关于离屏渲染的深入研究

Jason Yu Follow
Jul 5 · 15 min read

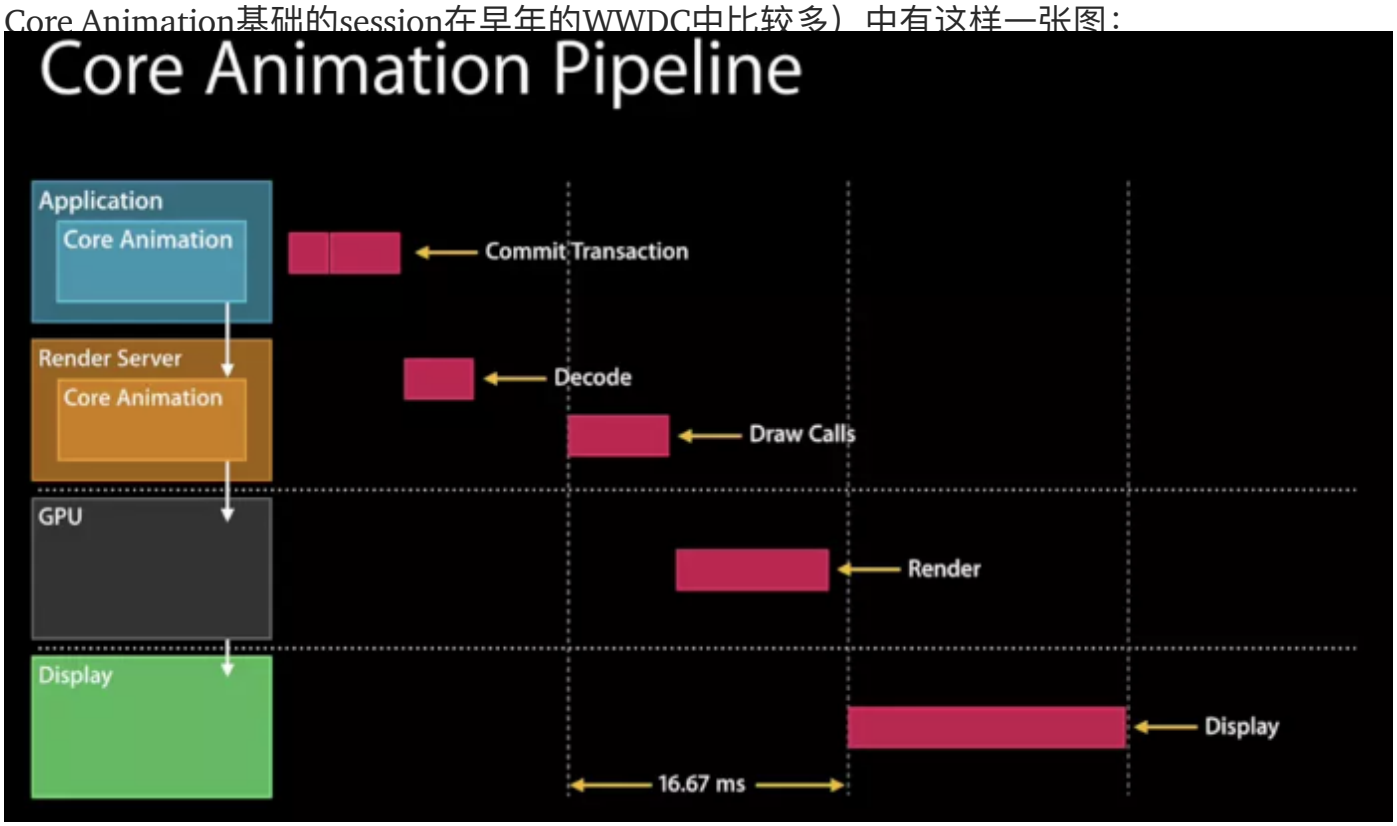
在平时的iOS面试中，我们经常会考察有关离屏渲染（Offscreen rendering）的知识点。一般来说，绝大多数人都能答出“圆角、mask、阴影会触发离屏渲染”，但是也仅止于此。如果再问得深入哪怕一点点，比如：

- 离屏渲染是在哪一步进行的？为什么？
- 设置cornerRadius一定会触发离屏渲染吗？

90%的候选人都没法非常确定地说出答案。作为一个客户端工程师，把控渲染性能是最关键、最独到的技术要点之一，如果仅仅了解表面知识，到了实际应用时往往会失之毫厘谬以千里，无法得到预期的效果。

iOS渲染架构

在WWDC的Advanced Graphics and Animations for iOS Apps（WWDC14 419，关于UIKit和Core Animation基础的session在早年的WWDC中比较多）中有这样一张图：



我们可以看到，在Application这一层中主要是CPU在操作，而到了Render Server这一层，CoreAnimation会将具体操作转换成发送给GPU的draw calls（以前是call OpenGL ES，现在慢慢转到了Metal），显然CPU和GPU双方同处于一个流水线中，协作完成整个渲染工作。

离屏渲染的定义

如果要在显示屏上显示内容，我们至少需要一块与屏幕像素数据量一样大的frame buffer，作为像素数据存储区域，而这也是GPU存储渲染结果的地方。如果有时因为面临一些限制，无法把渲染结果直接写入frame buffer，而是先暂存在另外的内存区域，之后再写入frame buffer，那么这个过程被称之为离屏渲染。

