个绘制调用来访问图形API（iOS系统中为OpenGL ES）。每个绘制调用需要进行大量的工作来访问图形API，从而导致了CPU方面显著的性能开销。

　　Unity在运行时可以将一些物体进行合并，从而用一个绘制调用来渲染他们。这一操作，我们称之为“批处理”。一般来说，Unity批处理的物体越多，你就会得到越好的渲染性能。

　　Unity中内建的批处理机制所达到的效果要明显强于使用几何建模工具（或使用Standard Assets包中的CombineChildren脚本）的批处理效果。这是因为，Unity引擎的批处理操作是在物体的可视裁剪操作之后进行的。Unity先对每个物体进行裁剪，然后再进行批处理，这样可以使渲染的几何总量在批处理前后保持不变。但是，使用几何建模工具来拼合物体，会妨碍引擎对其进行有效的裁剪操作，从而导致引擎需要渲染更多的几何面片。

**材质**

　　只有拥有相同材质的物体才可以进行批处理。因此，如果你想要得到良好的批处理效果，你需要在程序中尽可能地复用材质和物体。

　　如果你的两个材质仅仅是纹理不同，那么你可以通过 纹理拼合 操作来将这两张纹理拼合成一张大的纹理。一旦纹理拼合在一起，你就可以使用这个单一材质来替代之前的两个材质了。

　　如果你需要通过脚本来访问复用材质属性，那么值得注意的是改变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。

**动态批处理**

　　如果动态物体共用着相同的材质，那么Unity会自动对这些物体进行批处理。

　　动态批处理操作是自动完成的，并不需要你进行额外的操作。

**Tips:**

**提醒：**

1、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。

2、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。

3、请注意：属性数量的限制可能会在将来进行改变。

4、不要使用缩放尺度（scale）。分别拥有缩放尺度(1,1,1)和(2,2,2)的两个物体将不会进行批处理。

5、统一缩放尺度的物体不会与非统一缩放尺度的物体进行批处理。

　　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。

6、使用不同材质的实例化物体（instance）将会导致批处理失败。

7、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）。

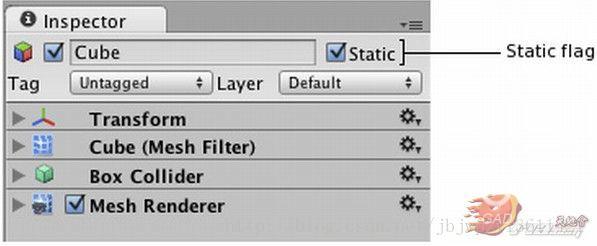
8、多通道的shader会妨碍批处理操作。比如，几乎unity中所有的着色器在前向渲染中都支持多个光源，并为它们有效地开辟多个通道。

9、预设体的实例会自动地使用相同的网格模型和材质。

**静态批处理**

　　相对而言，静态批处理操作允许引擎对任意大小的几何物体进行批处理操作来降低绘制调用（只要这些物体不移动，并且拥有相同的材质）。因此，静态批处理比动态批处理更加有效，你应该尽量低使用它，因为它需要更少的CPU开销。

　　为了更好地使用静态批处理，你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，如下图所示：



　　使用静态批处理操作需要额外的内存开销来储存合并后的几何数据。在静态批处理之前，如果一些物体共用了同样的几何数据，那么引擎会在编辑以及运行状态对每个物体创建一个几何数据的备份。这并不总是一个好的想法，因为有时候，你将不得不牺牲一点渲染性能来防止一些物体的静态批处理，从而保持较少的内存开销。比如，将浓密森里中树设为Static，会导致严重的内存开销。

　　静态批处理目前只支持Unity iOS Advanced。

**Unity3D - 性能优化之Draw Call**

　　nity（或者说基本所有图形引擎）生成一帧画面的处理过程大致可以这样简化描述：引擎首先经过简单的可见性测试，确定摄像机可以看到的物体，然后把这些物体的顶点（包括本地位置、法线、UV等），索引（顶点如何组成三角形），变换（就是物体的位置、旋转、缩放、以及摄像机位置等），相关光源，纹理，渲染方式（由材质/Shader决定）等数据准备好，然后通知图形API——或者就简单地看作是通知GPU——开始绘制，GPU基于这些数据，经过一系列运算，在屏幕上画出成千上万的三角形，最终构成一幅图像。

