Android (/tags/#Android) Performance (/tags/#Performance) Android Performance Patterns (/tags/#Android Performance Patterns)

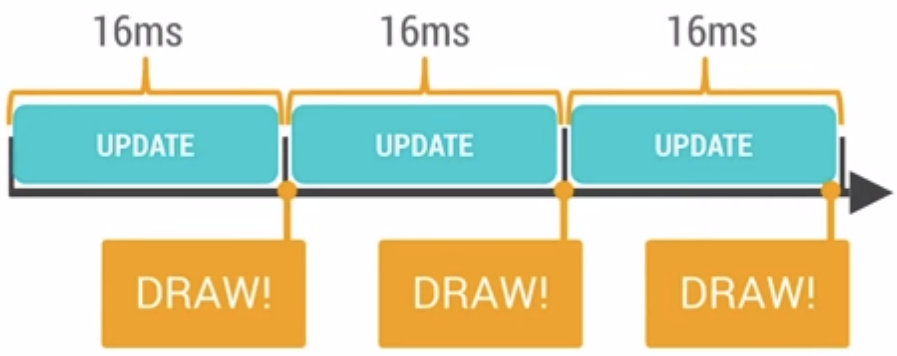
Android Performance Patterns——UI Performance

Android Performance Patterns系列学习思考和实践笔记

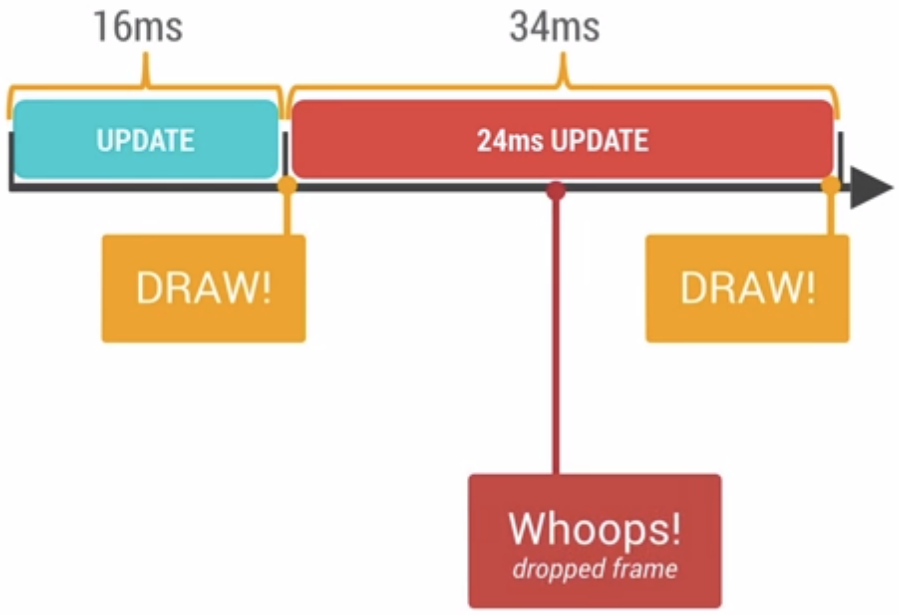
Posted by Cheson on March 8, 2017

0. Render Performance

本节重点为介绍用户能感知到的UI绘制时卡顿一类性能问题的直接来源。
Android系统每隔16ms发出VSYNC信号来触发UI绘制的动作，理想情况下，每次VSYNC信号到达时一帧的数据都已经准备好了，这样就能达到60fps的帧率，人眼看起来也就是非常流畅的效果。



但是如果某一帧画面的准备需要24ms，那么在第32ms的VSYNC信号到达时，将没有新的一帧可供渲染显示，画面则会保持在第一个16ms的那一帧，于是就出现了卡顿丢帧的现象。



此类出现的原因大多为layout过于复杂，界面层叠关系太多，动画多次绘制等。可以用HierarchyViewer来查看布局的复杂度；Show GPU Overdraw查看界面层叠程度；TraceView查看CPU情况。个人经验还可以由systrace来分析应用中每一帧的绘制时间。

1. Why 60fps

我们通常都会提到60fps与16ms，可是知道为何会是以程序是否达到60fps来作为App性能的衡量标准

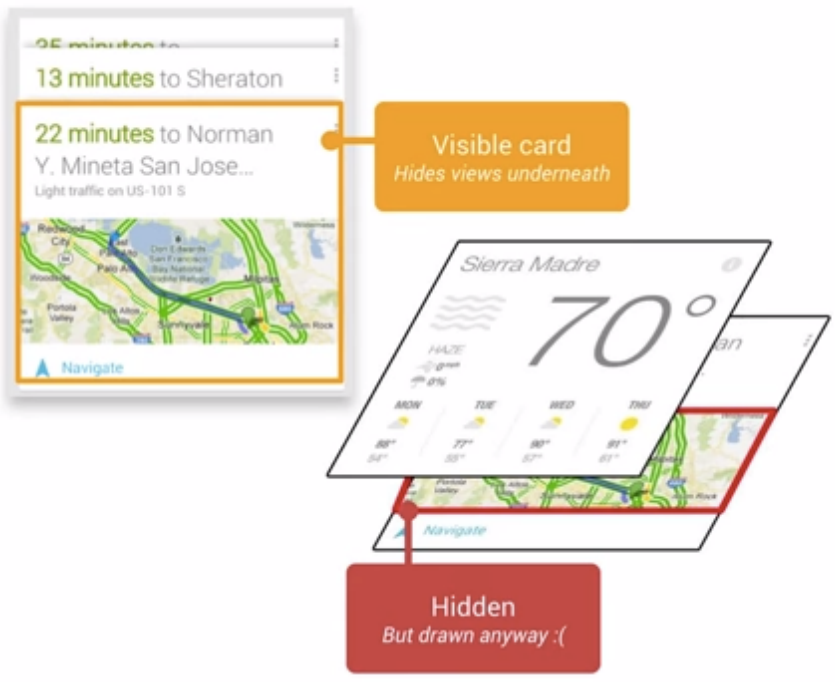
吗？这是因为人眼与大脑之间的协作无法感知超过60fps的画面更新。

12fps大概类似手动快速翻动书籍的帧率，这明显是可以感知到不够顺滑的。24fps使得人眼感知的是连续线性的运动，这其实是归功于运动模糊的效果。24fps是电影胶圈通常使用的帧率，因为这个帧率已经足够支撑大部分电影画面需要表达的内容，同时能够最大的减少费用支出。但是低于30fps是无法顺畅表现绚丽的画面内容的，此时就需要用到60fps来达到想要的效果，当然超过60fps是没有必要的。