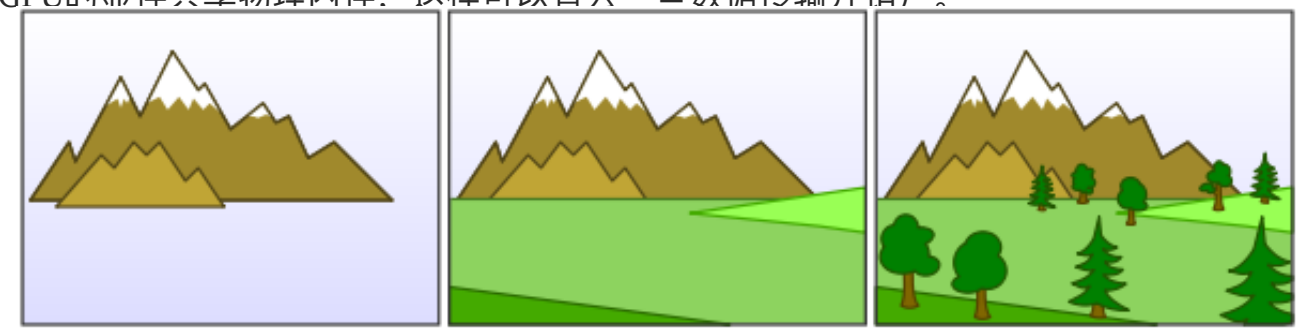
渲染结果先经过了离屏buffer，再到frame buffer

CPU"离屏渲染"

大家知道，如果我们在UIView中实现了drawRect方法，就算它的函数体内部实际没有代码，系统也会为这个view申请一块内存区域，等待CoreGraphics可能的绘画操作。
对于类似这种"新开一块CGContext来画图"的操作，有很多文章和视频也称之为"离屏渲染"（因为像素数据是暂时存入了CGContext，而不是直接到了frame buffer）。进一步来说，其实所有CPU进行的光栅化操作（如文字渲染、图片解码），都无法直接绘制到由GPU掌管的frame buffer，只能暂时先放在另一块内存之中，说起来都属于"离屏渲染"。
自然我们会认为，因为CPU不擅长做这件事，所以我们需要尽量避免它，就误以为这就是需要避免离屏渲染的原因。但是根据苹果工程师的说法，CPU渲染并非真正意义上的离屏渲染。另一个证据是，如果你的view实现了drawRect，此时打开Xcode调试的"Color offscreen rendered yellow"开关，你会发现这片区域不会被标记为黄色，说明Xcode并不认为这属于离屏渲染。
其实通过CPU渲染就是俗称的"软件渲染"，而**真正的离屏渲染发生在GPU**。

GPU离屏渲染

在上面的渲染流水线示意图中我们可以看到，主要的渲染操作都是由CoreAnimation的Render Server模块，通过调用显卡驱动所提供的OpenGL/Metal接口来执行的。通常对于每一层layer，Render Server会遵循"画家算法"，按次序输出到frame buffer，后一层覆盖前一层，就能得到最终的显示结果（值得一提的是，与一般桌面架构不同，在iOS中，设备主存和GPU的显存共享物理内存，这样可以省去一些数据传输开销）。



"画家算法"，把每一层依次输出到画布，先画的层会被一定程度覆盖

然而有些场景并没有那么简单。作为"画家"的GPU虽然可以一层一层往画布上进行输出，但

是无法在某一层渲染完成之后，再回过头来擦除/改变其中的某个部分——因为在当前层之前的若干层layer像素数据，已经在这次渲染中被永久覆盖，无法恢复了。这就意味着，**对于每一层layer，要么能找到一种通过单次遍历就能完成渲染的算法（最快最好的情况），要么就不得不另开一块内存，借助这个临时中转区域来完成一些复杂的、多次的修改/剪裁操作**。因此，离屏渲染实际上是一种无奈之举，以空间为代价，来换取一些高级操作的可能性。

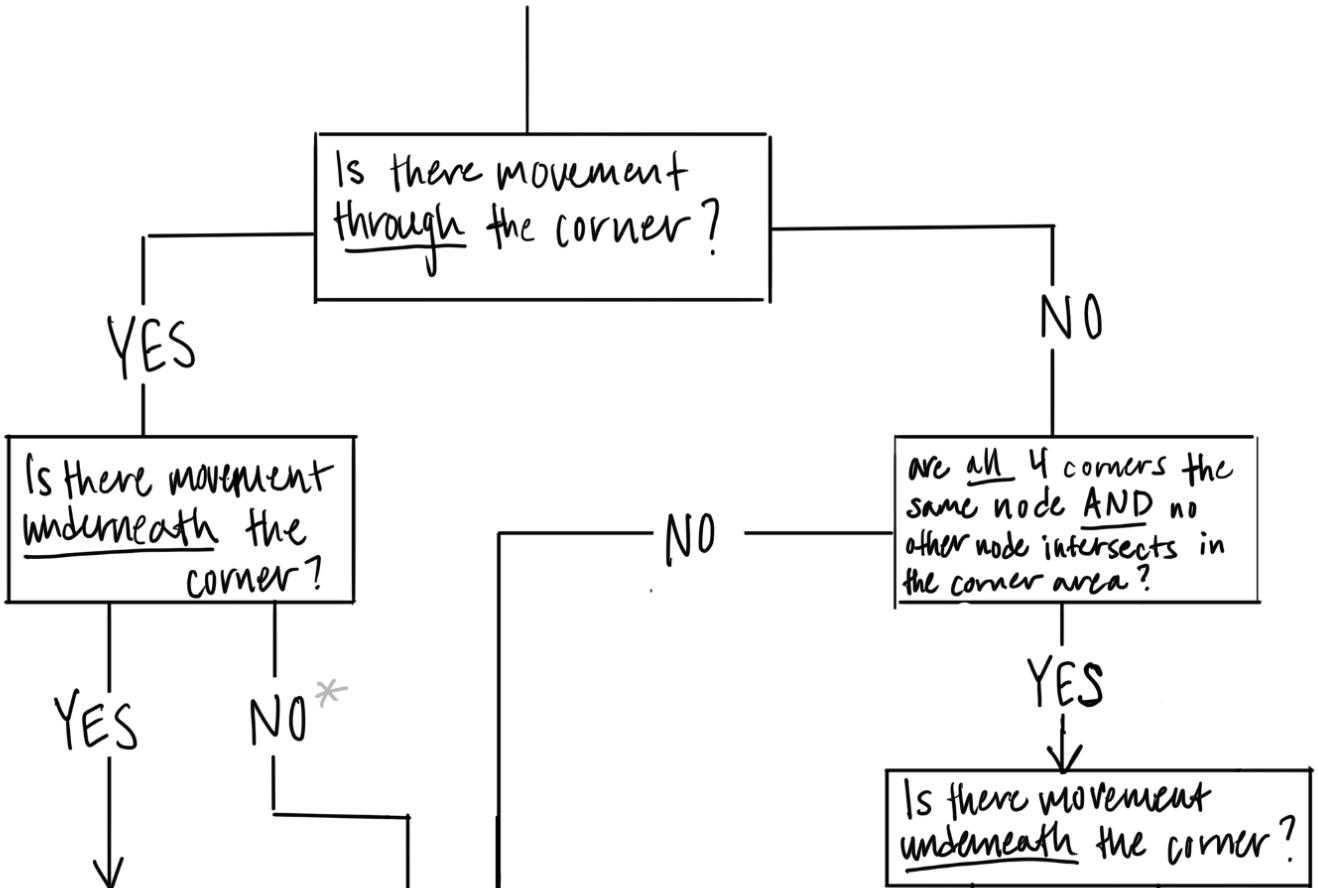
例如，如果要绘制一个带有圆角并剪切圆角以外内容的容器，就会触发离屏渲染。我的猜想是（如果读者中有图形学专家希望能指正）：