　　在Unity中，每次引擎准备数据并通知GPU的过程称为一次Draw Call。这一过程是逐个物体进行的，对于每个物体，不只GPU的渲染，引擎重新设置材质/Shader也是一项非常耗时的操作。因此每帧的Draw Call次数是一项非常重要的性能指标，对于iOS来说应尽量控制在20次以内，这个值可以在编辑器的Statistic窗口看到。

　　Unity内置了Draw Call Batching技术，从名字就可以看出，它的主要目标就是在一次Draw Call中批量处理多个物体。只要物体的变换和材质相同，GPU就可以按完全相同的方式进行处理，即可以把它们放在一个Draw Call中。Draw Call Batching技术的核心就是在可见性测试之后，检查所有要绘制的物体的材质，把相同材质的分为一组（一个Batch），然后把它们组合成一个物体（统一变换），这样就可以在一个Draw Call中处理多个物体了（实际上是组合后的一个物体）。

　　但Draw Call Batching存在一个缺陷，就是它需要把一个Batch中的所有物体组合到一起，相当于创建了一个与这些物体加起来一样大的物体，与此同时就需要分配相应大小的内存。这不仅会消耗更多内存，还需要消耗CPU时间。特别是对于移动的物体，每一帧都得重新进行组合，这就需要进行一些权衡，否则得不偿失。但对于静止不动的物体来说，只需要进行一次组合，之后就可以一直使用，效率要高得多。

　　Unity提供了Dynamic Batching和Static Batching两种方式。Dynamic Batching是完全自动进行的，不需要也无法进行任何干预，对于顶点数在300以内的可移动物体，只要使用相同的材质，就会组成Batch。Static Batching则需要把静止的物体标记为Static，然后无论大小，都会组成Batch。如前文所说，Static Batching显然比Dynamic Batching要高效得多，于是，Static Batching功能是收费的……

　　要有效利用Draw Call Batching，首先是尽量减少场景中使用的材质数量，即尽量共享材质，对于仅纹理不同的材质可以把纹理组合到一张更大的纹理中（称为Texture Atlasing）。然后是把不会移动的物体标记为Static。此外还可以通过CombineChildren脚本（Standard Assets/Scripts/Unity Scripts/CombineChildren）手动把物体组合在一起，但这个脚本会影响可见性测试，因为组合在一起的物体始终会被看作一个物体，从而会增加GPU要处理的几何体数量，因此要小心使用。

　　对于复杂的静态场景，还可以考虑自行设计遮挡剔除算法，减少可见的物体数量同时也可以减少Draw Call。

　　总之，理解Draw Call和Draw Call Batching原理，根据场景特点设计相应的方案来尽量减少Draw Call次数才是王道，其它方面亦然。

**U3D DrawCall优化手记**

　　http://www.cnblogs.com/ybgame/p/3588795.html

　　在最近，使用U3D开发的游戏核心部分功能即将完成，中间由于各种历史原因，导致项目存在比较大的问题，这些问题在最后，恐怕只能通过一次彻底的重构来解决

　　现在的游戏跑起来会有接近130-170个左右的DrawCall，游戏运行起来明显感觉到卡，而经过一天的优化，DrawCall成功缩减到30-70个，这个效果是非常显著的，并且这个优化并没有通过将现有的资源打包图集来实现，图集都是原有的图集，如果从全局的角度对图集再进行一次优化，那么DrawCall还可以再减少十几个

**本次优化的重点包括：层级关系和特效**

　　对于U3D，我是一个菜鸟，对于U3D的一些东西是一知半解，例如DrawCall，我得到的是一些并不完全正确的信息，例如将N个纹理打包成一个图集，这个图集就只会产生一个DrawCall，如果不打成图集，那么就会有N个DrawCall，这个观点在很多人的认识里都是正确的，因为可以通过简单的操作来验证，但严格来说，这个观点是错误的，因为它还受层级关系影响！

**渲染顺序**

　　U3D的渲染是有顺序的，U3D的渲染顺序是由我们控制的，控制好U3D的渲染顺序，你才能控制好DrawCall

　　一个DrawCall，表示U3D使用这个材质/纹理，来进行一次渲染，那么这次渲染假设有3个对象，那么当3个对象都使用这一个材质/纹理的时候，就会产生一次DrawCall，可以理解为一次将纹理输送到屏幕上的过程，（实际上引擎大多会使用如双缓冲，缓存这类的手段来优化这个过程，但在这里我们只需要这样子认识就可以了），假设3个对象使用不同的材质/纹理，那么无疑会产生3个DrawCall