开发app的性能目标就是保持60fps，这意味着每一帧你只有16ms=1000/60的时间来处理所有的任务。

2. Understanding Overdraw

这一节重点介绍了Overdraw（过度绘制），过度绘制是导致界面卡顿的一个来源，其根源在于过多层次的界面堆叠，例如下图

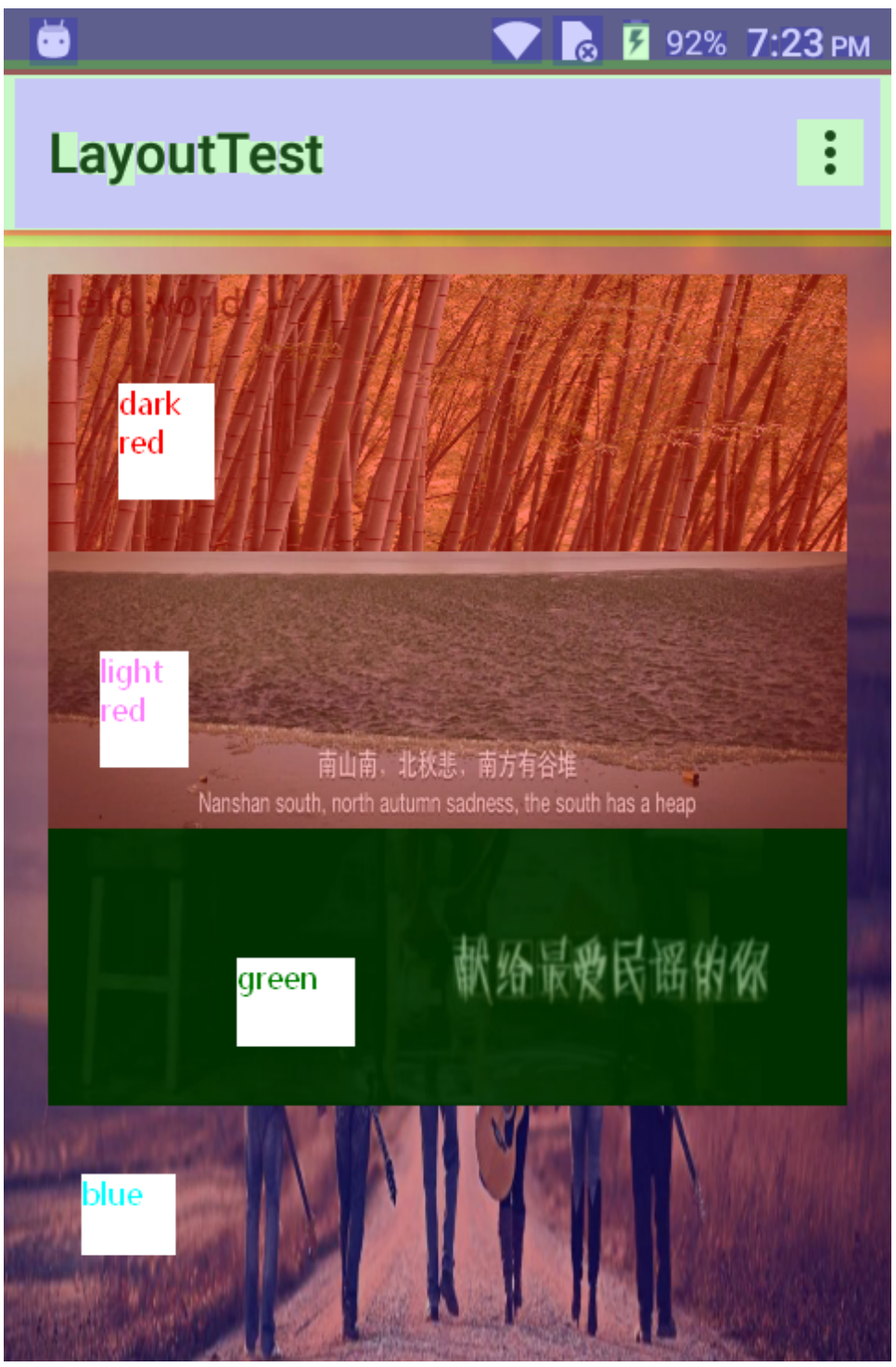


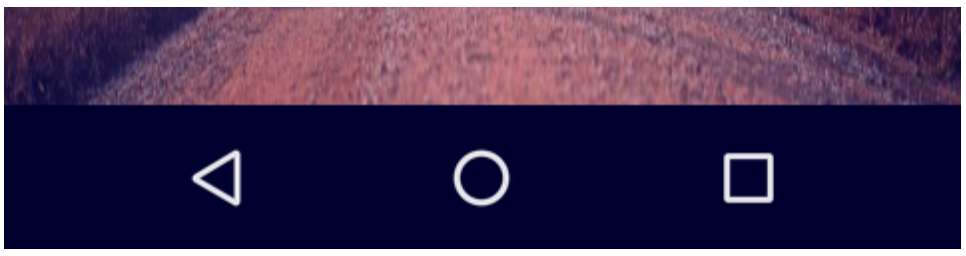
在多层次的UI结构中，当不可见的部分也需要被计算绘制时，那就是对CPU和GPU资源的浪费，反过来说就是对性能的拖累。此类问题可以用开发者模式中的Show GPU Overdraw来查看界面的覆盖层次

不同颜色的布局层次如上图中所示，优化的目标就是尽量减少红色的部分。自己编写了以下一段示例布局代码来进一步理解Overdraw

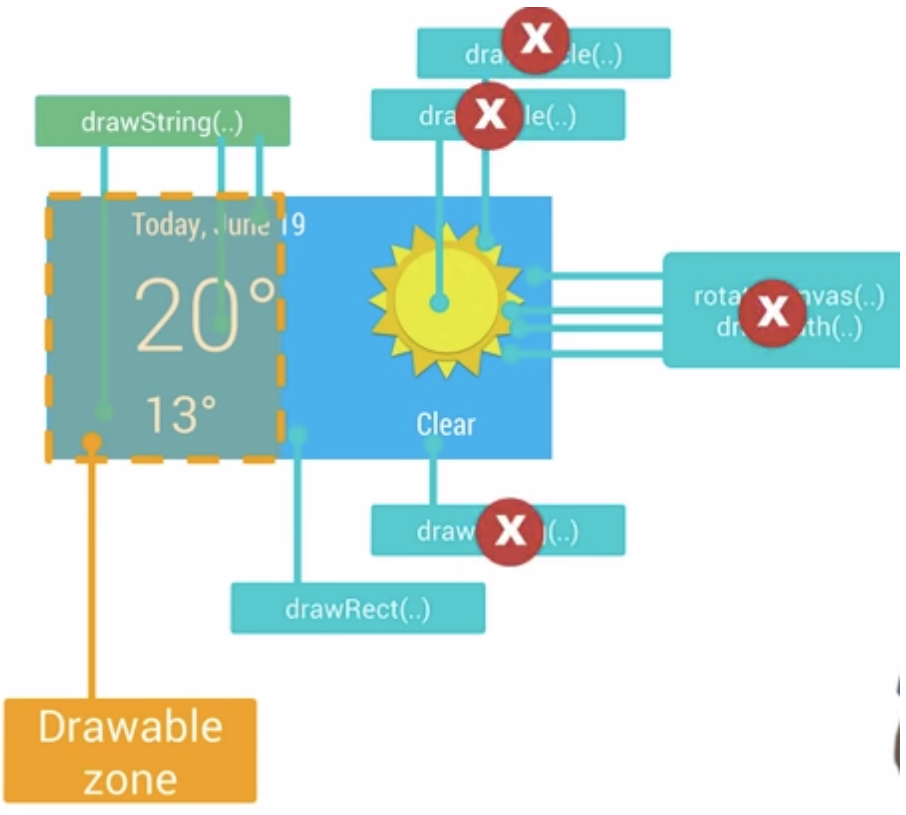
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=".MainActivity"
    android:background="@drawable/homebg">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="300dp"
        android:background="@drawable/bg1">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:background="@drawable/bg2">