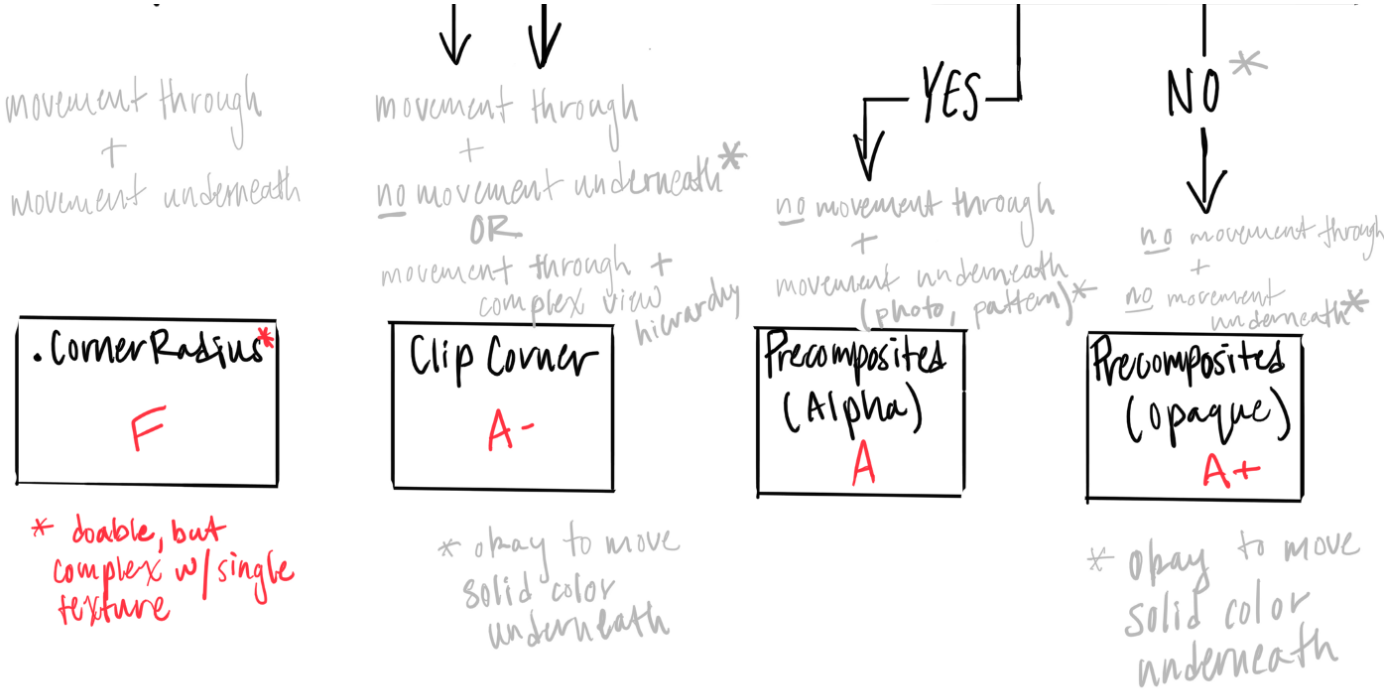
- 将一个layer的内容裁剪成圆角，可能不存在一次遍历就能完成的方法
- 容器的子layer因为父容器有圆角，那么也会需要被裁剪，而这时它们还在渲染队列中排队，尚未被组合到同一块画布上，自然也无法统一裁剪

此时我们就不得不开辟一块独立于frame buffer的空白内存，先把容器以及其所有子layer依次画好，然后把四个角“剪”成圆形，再把结果画到frame buffer中。这就是GPU的离屏渲染。

常见离屏渲染场景分析

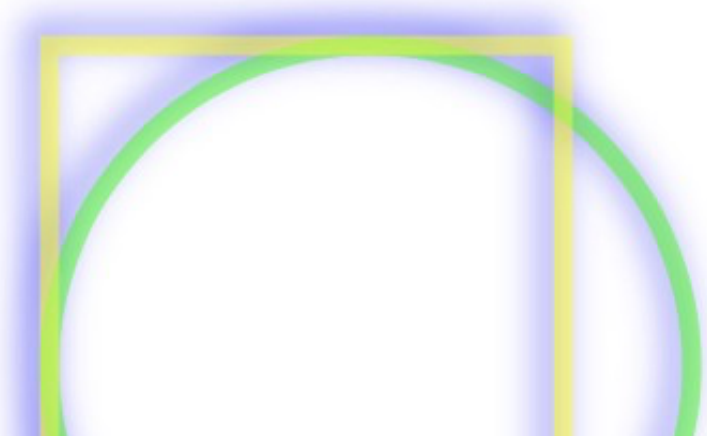
- cornerRadius+clipsToBounds，原因就如同上面提到的，不得已只能另开一块内存来操作。而如果只是设置cornerRadius（如不需要剪切内容，只需要一个带圆角的边框），或者只是需要裁掉矩形区域以外的内容（虽然也是剪切，但是稍微想一下就可以发现，对于纯矩形而言，实现这个算法似乎并不需要另开内存），并不会触发离屏渲染。关于剪切圆角的性能优化，根据场景不同有几个方案可供选择，非常推荐阅读AsyncDisplayKit中的一篇文档。

