# UGUI性能优化的一些建议

## 一．背景和目的

目前越来越多的unity项目组在UI解决方案的选型上，用的是原生的uGUI，看了网络上各种大神提供的经验，在此进行一个记录与总结。

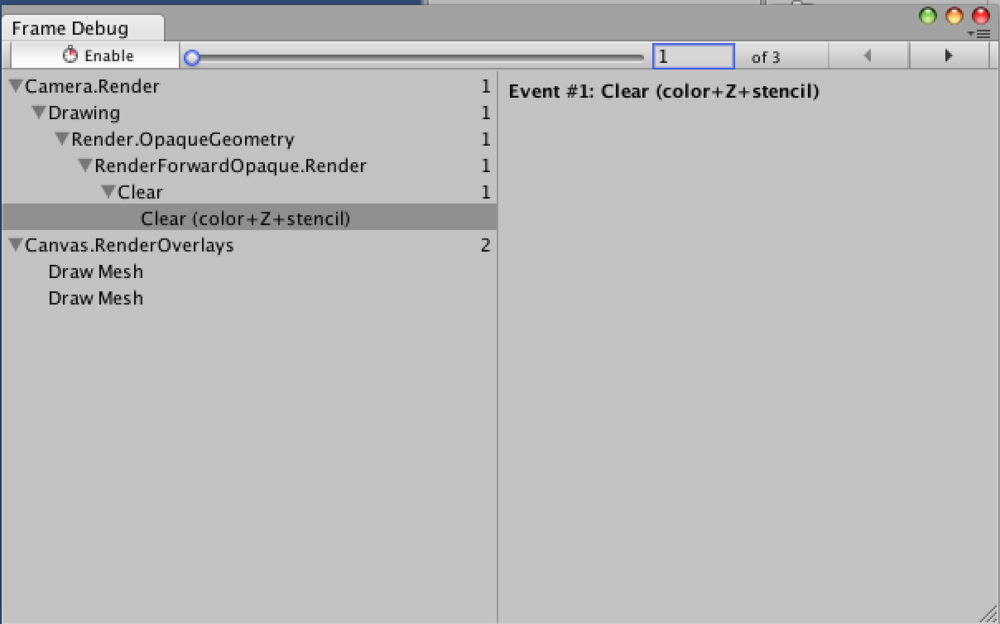
本文接下来将会对uGUI的**Runtime性能**进行着重讨论，其它的因素也很多而且很重要，但是一篇文章讲清楚一件事就好了，以后还有uGUI的实践与使用技巧的时候，再作分享。

**二．影响uGUI性能的因素与检查工具**

游戏中的UI与其它游戏中的元素本质上是一样的，相对来说的不同点在于，**UI通常是由2D的图片组合而成，会包含较多的透明元素与渐变元素，而且一般来说会显示在屏幕的最顶层**。

因此，从共同点上说，UI的Runtime性能消耗也可以划分为CPU消耗、GPU消耗与内存消耗。其中对于每一部分的具体的消耗以及优化，有诸多大神在网络上发表过文章，比如[深入浅出聊Unity3D项目优化：从Draw Calls到GC](https://link.jianshu.com/?t=http://blog.jobbole.com/84323/)。但是，总有一个大家都绕不开的点，那就是Drawcall，Drawcall数量直接影响了游戏的帧率，解决了Drawcall问题，应该算是解决了80%的问题，所以接下来就着重针对uGUI的特性讲一讲UI系统的Drawcall。

Unity 5.0版本之后在Drawcall查看方面有一个非常有用的工具，Frame Debugger，通过[Window->Frame Debugger]打开。