            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="100dp"
                android:background="@drawable/bg3">

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:text="@string/hello_world"/>
            </LinearLayout>
        </LinearLayout>
    </LinearLayout>
</RelativeLayout>
```

从布局上看最多有四层背景图片需要绘制，在机器上呈现的效果如下，此类问题就可以通过去除不显示的区域的绘制来达到减少绘制层级的问题。





当遇到过于复杂的自定义布局(在onDraw方法中实现了重绘)，此时系统将无法检测到Over draw。这种情况下可以通过Canvas类提供的两个方法来做优化。Canvas.clipRect() (https://developer.android.com/reference/android/graphics/Canvas.html)：这个方法指定了一块矩形区域，在此区域内的View会被绘制，此区域外的绘制指令将不会被执行，部分在此区域内的内容也会被绘制。此方法可以用来帮助很好的绘制那些拥有多个重叠界面的自定义界面。



Canvas.quickreject() (https://developer.android.com/reference/android/graphics/Canvas.html)：这个方法用来判断指定的矩形区域是否没有和当前区域完全相交。

> *Return true if the specified rectangle, after being transformed by the current matrix, would lie completely outside of the current clip. Call this to check if an area you intend to draw into is clipped out (and therefore you can skip making the draw calls).*

以上两个方法暂未实际用于优化中，理解还未深刻，后续还会更新体验。

另有两个疑问还未寻求到解答，暂且记录在这里：1、布局的扁平化对over draw是否有改善？2、移除非必要的控件肯定是能减少布局的onMesure和onDraw时间，那么通常做的将控件设置为INVISIBLE或者GONE时，是否还会进行onMesure和onDraw呢？

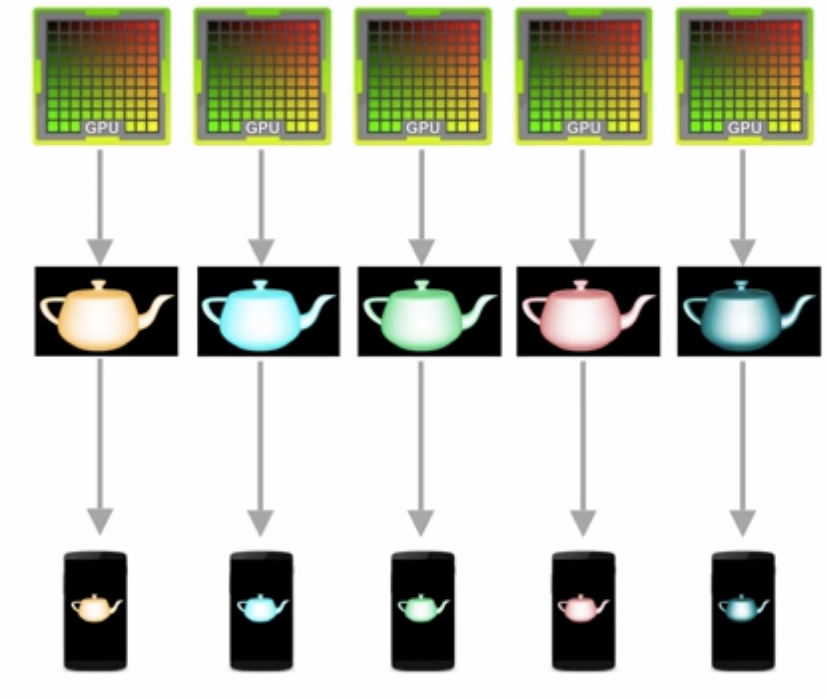
3. Understand VSYNC

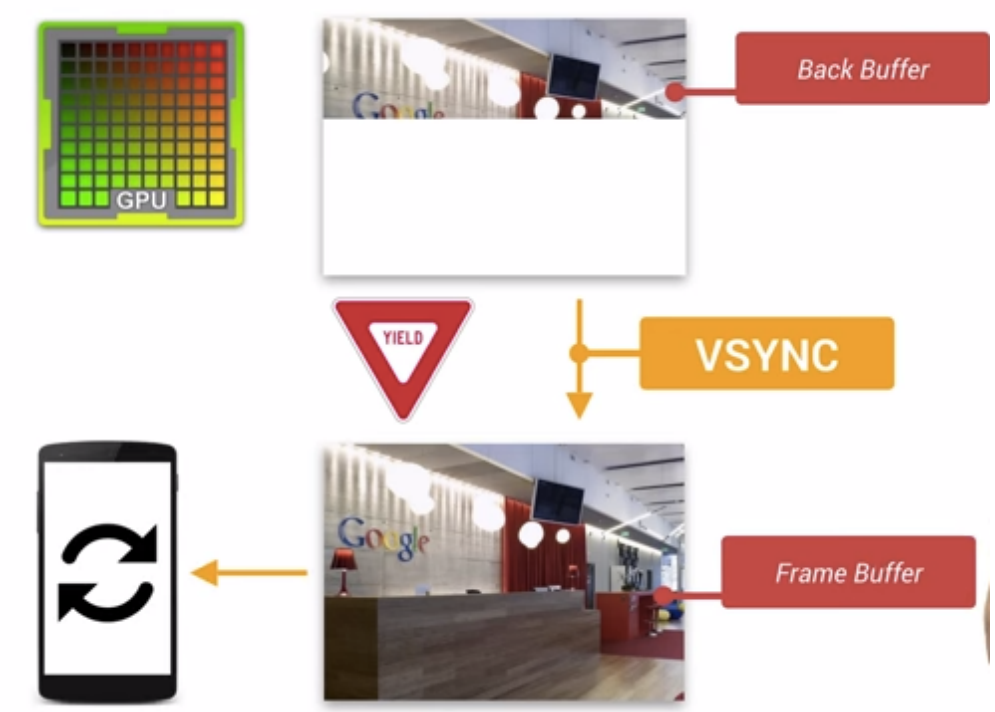
本节的是UI Performance中最难理解的核心，知识点在于理解GPU的帧率和硬件的刷新频率之间的关系以及双缓冲和三缓冲机制。

3.1 帧率和刷新频率

首先来看下GPU的帧率和硬件的刷新频率之间的关系，先区别下帧率和刷新频率这两个概念：帧率（Frame Rate），代表了GPU在一秒内绘制操作的帧数，例如30fps，60fps；刷新频率（Refresh Rate），代表了屏幕在一秒内刷新频率的次数，此参数在TP Firmware中固定，例如60HZ。刷新频率是一个固定的值，而VSYNC信号是和刷新频率同步的，而帧率是一个动态的值，和当前GPU的负载以及处理

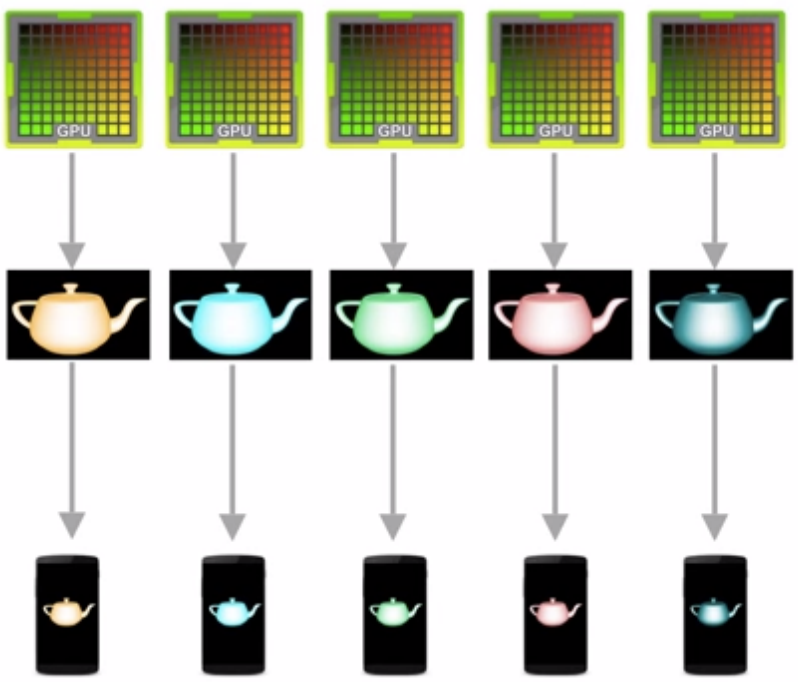
的数据内容相关。在理想的情况下，例如刷新频率为60HZ，帧率为60fps的情况下，两者是非常和谐的同步协作的。





而一般更多情况下是帧率和刷新频率不同步，当帧率高于刷新频率时在胡凯的翻译资料中是这样描述的

> *如果发生帧率与刷新频率不一致的情况，就会容易出现Tearing的现象(画面上下两部分显示内容发生断裂，来自不同的两帧数据发生重叠)*