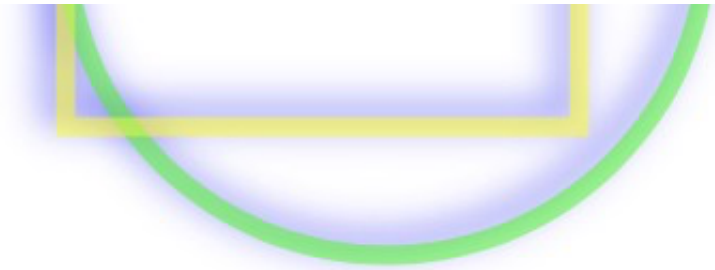




ASDK中对于如何选择圆角渲染策略的流程图，非常实用

- shadow，其原因在于，虽然layer本身是一块矩形区域，但是阴影的形状却未必是矩形，而是与layer中”非透明区域“的形状一致。这就意味着**需要先知道这个形状是什么样的（由layer与其所有子结构组合所决定），阴影只能在这之后得到**。但矛盾的是，阴影需要显示在所有layer内容的下方，那么**根据画家算法，下层的阴影又必须先被渲染**。因为这个矛盾无法被调和，这样一来又只能另外申请一块内存，把本体内容都先画好，再根据渲染结果的形状，添加阴影到frame buffer，最后把内容画上去（这只是我的猜测，实际情况可能更复杂）。不过如果我们能够预先告诉CoreAnimation（通过shadowPath属性）阴影的几何形状，那么阴影当然可以先被独立渲染出来，不需要依赖layer本体，也就不再需要离屏渲染了。



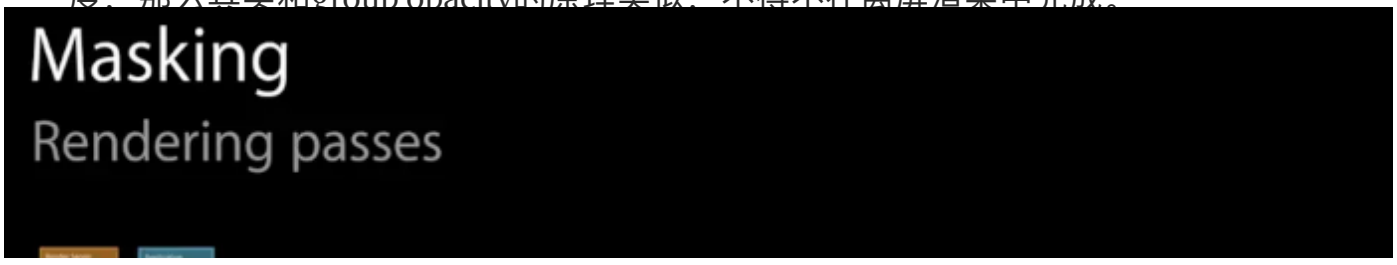


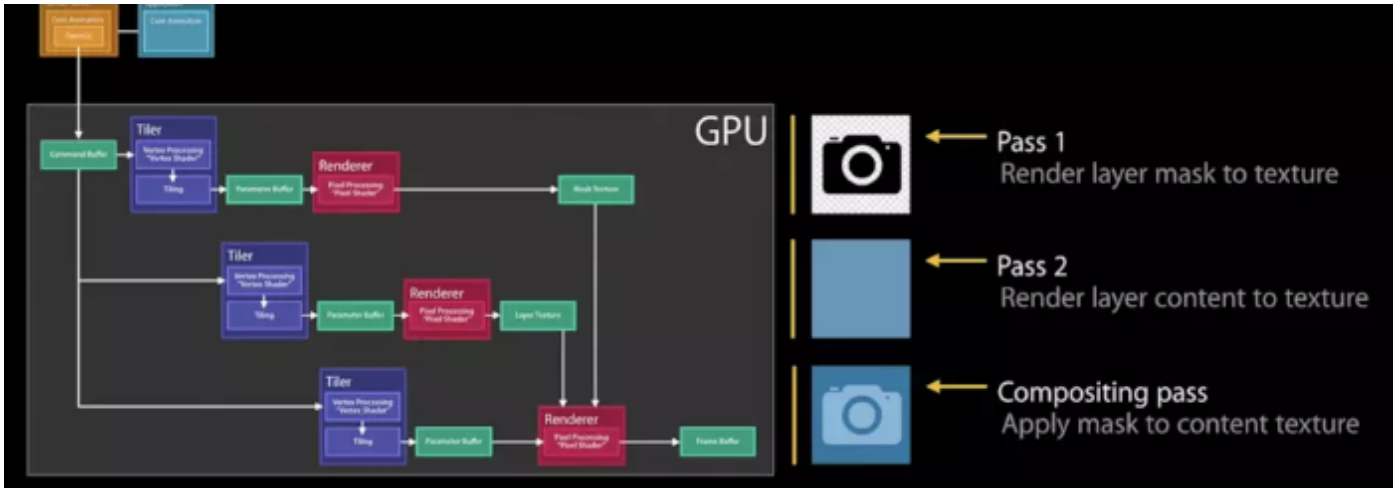
阴影会作用在所有子layer所组成的形状上，那就只能等全部子layer画完才能得到

- group opacity，其实从名字就可以猜到，alpha并不是分别应用在每一层之上，而是只有到整个layer树画完之后，再统一加上alpha，最后和底下其他layer的像素进行组合。显然也无法通过一次遍历就得到最终结果。将一对蓝色和红色layer叠在一起，然后在父layer上设置opacity=0.5，并复制一份在旁边作对比。左边关闭group opacity，右边保持默认（从iOS7开始，如果没有显式指定，group opacity会默认打开），然后打开offscreen rendering的调试．我们会发现右边的那一组确实是离屏渲染了。