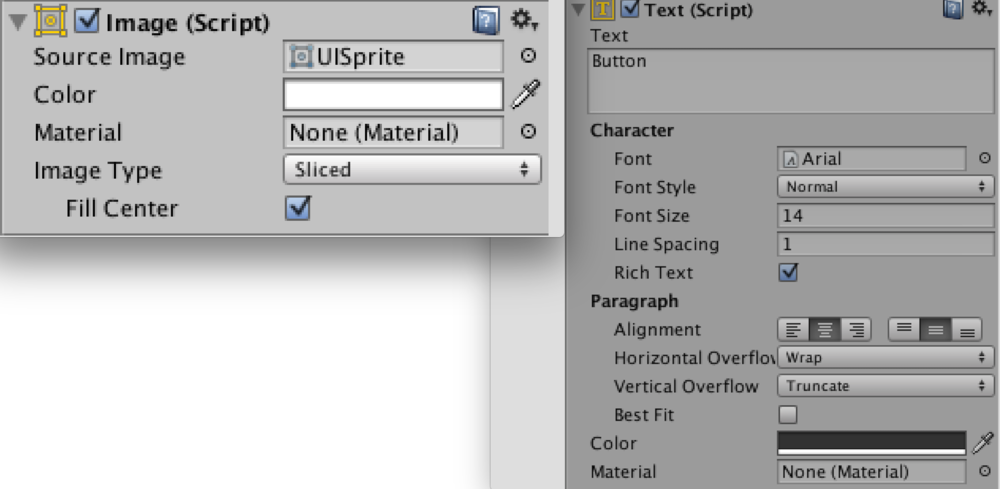
Frame Debugger(Only in Unity5.0+)

使用该工具时，游戏会暂停，然后Unity会将当前正在执行的一帧的内容缓存下来，其中所有Drawcall你都可以进行前进与后退操作，从而能够从Drawcall级别分析开销。

此外，在用FD看UI性能时，有一个**小窍门**就是新开一个空的Scene，然后将你的UI Prefab拖到该空场景中，此时就不会受场景中其它物体的影响而只显示UI的Drawcall了。

**三．uGUI性能优化**

在降低Drawcall方面，一个非常重要的概念就是Batch，因为一次Drawcall相当于CPU与GPU进行一次沟通的成本，如果CPU能一次多打包一些信息给GPU，那么Drawcall数量自然就下来了，这个打包传输信息给GPU的过程就叫做Batch，批处理。那么什么情况下这些信息可以打包呢？**从uGUI的角度，如果你的UI中组件的材质与纹理均相同，这几个组件就可以被Batch**。在Image组件中，材质对应Source Image，纹理则对应Material；在Text组件中材质对应Font，纹理也是Material。以上对应大部分情况适用，在少部分特殊shader下会失效（待深入研究）。



Common UI Components

原理是这样，但是实际用起来还需要一些技巧，遵循Unity的一些渲染次序的规则，才能真正的实现性能优化。以下就一一进行讨论。

**四．打包图集**

上面有提到Source Image图集，所谓的图集，就是将好多张零碎的2D小图片通过Unity自带的Sprite Packer或第三方的Texture Packer合并到一张大图，这样做有几大好处，

1. 图片尺寸为2的次幂时，GPU处理起来会快很多，小图自己是做不到每张图都是2的次幂的，但打成一张大图就可以（浪费一点也无所谓）；
2. CPU在传送资源信息给GPU时，只需要传一张大图就可以了，因为GPU可以在这张图中的不同区域进行采样，然后拼出对应的界面。注意，这就是为什么需要用同一个Source Image图集的原因，是Batch的关键，因为一个Drawcall就把所有原材料传过去了，GPU你画去吧。

但是显然把所有图片打成一张图集是不合理的，因为这张图可能非常大，所以就要按照一定规则将图片进行分类。在分类思路上，我们希望做到Drawcall尽可能少，同时资源量也尽可能少（多些重用），但这两者某种程度上是互斥的，所以折衷一下，可以遵循以下思路：

* 设计UI时要考虑重用性，如一些边框、按钮等，这些作为共享资源，放在1~3张大图集中，称为**重用图集**；
* 其它非重用UI按照功能模块进行划分，每个模块使用1~2张图集，为**功能图集**；
* 对于一些UI，如果同时用到**功能图集**与**重用图集**，但是其**功能图集**剩下的“空位”较多，则可以考虑将用到的**重用图集**中的元素单独拎出来，合入**功能图集**中，从而做到让UI只依赖于**功能图集**。也就是通过一定的冗余，来达到性能的提升。