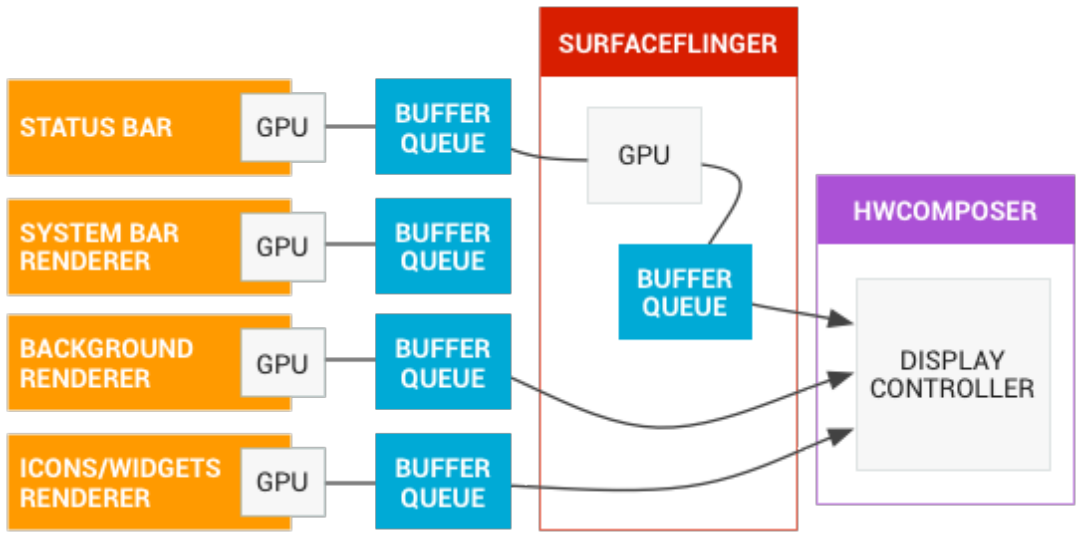
而后面又出现了这样的描述

> *通常来说，帧率超过刷新频率只是一种理想的状况，在超过60fps的情况下，GPU所产生的帧数据会因为等待VSYNC的刷新信息而被Hold住，这样能够保持每次刷新都有实际的新的数据可以显示*

我的理解这两段描述是存在矛盾的，在**GPU**中渲染时，正常情况下使用双缓冲的机制，姑且记做**A**和**B**，两个**buffer**是轮流当做显示和缓冲来使用的，而一帧画面的显示是在收到**VSYNC**信号之后才会发生，那么被**post**到**frame buffer**中的画面是来自**A**或者**B**的，什么情况下会出现两帧重叠而产生的断裂呢？

帧率超过**60fps**是一种理想的状态，这个是可以理解的。**GPU**中的一帧绘制完之后会等待**VSYNC**的到来才会做**post**的动作，那么如果**GPU**缓冲中的数据可以被**hold**住的话，对于界面刷新来说无疑是一个非常好的效果。然后基于这些资料会思考的一个问题，**GPU**是如何**hold**住已经画完的帧呢？难道在**VSYNC**信号到来之前画完一帧之后就不再绘制了吗？这样的画岂不是浪费了**GPU**的性能，而且从帧率的概念上来理解，帧率是不可能超过刷新频率的。

这个链接http://source.android.com/devices/graphics/index.html (http://source.android.com/devices/graphics/index.html)中描述了Android系统中绘制的原理，其中有部分解释了图像绘制时数据流通道的原理



The objects on the left are renderers producing graphics buffers, such as the home screen, status bar, and system UI. SurfaceFlinger is the compositor and Hardware Composer is the composer.

GPU渲染完成的帧数据会有Buffer Queue来做保存，这个就解释了GPU帧率超过刷新频率时，超前绘制的帧数据是如何hold住的。然而画面断层的现象此处还是无法解释，在网上搜罗到了这篇资料来解读断层问题 Android 关于display的几个问题 (http://blog.csdn.net/u014717231/article/details/53127933)

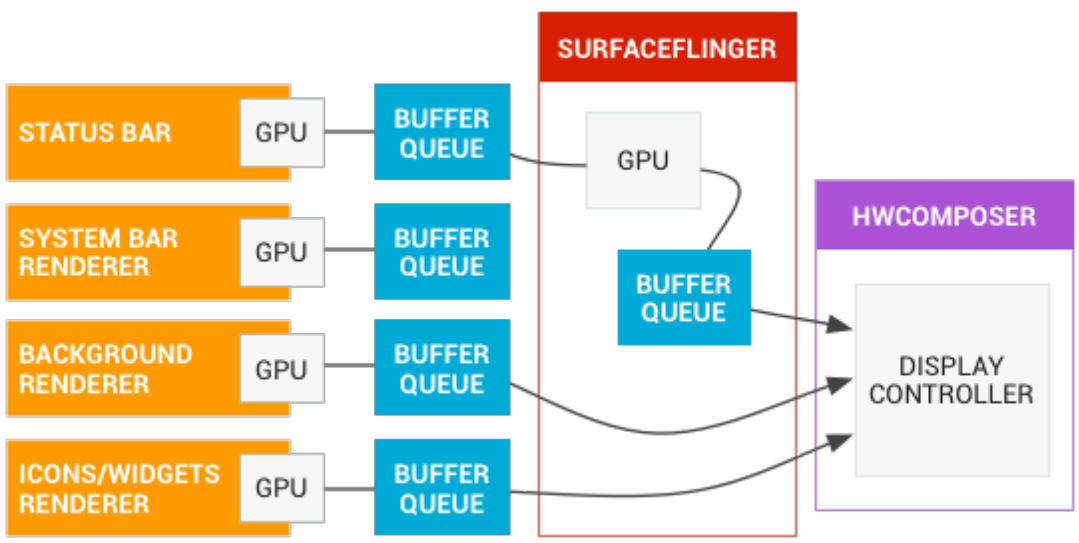
> *切屏又叫Tearing，即“断层”“分屏”“撕裂”现象 Tearing产生的原因：BB通过LCD IF将数据刷到DRIVER IC的GRAM的时间T0与DIRVER IC将GRAM数据刷到LCD的玻璃上的的时间T1不同步，导致Baseband没写完一帧Driver IC就更新了。 针对有FMARK的屏，硬件将FMARK脚接到BB LCDC TE脚，软件上开启TE功能。原理是，LCM GRAM中的显示信息被刷新到LCD PANEL后，LCD Driver IC会发出一个同步sync信号告诉BB可以更新 Frame Buffer内容到LCM Gram了。保证了上述T0 、T1时间上不会重叠 针对没有FMARK的屏，只能尽量调整LCD IF送数据的时序及LCD Driver IC刷新频率，使T0、T1达到同步*

此处解释了一种LCD IF送数据的时序和IC刷新的时序不同步导致的切屏现象。我们在项目中遇到此类问题时会先区分是FrameBuffer中的数据有问题还是IC的时序问题，那么有一个简单的方式可以来做这一判定。当出现切屏现象时，保留现场，用**DDMS**工具来截屏，而截屏是直接从**FrameBuffer**中取得数据，如果截屏界面完整则可以由驱动来做进一步分析**IC**，如果截屏也有切屏现象，则很明显是上层准备的数据就有问题了。

另一种方法则需要**root**权限来操作，然后用工具查看：

