UGUI性能相关

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，相关文章，比如[深入浅出聊Unity3D项目优化：从Draw Calls到GC](http://blog.jobbole.com/84323/)。总有一个绕不开的点，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

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

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。

**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）。

具体操作：

1.打包图集：

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。

UI的摆放顺序：略麻烦

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；

**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。
* 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。

# 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；

**我们要做的事**

1.对于atlas总来的来说就是让同一时刻，UI中平均包含的atlas尽可能少。  
经常用的图片要打成一个atlas。  
2.其他atlas按功能划分。比如，背包界面，除了公用图集中出现的元素以外，所有出现的图片都打成一个atlas。  
3.可重用的组件尽量在前期就都考虑到，因为到后期可能会产生混乱，同一个按钮在A界面出现也在B界面出现。（当然除了为了减少draw call这么做以外）  
4.如果在某种情况下对顶点数限制不高，可以考虑将图片做成对称的，然后用原图的一半拼出想要的效果

5.对于drawcall来说，我们UI中有N层，N越大越靠前，会遮住后面的组件。我们从0到N数，然后看每个组件的材质，我们要尽量让这个材质变化的次数减少。  
6.相同图集的组件在不影响最终效果的情况下尽量使得相同图集的组件相邻。  
7.相同字体的文本控件在不影响最终效果的情况下劲量使得相同字体的组件相邻。  
8.对于不同的图集尽量不要去穿插，字体也是一样（纵向交叉，Herichy中的顺序）。  
9.如果UI元素可以没有交叉区域尽量不交叉，特别是文字。（横向交叉，矩形区域的交叉）  
10.如果可能的话可以把层叠交叉的文字做到图片中

11.如果不用mask可以实现的功能尽量不用。Mask组件用到了透明，所以他会被绘制两次。所以，带mask的组件不适用dynamic batching的规则。

1. 尽可能保证Text在Image的上方。字体的DrawCall就可能自动会优化，而不需要像NGUI那样让字体跟图片pack在一起来减少DrawCall。
2. 尽可能保证Image之间不存在不必要的重叠。

UI变色：修改材质和修改顶点色两种方式

1. 合并DrawCall一定要是相同的材质，修改材质会将本来已合并的DrawCall分离开，就导致了DrawCall的增加及Canvas的重建。
2. 修改顶点色（Button中Transition中的ColorTint方式）只会修改Canvas中的数据。

UI边框：Sliced & Fill Center

1. Sliced模式的Sprite更节省纹理尺寸。
2. 中空的边框不应该勾选Fill Center，可减小Fill rate。

Mask的代价，通过Stencil buffer实现

1. 移动平台需要Use 24-bit Depth Buffer（Tegra GPU 2、3上不支持，4支持）。
2. Mask中的UI元素无法与其他batch，从而增加DrawCall。
3. 可以的话尝试用Filled Sprite代替。

动画：Text vs Image

1. 如果Image上做了动画则Canvas需要重建，Image的顶点数一般不会很多，开销不会很大。
2. Text与文本内容（生成的网格数）相关，如果文字比较多则相比Image开销可能会翻倍。

Canvas的重建主要就是为了合并DrawCall，可以将有动画的文字放在单独的Canvas中去，手动分离DrawCall（增加DrawCall）后就不会再要去跟别的文字作合并，该Canvas就不需要再重建故减少了重建开销。

尽可能使用缓冲池。如人物身上的伤害数字，生成时会有较大的开销因为有Mesh的生成等过程。

unity官网对性能优化的建议，干货，比较实用，建议都看一下：https://unity3d.com/cn/learn/tutorials/topics/best-practices/guide-optimizing-unity-ui?playlist=30089

翻译：

http://www.manew.com/home.php?mod=space&uid=105915&do=thread&view=me&from=space

以上链接中的系列文章：UI 优化指南，UNITY UI 基本原理， Unity UI分析工具， 填充率，画布和输入， 优化UI控件， 其他的UI优化技术和技巧。

# 如何快速优化手游性能问题？从UGUI优化说起

http://www.mamicode.com/info-detail-1639308.html

优化文章 http://blog.csdn.net/poem\_of\_sunshine/article/details/52577414

UGUI大概使用浏览 http://blog.csdn.net/poem\_of\_sunshine/article/details/52577414

注意：图集不能放入Resources中不然不会进行打包

1.图集打包：命名规范，格式灵活（也可以自定义打包相关设置）可以参考momo相关博客。目前打算使用默认

2图集分类

项目中UI管理 ViewManager UI基类 ViewBase

目前只分析View相关不考虑 tip

Start（）初始化方法：iViewList初始化 继承链表用于存储View的容器

iCurrent 当前的View

iBackView 上一个view

PopView（view） 删去最外面的view，当删去的view是当前的view 当前view设置为链表末尾的view，并显示出来。大富翁中当购买界面Destroy之后，显示出SceneView就是调用了此方法（具体调用与ViewBase中也有关联）

PushView（bReplace,view，...）首先将当前界面赋值为backView。当压入一个view时候，第一个参数来控制上一个view是Hide还是Destroy。然后创建view，view不为空，隐藏当前的界面，把view的OnDestroyFinish委托赋值Popview(具体调用与ViewBase中也有关联)

链表中压入view，将当前的界面赋值为view

最后是如果view不为空，如果backView 不为空并且view的名字不等于backView的名字 iBackView进行构造，key为view的名字，value为backView的名字和参数？

否则 backView刷新

界面管理使用的容器是链表 System/list中提供具体接口

增加两个接口：lua字符串拼接不可以用‘+’

--打开界面不关闭上层界面

function ShowView(view, ...)

view = SafeDoFunc(CreateView, view, ...)

iView[view.viewName] = view

logError("ViewName"..iView[view.viewName].viewName)

if view then

iViewList:push(view)

view.OnDestroyFinish = PopView

iCurrent = view

end

return view

end

--打开界面关闭互斥界面

function ShowViewCloseOtherView(view,otherView,...)

if otherView then

logError(otherView)

logError("OtherViewName"..iView[otherView].viewName)

local iter = iViewList:find(iView[otherView])

if iter then

iView[otherView]:Destroy()

iView[otherView]=nil

end end

end

view = SafeDoFunc(CreateView, view, ...)

if view then

iViewList:push(view)

view.OnDestroyFinish = PopView

iCurrent = view

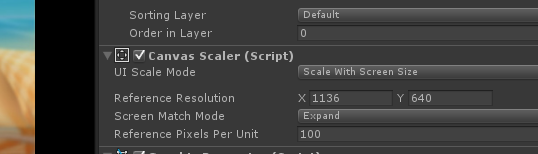
end

return view

end

UI制作流程：

设置画布分辨率为



背景图锚点使用stretch

元素UI 还原真实分辨率 或者等比例缩放 或者锚点