同样的两个view，右边打开group opacity（默认行为）的被标记为Offscreen rendering

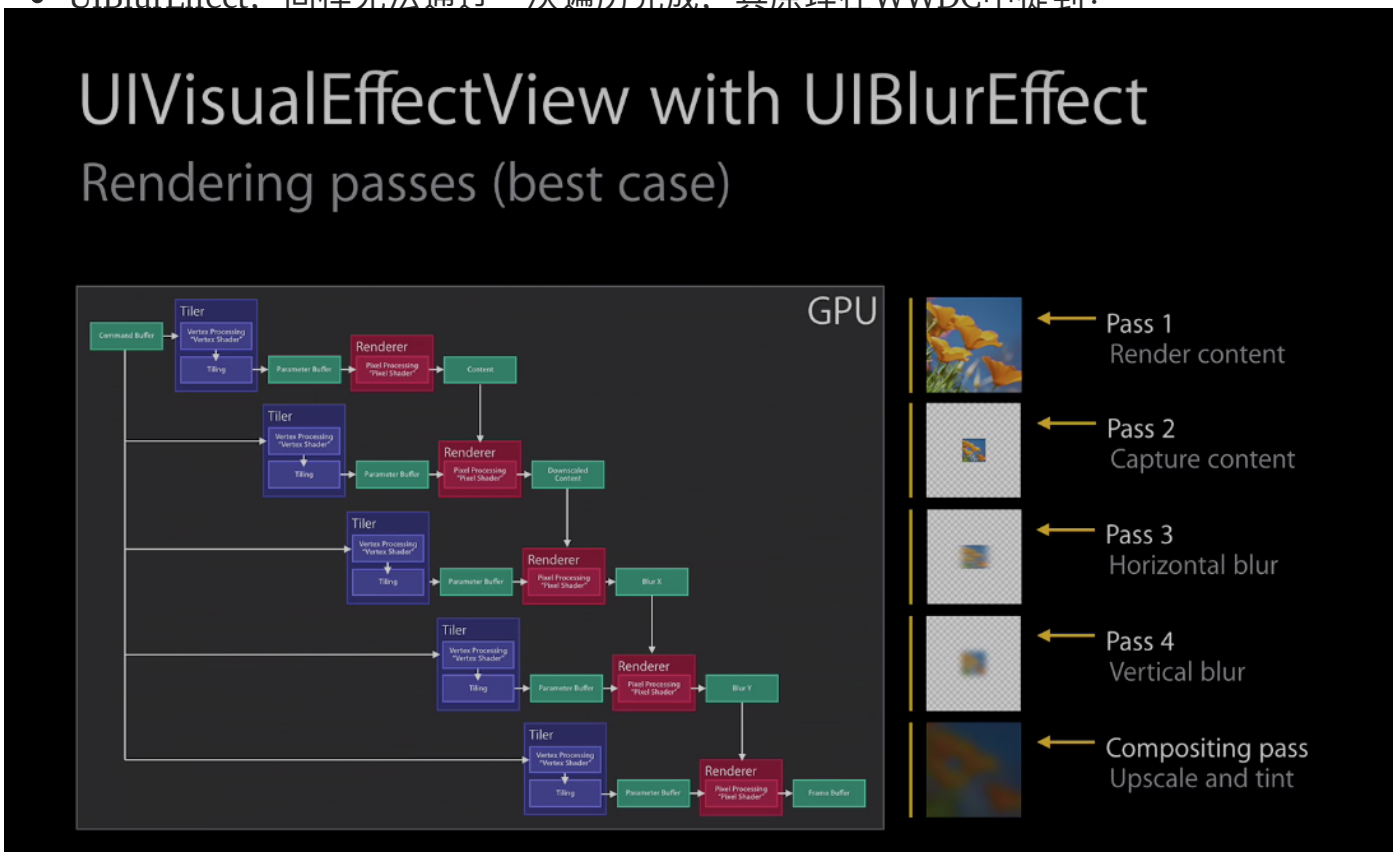
- mask，我们知道mask是应用在layer和其所有子layer的组合之上的，而且可能带有透明度，那么其实和group opacity的原理类似，不得不在离屏渲染中完成。