```
adb shell
cat /dev/graphics/fb0 > /data/fbxx.bin
exit
adb pull /data/fbxx.bin
```

当帧率超过刷新频率的情况讨论了很多，下面在来看下当帧率低于刷新频率时会出现什么样的情况。其实在绝大多数使用情况下界面的绘制都是处于这样的情况下，需要明确一个误区，帧率低于刷新频率时并非是一种坏的情况，很多时候界面保持不动或者没有必要那么快的更新，这些时候都是正常的。而当帧率从**60**帧突然降低到**60**帧以下时，就会发生界面卡顿之类的糟糕体验。



3.2 缓冲机制

本节中的第二个难点就是理解双缓冲和三缓冲机制，在胡凯翻译的原文中没有对此机制做详述，提供了两个额外的知识链接http://source.android.com/devices/graphics/index.html (http://source.android.com/devices/graphics/index.html)和http://article.yeeyan.org/view/37503/304664 (http://article.yeeyan.org/view/37503/304664)。从第二个链接的译文中可以初窥缓冲机制的简单设计：双缓冲已经是对GPU性能的一种提升，唯一不足的情况在于当VSYNC信号到来时，缓冲A的第0帧正在显示而缓冲B的第1帧超过了16ms还未准备好，此时只能继续保持显示缓冲A的第0帧。而且在第2帧数据到来时没有缓冲区可以给其绘制，所以又要等到B缓冲渲染结束才能做下一步，也就是所谓的一步慢步步慢。在Jelly Bean中对此做了优化，加入了第三个缓冲区C，当出现以上问题时，会创建出第三个缓冲区C，用来渲染第3帧，在这种情况下虽然第2帧依旧无法显示，但是当32秒的VSYNC到来时，可以将缓冲区C中的帧送到屏幕显示，这样随会跳过一帧，但是保证了后续的显示是平滑的，充分利用了GPU的性能。

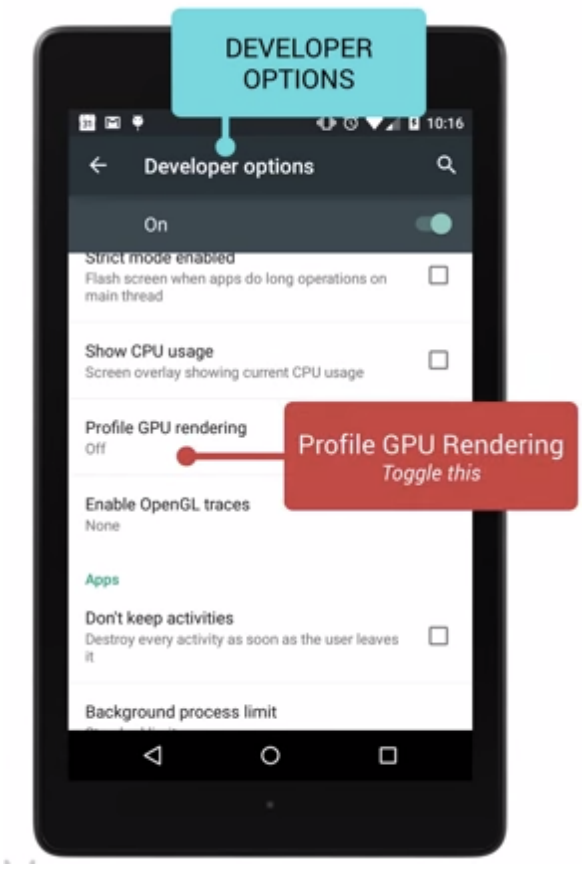
按照这个模式进一步思考，如果在系统性能更差的情况下，三个缓冲都不够用，那么是否可以考虑创建更多的缓冲呢？甚至是预留一个缓冲队列来动态分配呢？

4. Tool:Profile GPU Rendering

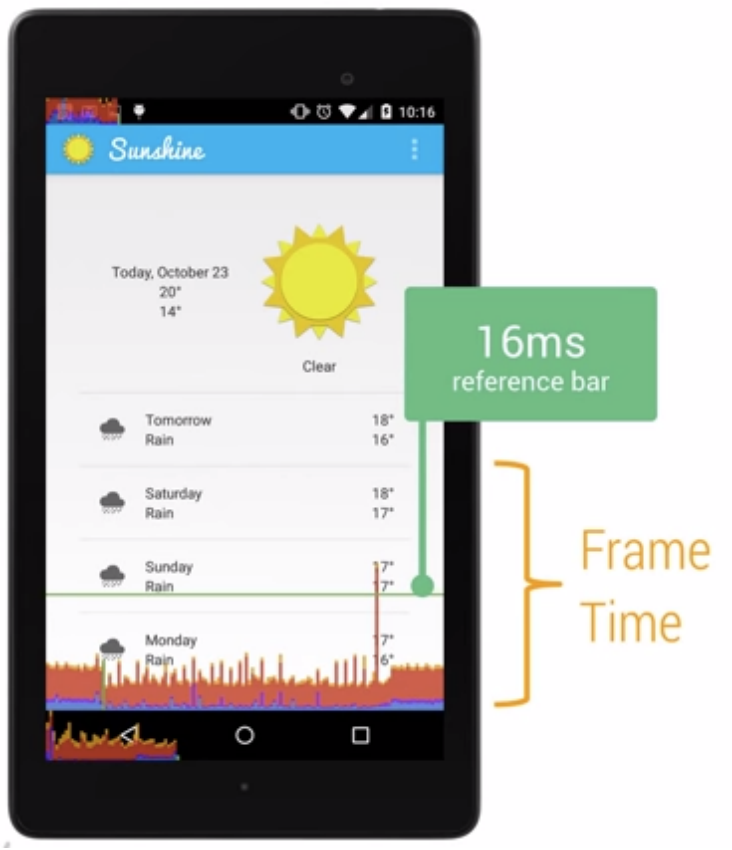
4.1 基础

上一节中介绍了界面渲染性能问题的原理，既然刷新频率是固定的，那么变数就在于GPU的帧率了。那么如何去查看和分析某个界面的显示帧率是否存在问题呢？本节带来了一个分析工具的介绍，位于开发者模式下的Profile GPU Rendering。先罗列下google提供的基础资料：

如何开启工具**:** 找到Profile GPU Rendering，选中on screen as bars，这样就可以在屏幕上以柱状图显示每一帧绘制的耗时。后面会再介绍另一种使用方法“in adb shell dumpsys gfxinfo”



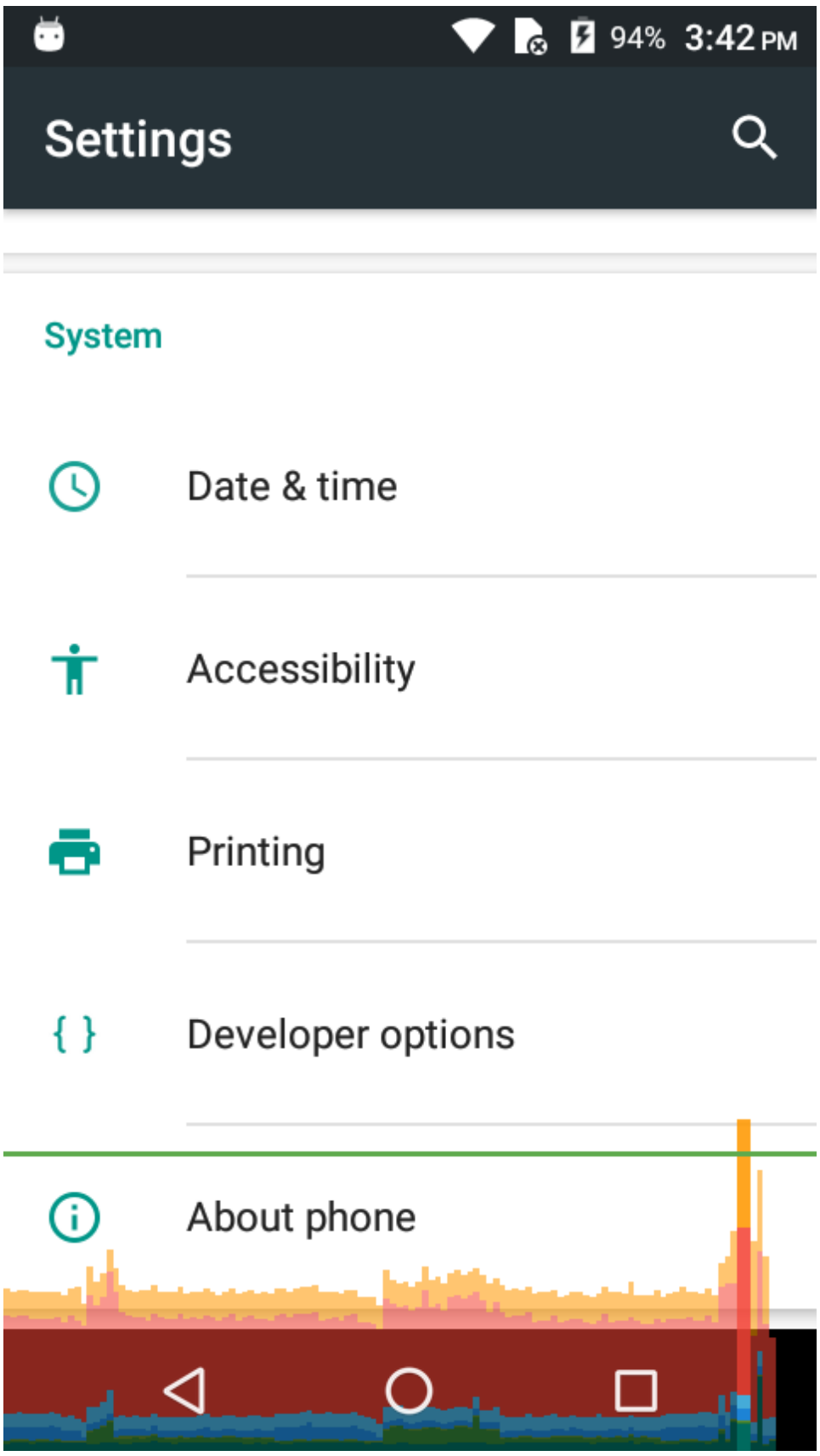
16ms基准线: 柱状图越高表示渲染使用的时间越长，google很友好的做了一条基准线来让我们便于判断渲染时间的大致水平，中间的绿线代表了绘制时间为16ms，为何会选择16ms，这一点在第1节中解释过原理，我们需要确保每一帧花费的总时间都低于这条横线，这样才能够避免出现卡顿的问题。