　　接下来我们的3个对象使用2个材质，A和B使用材质1，C使用材质2，这时候来看，应该是有2个DrawCall，或者3个DrawCall。应该是2个DrawCall啊，为什么会有3个DrawCall？？？而且是有时候2个，有时候3个。我们按照上面的DrawCall分析流程来分析一下：

1、渲染A，使用材质1

2、渲染B，使用材质1

3、渲染C，使用材质2

　　在这种情况下是2个DrawCall，在下面这种情况下，则是3个DrawCall

1、渲染A，使用材质1

2、渲染C，使用材质2

3、渲染B，使用材质1

　　因为我们没有控制好渲染顺序（或者说没有去特意控制），所以导致了额外的DrawCall，因为A和B不是一次性渲染完的，而是被C打断了，所以导致材质1被分为两次渲染

　　那么是什么在控制这个渲染顺序呢？首先在多个相机的情况下，U3D会根据相机的深度顺序进行渲染，在每个相机中，它会根据你距离相机的距离，由远到近进行渲染，在UI相机中，还会根据你UI对象的深度进行渲染

　　那么我们要做的就是，对要渲染的对象进行一次规划，正确地排列好它们，规则是，按照Z轴或者深度，对空间进行划分，然后确定好每个对象的Z轴和深度，让使用同一个材质的东西，尽量保持在这个空间内，不要让其他材质的对象进入这个空间，否则就会打断这个空间的渲染顺序

　　在这个基础上，更细的规则有：

　　场景中的东西，我们使用Z轴来进行空间的划分，例如背景层，特效层1，人物层，特效层2NGUI中的东西，我们统一使用Depth来进行空间的划分人物模型，当人物模型只是用一个材质，DrawCall只有1，但是用了2个以上的材质，DrawCall就会暴增（或许对材质的RenderQueue进行规划也可以使DrawCall只有2个，但这个要拆分好才行），3D人物处于复杂3D场景中的时候，我们的空间规则难免被破坏，这只能在设计的时候尽量去避免这种情况了

　　使用了多个材质的特效，在动画的过程中，往往会引起DrawCall的波动，在视觉效果可以接受的范围内，可以将特效也进行空间划分，假设这个特效是2D显示，那么可以使用Z轴来划分空间

**打包图集**

　　每个材质/纹理的渲染一定是会产生DrawCall的，这个DrawCall只能通过打包图集来进行优化

　　制作图集一般遵循几个规则：

　　从功能角度进行划分，例如UI可以划分为公共部分，以及每个具体的界面，功能上，显示上密切相关的图片打包到一起不要一股脑把所有东西打包到一个图集里，特别是那些不可能同时出现的东西，它们就不应该在一个图集里，这样的图集意义不大，减少不了DrawCall，并且一个你不需要显示的图片，会一直占用你的内存，这让我非常不爽注意控制图集的大小，不要让图集太大，一个超级大图集的DrawCall消耗或许顶的上十几个小图集的消耗字符图集，在使用BMFont或者其他工具生成图片字的时候，我们往往是直接导入一大串文字，然后直接生成图片，但实际上这上面的操作也有优化空间，例如BMFont生成的图片大小，是可以设置的，有两个规则，一个规则是导出的图片尽量小，另一个是导出的图片尽量少，默认的大小应该是512x512，假设你生成的图片256x256就可以容纳，那么多做一个操作你可以节省这么多空间，另外当你输入多几个字，就导致增加一张图片时，例如1024变成2048，那么你可以考虑使用3张512的图片，这样也会节省空间经过精心划分的图集在加上精心规划的渲染顺序，DrawCall会有一个质的优化

**特效清理**

　　U3D提供了非常便捷的方法让我们很轻易地使用美术给过来的特效，懒惰的U3D程序猿会直接放入U3D，甚至不去看这是个什么特效，我们的特效一般都是一瞬间的事情，例如技能特效，或者其他什么特效，那么特效播放完，这个特效我们就看不到了，但假设这个特效在播放结束的时候，没有将自身的Active属性设置为false，那么它就会继续占用你的DrawCall，消耗你设备的计算能力，所以程序需要保证当一个特效播放完之后，能够被消耗，或者设置为非激活的状态，可以使用一些公共方法来完成特效播放完之后的清理工作（自己实现2个静态函数，一个播放完销毁，一个播放完设置未激活）

　　完成DrawCall的优化之后，接下来就是内存的优化了，（内存优化手记 待续）