WWDC中苹果的解释，mask需要遍历至少三次

- UIBlurEffect，同样无法通过一次遍历完成，其原理在WWDC中提到：



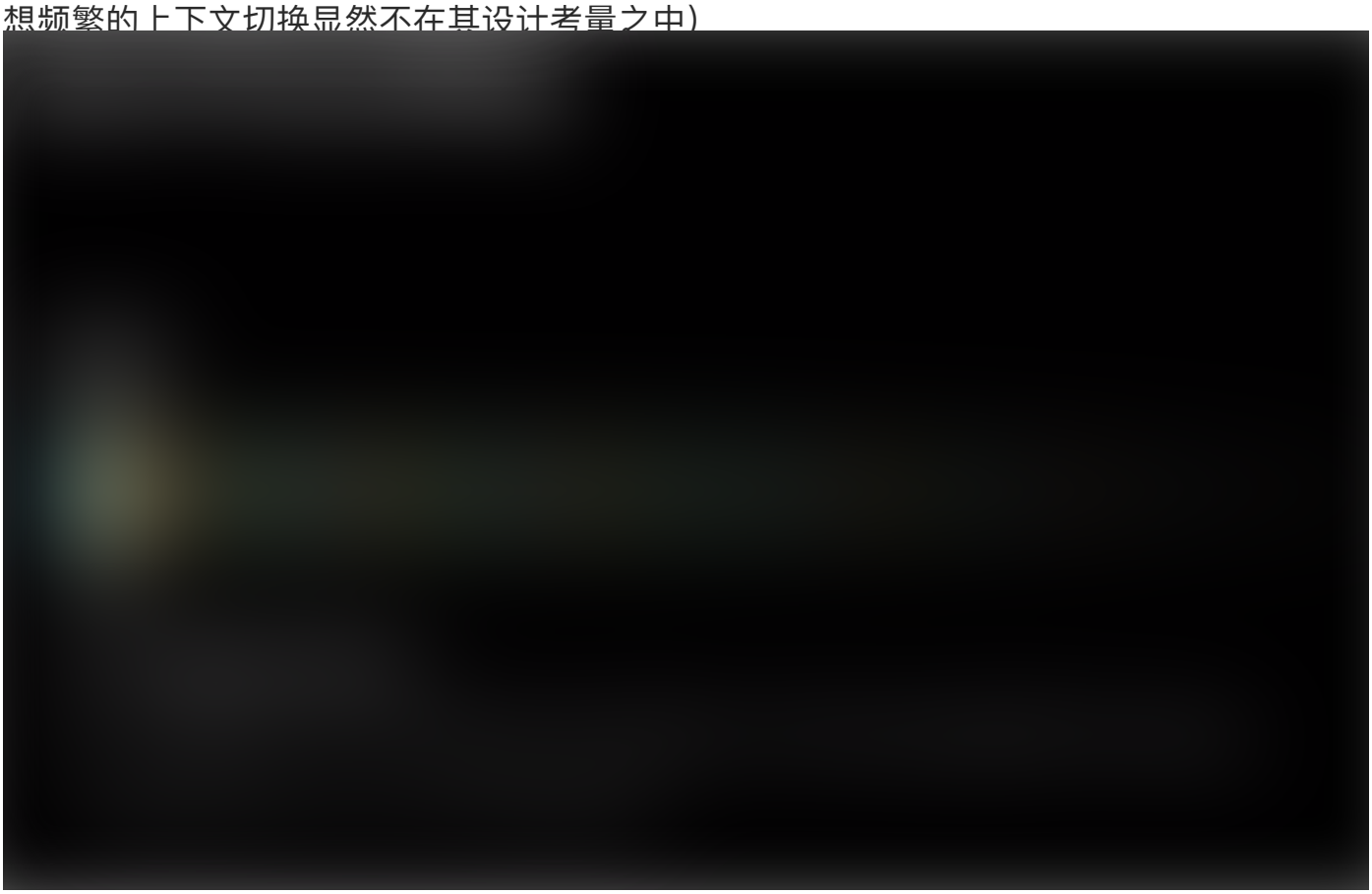
- 其他还有一些，类似allowsEdgeAntialiasing等等也可能会触发离屏渲染，原理也都是类似：如果你无法仅仅使用frame buffer来画出最终结果，那就只能另开一块内存空间来储存中间结果。这些原理并不神秘。

GPU离屏渲染的性能影响

GPU的操作是高度流水线化的。本来所有计算工作都在有条不紊地正在向frame buffer输出，此时突然收到指令，需要输出到另一块内存，那么流水线中正在进行的一切都不得不被丢弃，切换到只能服务于我们当前的“切圆角”操作。等到完成以后再次清空，再回到向frame buffer输出的正常流程。

在tableView或者collectionView中，滚动的每一帧变化都会触发每个cell的重新绘制，因此一旦存在离屏渲染，上面提到的上下文切换就会每秒发生60次，并且很可能每一帧有几十张

的图片要求这么做，对于GPU的性能冲击可想而知（GPU非常擅长大规模并行计算，但是我想频繁的上下文切换显然不在其设计考量之中）



每16ms就需要根据当前滚动位置渲染整个tableView，是个不小的性能挑战

善用离屏渲染

尽管离屏渲染开销很大，但是当我们无法避免它的时候，可以想办法把性能影响降到最低。优化思路也很简单：既然已经花了不少精力把图片裁出了圆角，如果我能把结果缓存下来，那么下一帧渲染就可以复用这个成果，不需要再重新画一遍了。

CALayer为这个方案提供了对应的解法：shouldRasterize。一旦被设置为true，Render Server就会强制把layer的渲染结果（包括其子layer，以及圆角、阴影、group opacity等等）保存在一块内存中，这样一来在下一帧仍然可以被复用，而不会再次触发离屏渲染。有几个需要注意的点：

- shouldRasterize的主旨在于**降低性能损失，但总是至少会触发一次离屏渲染**。如果你的layer本来并不复杂，也没有圆角阴影等等，打开这个开关反而会增加一次不必要的离屏渲染
- 离屏渲染缓存有空间上限，最多不超过屏幕总像素的2.5倍大小
- 一旦缓存超过100ms没有被使用，会自动被丢弃
- layer的内容（包括子layer）必须是静态的，因为一旦发生变化（如resize，动画），之前辛苦处理得到的缓存就失效了。如果这件事频繁发生，我们就又回到了“每一帧都需要离屏渲染”的情景，而这正是开发者需要极力避免的。针对这种情况，Xcode提供了“Color Hits Green and Misses Red”的选项，帮助我们查看缓存的使用是否符合预期
- 其实除了解决多次离屏渲染的开销，shouldRasterize在另一个场景中也可以使用：如果layer的子结构非常复杂，渲染一次所需时间较长，同样可以打开这个开关，把layer绘