三种颜色**:** google介绍的Profile GPU Rendering的柱状图每条线都有三种颜色组成：蓝色——测量绘制Display List的时间；红色——OPEN GL渲染Display List的时间；黄色——CPU等待GPU的时间。

以上为google视频中介绍的信息，拥有这些信息也只是大致了解了Profile GPU Rendering工具的使用入门而已，离真正能够掌握该工具用来分析渲染性能还有很大的距离。其中有几点问题：**1、Display List**

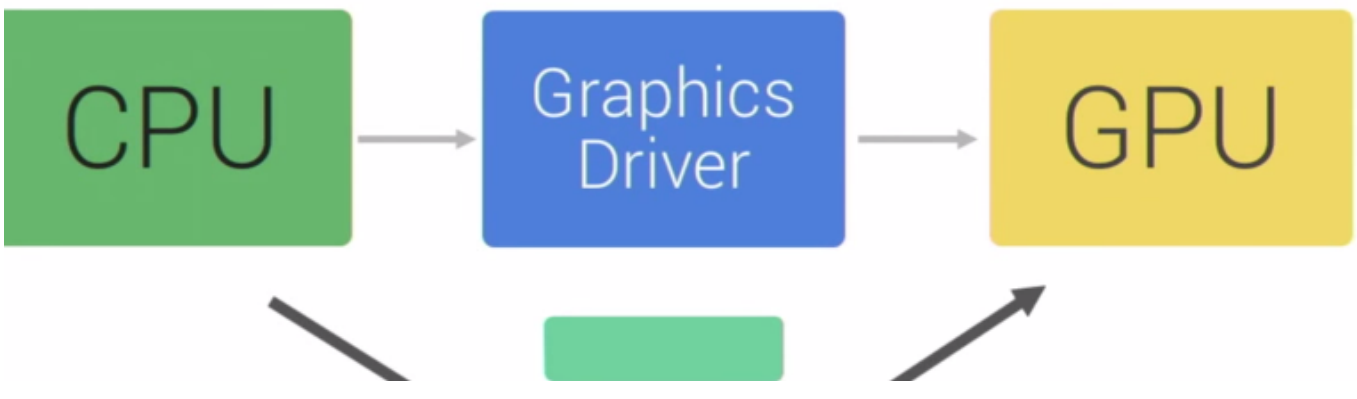
的含义；**2**、三种颜色代表的动作如何理解；**3**、当渲染时间超过**16ms**时就意味着卡顿了吗；**4**、在**Android6.0**上呈现的柱状图不止三种颜色

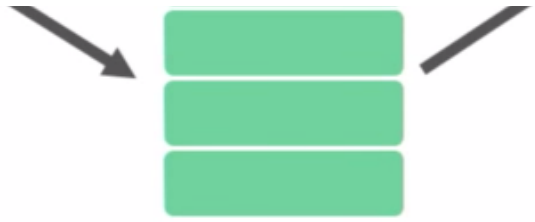


又搜罗了几篇介绍该工具的资料，结合前面的背景知识，加深了对此的理解。附上两篇比较有心得的相关资料玄学曲线并不玄 教你如何看懂GPU呈现 (http://tech.hexun.com/2016-01-27/182036598.html)和Android性能优化系列——Profile GPU Rendering (http://blog.csdn.net/xu_fu/article/details/45008779)

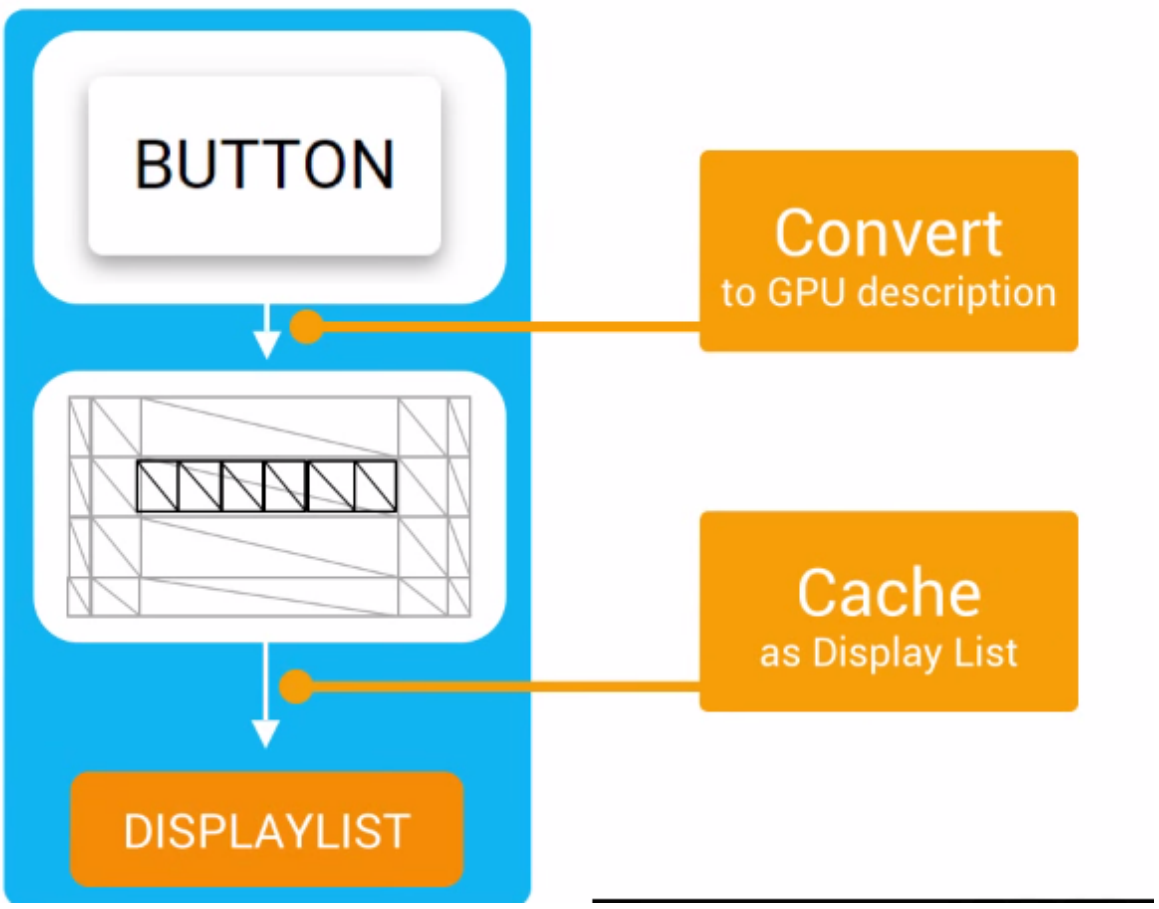
4.2 Display List

先来看第一个问题**Display List**是什么： CPU和GPU是没有直接通信的API的，中间需要一个叫做Graphics Driver的驱动来连接，在Graphics Driver中维护了一个队列，CPU将display list放到这个队列中，而GPU从中取出display list进行绘制。





这里的display list虽没有明确的给出它的定义，但是从绘制的原理可以理解它大致的含义就是cpu通过测量和绘制之后产生的显示元素的队列，例如一个activity中的按钮，文本，图片等元素绘制命令的集合。其原因在于在视图被渲染之前需要先转换成GPU能接受的格式，简单的可能是几个绘制命令，复杂的也可以是嵌入在canvas里的自定义路径。



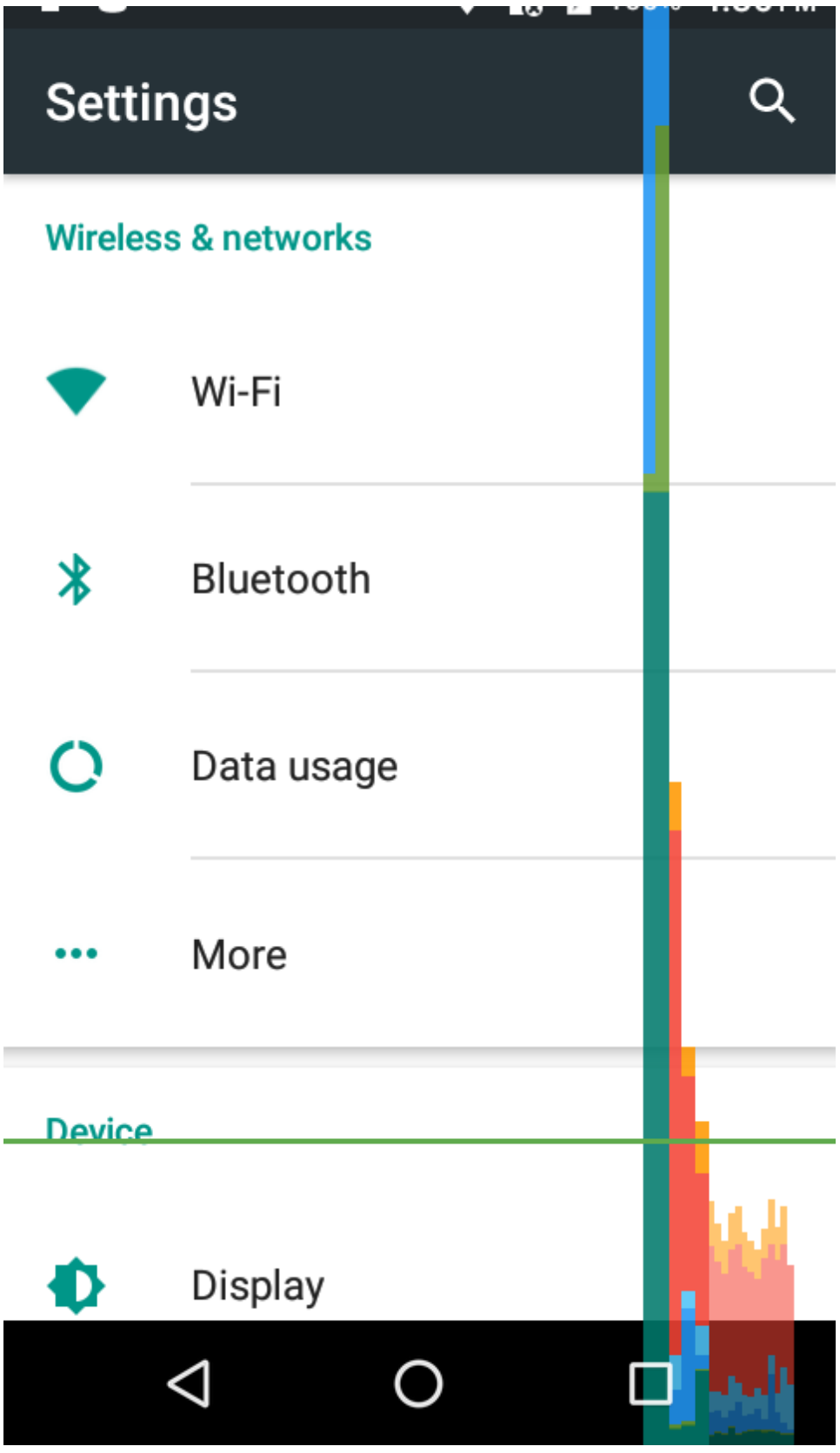
4.3 三种颜色

理解了display list之后就可以接着理解第二个问题三种颜色的动作如何理解。

蓝色（Draw）——测量绘制Display List的时间

> *The blue section of that bar represents draw time or rather how long it took to create and update your display lists in Java. Remember that before a view can actually be rendered, it has to first be transformed into a GPU-friendly format. On