P.S. 如果你用Unity自带的Sprite Packer去打包图集，那么你可能要在运行模式下才能看到效果。

**五．Unity GUI层级合并规则与批次生成规则**

uGUI的层叠顺序是按照Hierarchy中的顺序从上往下进行的，也就是越靠上的组件，就会被画在越底部。所以UI就是这样一层一层地叠上去画出来的。当然这样一个一个地画效率肯定是不能接受的，所以要合并，要Batch，Unity自身就提供了一个算法去决定哪些层应该合并到一起，并以什么样的顺序进行绘制。**所有相邻层的可Batch的UI元素将会在一个Drawcall完成**。接下来就来讨论一下Unity的层级合并与计算算法。

Unity的UI渲染顺序的确定有2个步骤，第一步计算每个UI元素的**层级号**；第二步合并相同**层级号**中可以Batch的元素作为一个**批次**，并对**批次**进行排序；

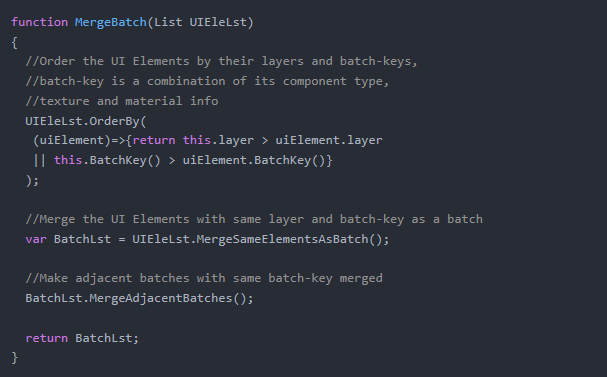
先从直观的角度来解释**计算层级号的算法**：如果有一个UI元素，它所占的屏幕范围内（通常是矩形），如果没有任何UI在它的底下，那么它的**层级号**就是0（最底下）；如果有一个UI在其底下且该UI可以和它Batch，那它的**层级号**与底下的UI层级一样；如果有一个UI在其底下但是无法与它Batch，那它的**层级号**为底下的UI的层级+1；如果有多个UI都在其下面，那么按前两种方式遍历计算所有的**层级号**，其中最大的那个作为自己的**层级号**。

这里也给一下伪代码，假设所有UI元素（抛弃层级关系）都按从上往下的顺序被装在一个list中，那么每个UI元素对应的**层级号**计算可以参考以下：



有了层级号之后，就要**合并批次**了，此时，Unity会将每一层的所有元素进行一个排序（按照材质、纹理等信息），合并掉可以Batch的元素成为一个**批次**，**目前已知的排序规则是，Text组件会排在Image组件之前渲染**，而同一类组件的情况下排序规则未知（好像并没什么规则）。**经过以上排序，就可以得到一个有序的批次序列了**。这时，**Unity会再做一个优化，即如果相邻间的两个批次正好可以Batch的话就会进行Batch**。举个栗子，一个层级为0的ImageA，一个层级为1的ImageB（2个Image可Batch）和一个层级为0的TextC，Unity排序后的批次为TextC->ImageA->ImageB，后两个批次可以合并，所以是2个Drawcall。再举个栗子，一个层级为0的TextD，一个层级为1的TextE（2个Text可Batch）和一个层级为0的ImageF，Unity排序后的批次为TextD->ImageF->TextE，这时就需要3个Drawcall了！（是不是有点晕，再回顾下黑体字）

以下的伪代码有些偷懒，实在懒得写排序、合并之类的，一长串也不好读，几个步骤列一下，其它看上面那段文字脑补下吧...