制到一块缓存，然后在接下来复用这个结果，这样就不需要每次都重新绘制整个layer树了。

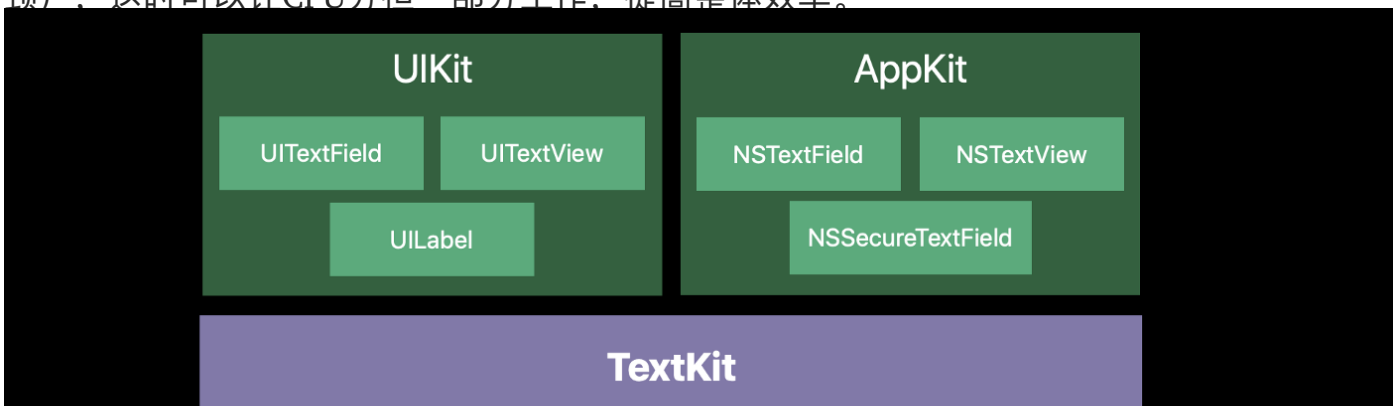
什么时候需要CPU渲染

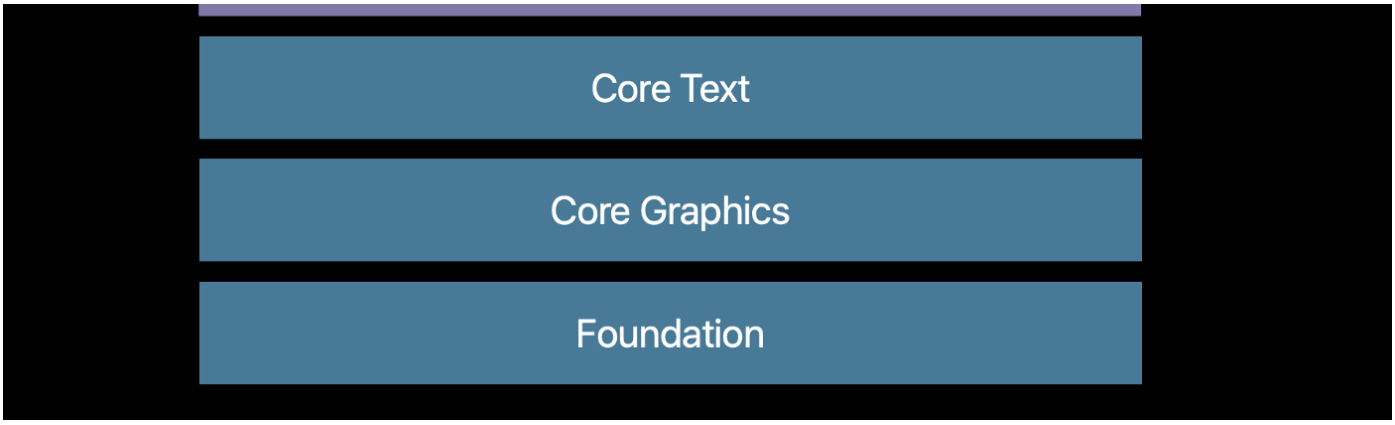
渲染性能的调优，其实始终是在做一件事：**平衡CPU和GPU的负载，让他们尽量做各自最擅长的工作。**

平衡CPU和GPU的负载

绝大多数情况下，得益于GPU针对图形处理的优化，我们都会倾向于让GPU来完成渲染任务，而给CPU留出足够时间处理各种各样复杂的App逻辑。为此Core Animation做了大量的工作，尽量把渲染工作转换成适合GPU处理的形式（也就是所谓的硬件加速，如layer composition，设置backgroundColor等等）。

但是对于一些情况，如文字（CoreText使用CoreGraphics渲染）和图片（ImageIO）渲染，由于GPU并不擅长做这些工作，不得不先由CPU来处理好以后，再把结果作为texture传给GPU。除此以外，有时候也会遇到GPU实在忙不过来的情况，而CPU相对空闲（GPU瓶颈），这时可以让CPU分担一部分工作，提高整体效率。





来自WWDC18 session 221，可以看到Core Text基于Core Graphics

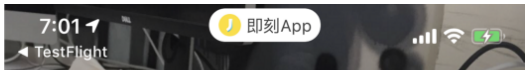
一个典型的例子是，我们经常会使用CoreGraphics给图片加上圆角（将图片中圆角以外的部分渲染成透明）。整个过程全部是由CPU完成的。这样一来既然我们已经得到了想要的效果，就不需要再另外给图片容器设置cornerRadius。另一个好处是，我们可以灵活地控制裁剪和缓存的时机，巧妙避开CPU和GPU最繁忙的时段，达到平滑性能波动的目的。
这里有几个需要注意的点：