the simple side of this, it could just be a few Draw commands. But on the complex end, we could be tessellating a custom path coming from your canvas object. Once done ,the results are then cached as a display list object by the system. This blue bar is recording how much time it takes to complete these two steps for all the views that need to be updated on the screen this frame. When you see this bar shoot high, it could mean that a bunch of views suddenly became invalidated, or it may be a few custom views who might have some extremely complex logic in their onDraw funtion.*

代表了在视图创建或者更新时，发生在java层的绘制并在绘制完成后以display lists的形式缓存到cache中的耗时。当看到蓝条很高时，有可能是一批视图突然失效了，或者有一些自定义的视图在**onDraw**方法中的逻辑非常复杂引起的。第一点原因在切换界面时经常能看到，例如从launcher进入到Settings，虽然Settings的界面并不是很复杂，但是由于大批的视图失效需要重绘。第二点可以作为一个排查问题的方向，调查自定义**View**的**onDraw**方法中是否有耗时操作。一般看来，当蓝色部分普遍小于16ms时，界面刷新一般是不会有问题了。当然最佳的情况当然是整体都小于16ms。



红色（Execute）——OPEN GL渲染Display List的时间

> *The red section of the bar represents execute time.This is the time spent by Android's 2D renderer to execute display list. See, in order to draw to the screen, Android needs to draw your display list information by interacting with the OpenGL ES API which effectively passes along data to the GPU, which then, ultimately, ends up putting pixels on the screen. Remember that for more complex views like a custom view, the more complex the commands needed for OpenGL to draw it. And when you see this red