根据以上规则，就可以得出一些“摆UI”的技巧：

* 有相同材质和纹理的UI元素是可以Batch的，可以Batch的UI上下叠在一块不会影响性能，但是如果不能Batch的UI元素叠在一块，就会增加Drawcall开销。
* 要注意UI元素间的层叠关系，建议用“T”工具查看其矩形大小，因为有些图片透明，但是却叠在其它UI上面了，然后又无法Batch的话，就会无故多许多Drawcall；
* UI中出现最多的就是Image与Text组件，当Text叠在Image上面（如Button)，然后Text上又叠了一个图片时，就会至少多2个Drawcall，可以考虑将字体直接印在下面的图片上；
* 有些情况可以考虑人为增加层级从而减少Drawcall，比如一个Text的层级为0，另一个可Batch的Text叠在一个图片A上，层级为1，那此时2个Text因为层级不同会安排2个Drawcall，但如果在第一个Text下放一个透明的图片（与图片A可Batch），那两个Text的层级就一致了，Drawcall就可以减少一个。

### 六．少用Mask

Mask对于uGUI性能来说是噩梦一般的存在，因为很可能因为这个东西，导致Drawcall数量成倍增长。

Mask实现的具体原理是一个Drawcall来创建Stencil mask(来做像素剔除)，然后画所有子UI，再在最后一个Drawcall移掉Stencil mask。这头尾两个Drawcall无法跟其他UI操作进行Batch，所以表面上看加个Mask就会多2个Drawcall，但是，因为Mask这种类似“汉堡包式”的渲染顺序，所有Mask的子节点与其他UI其实已经处在两个世界了，上面提到的层级合并规则只能分别作用于这两个世界了，所以很多原本可以合并的UI就无法合并了。

所以，在使用uGUI时，有一些建议：

* 应该尽量避免使用Mask，其实Mask的功能有些时候可以变通实现，比如设计一个边框，让这个边框叠在最上面，底下的UI移动时，就会被这个边框遮住；
* 如果要使用Mask时，需要评估下Mask会带来的性能损耗，并尽量将其降到最低。比如Mask内的UI是动态生成的话（比如List组件），那么需要注意UI之间是否有重叠的现象。

# 七．总结

uGUI的性能其实涉及到的方面很多，这里列出来的只是目前能想到的，因为个人能力有限，可能出些纰漏。对于文中的一些建议，这里整理一下得出一些最佳实践：

* 设计UI时要考虑重用性，如一些边框、按钮等，这些作为共享资源，放在1~3张大图集中，称为**重用图集**；
* 其它非重用UI按照功能模块进行划分，每个模块使用1~2张图集，为**功能图集**；
* 对于一些UI，如果同时用到**功能图集**与**重用图集**，但是其**功能图集**剩下的“空位”较多，则可以考虑将用到的**重用图集**中的元素单独拎出来，合入**功能图集**中，从而做到让UI只依赖于**功能图集**。也就是通过一定的冗余，来达到性能的提升。
* 有相同材质和纹理的UI元素是可以Batch的，可以Batch的UI上下叠在一块不会影响性能，但是如果不能Batch的UI元素叠在一块，就会增加Drawcall开销。
* 要注意UI元素间的层叠关系，建议用“T”工具查看其矩形大小，因为有些图片透明，但是却叠在其它UI上面了，然后又无法Batch的话，就会无故多许多Drawcall；
* UI中出现最多的就是Image与Text组件，当Text叠在Image上面（如Button)，然后Text上又叠了一个图片时，就会至少多2个Drawcall，可以考虑将字体直接印在下面的图片上；
* 有些情况可以考虑人为增加层级从而减少Drawcall，比如一个Text的层级为0，另一个可Batch的Text叠在一个图片A上，层级为1，那此时2个Text因为层级不同会安排2个Drawcall，但如果在第一个Text下放一个透明的图片（与图片A可Batch），那两个Text的层级就一致了，Drawcall就可以减少一个。
* 应该尽量避免使用Mask，其实Mask的功能有些时候可以变通实现，比如设计一个边框，让这个边框叠在最上面，底下的UI移动时，就会被这个边框遮住；
* 如果要使用Mask时，需要评估下Mask会带来的性能损耗，并尽量将其降到最低。比如Mask内的UI是动态生成的话（像List组件），那么需要注意生成的UI之间是否有重叠的现象；
* 有空好好看下**Unity GUI层级合并规则与批次生成规则**这一节。