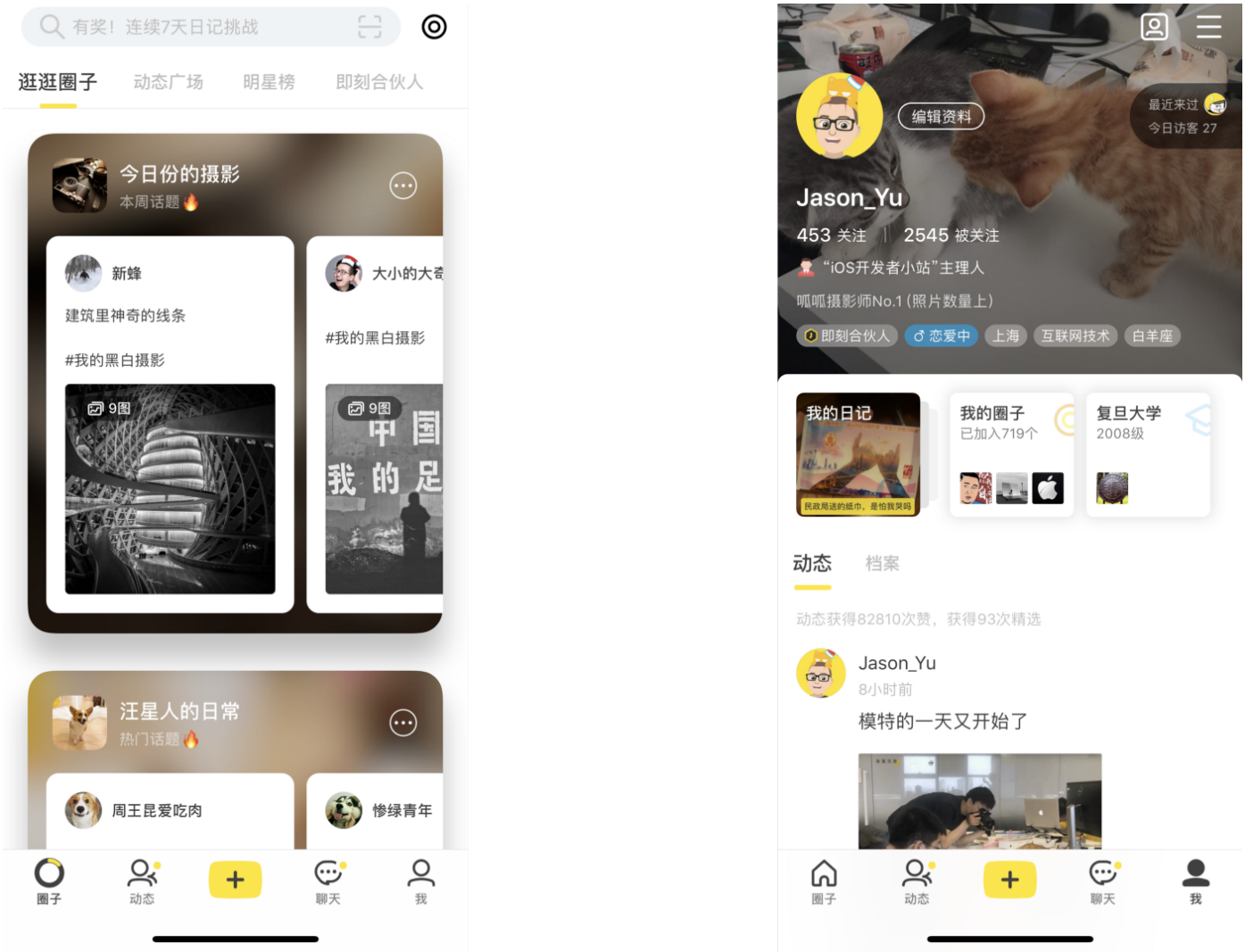
- 渲染不是CPU的强项，调用CoreGraphics会消耗其相当一部分计算时间，并且我们也不愿意因此阻塞用户操作，因此一般来说CPU渲染都在后台线程完成（这也是AsyncDisplayKit的主要思想），然后再回到主线程上，把渲染结果传回CoreAnimation。这样一来，多线程间数据同步会增加一定的复杂度
- 同样因为CPU渲染速度不够快，因此只适合渲染静态的元素，如文字、图片（想象一下没有硬件加速的视频解码，性能惨不忍睹）
- 作为渲染结果的bitmap数据量较大（形式上一般为解码后的UIImage），消耗内存较多，所以应该在使用完及时释放，并在需要的时候重新生成，否则很容易导致OOM
- 如果你选择使用CPU来做渲染，那么就没有理由再触发GPU的离屏渲染了，否则会同时存在两块内容相同的内存，而且CPU和GPU都会比较辛苦
- 一定要使用Instruments的不同工具来测试性能，而不是仅凭猜测来做决定

即刻的优化

由于在iOS10之后，系统的设计风格慢慢从扁平化转变成圆角卡片，即刻的设计风格也随之发生变化，加入了大量圆角与阴影效果，如果在处理上稍有不慎，就很容易触发离屏渲染。为此我们采取了以下一些措施：

- 即刻大量应用AsyncDisplayKit(Texture)作为主要渲染框架，对于文字和图片的异步渲染操作交由框架来处理。关于这方面可以看我之前的一些介绍
- 对于图片的圆角，统一采用“precomposite”的策略，也就是不经由容器来做剪切，而是预先使用CoreGraphics为图片裁剪圆角
- 对于视频的圆角，由于实时剪切非常消耗性能，我们会创建四个白色弧形的layer盖住四个角，从视觉上制造圆角的效果
- 对于view的圆形边框，如果没有backgroundColor，可以放心使用cornerRadius来做
- 对于所有的阴影，使用shadowPath来规避离屏渲染
- 对于特殊形状的view，使用layer mask并打开shouldRasterize来对渲染结果进行缓存
- 对于模糊效果，不采用系统提供的UIVisualEffect，而是另外实现模糊效果（CIGaussianBlur），并手动管理渲染结果





即刻客户端中有大量的圆角、阴影等效果

总结

- CPU渲染虽然也是“离屏”，但是通常提到的离屏渲染是发生在GPU
- 如果一个layer无法在一次遍历就完成绘制，那么就不得不触发离屏渲染
- 离屏渲染的开销主要在与frame buffer与离屏buffer之间的上下文切换。如果无法避免，也可以通过有效利用shouldRasterize，减少触发的次数
- CPU和GPU是相互扶持的关系。CPU渲染效率不高，但是较为通用灵活；GPU擅长并行计算，但也有捉襟见肘之时，此时CPU可以适当给与帮助

离屏渲染牵涉了很多Core Animation、GPU和图形学等等方面的知识，在实践中也非常考验一个工程师排查问题的基本功、经验和判断能力——如果在不恰当的时候打开了shouldRasterize，只会弄巧成拙。

从一个更广阔的视角看，离屏渲染也仅仅是渲染性能优化中的一部分，而能否保证UI性能过关，将会直接影响到用户日常的操作体验。渲染技术作为客户端工程师的关键技术能力之一，值得持续研究。

推荐资料

Andy Matuschak关于离屏渲染的解释
Objc.io: Moving Pixels onto the Screen
Mastering Offscreen Render
WWDC 2011 421 Core Animation Essentials

WWDC 2011 121 Understanding UIKit Rendering
WWDC 2014 419 Advanced Graphics and Animations for iOS Apps
WWDC 2010 135 Advanced Performance Optimization on iPhone OS Part 1
《Core Animation: Advanced Techniques》

iOS App Development Rendering Offscreen Rendering Performance Tuning iOS

165 claps

WRITTEN BY

Jason Yu

Follow

More From Medium

Related reads Also tagged Rendering Related reads

Facebook Wants to Cut Down on Misinformation. So

Adventures in the machine-learning land of drones &

WWDC 2018, first impressions

Why Isn't it Doing Anything About Infowars?

Was... Post
Jul 20, 201...

361

lidars

C... Digital
Jun 21 · 15...

53

Rubens
Jun 5,...

281