bar shoot high, these complex views are a likely culprit. It's also worth noting that large spikes in this bar can come from re-submitting a load of views to be redrawn again. These views haven't necessarily been invalidated. But if something happens, like a view rotates, then we need to Go back and clean up areas of the screen that might be affected by that redrawing the views underneath it.*

红色部分代表2D渲染执行display list的时间。为了呈现display lsit信息，OpenGL需要绘制数据并传递给GPU进行渲染，最后呈现在屏幕上。当界面越复杂，OpenGL需要执行的绘制命令就越多。当看到红条很高时，比较复杂的界面都可能是罪魁祸首，还需要留意的是峰值，峰值比较高时也可能是有些view被多次重绘了，其实可能并没有重绘的必要。界面的复杂度我们可以通过sdk下的hierarchyviewer工具来查看。　　实际使用中，比如我们平时刷淘宝App时遇到出现多张缩略图需要加载时，那么红色会突然跳很高，但是此时你的页面滑动其实是流畅的，虽然等了零点几秒图片才加载出来，但其实这可能并不意味着你卡住了。

黄色（Process）——CPU等待GPU的时间

> *The oragin section of the bar represents the process time. Or rather, this is where the CPU tells the GPU that it's done rendering a frame. This action is a blocking call, and as such the CPU will sit around and wait for the GPU to acknowledge that it's received the command. If this bar is getting large, then it means that you're doing a lot of work on the GPU resulting from many complex views that require a lot of OpenGL rendering commands to be processed.*

为什么CPU需要等待GPU？在3.2中的双缓冲和三缓冲机制中提到了目前GPU最多可使用的缓冲数为三个，当CPU渲染完一帧时需要通知GPU我已经绘制完了需要GPU腾出缓冲区来处理这一帧画面，而这个消息是个阻塞的消息，当收到GPU的答复之前CPU就不做新的绘制动作了。而收不到GPU答复的原因就在于GPU过于繁忙了，CPU的绘制动作领先了GPU。这个也验证了我在前面提到的想法，是否可以增多缓冲区来解决此类问题呢。在实际测试中，操作一个应用时，当看到偶尔出现的黄色偏高，那么可能就会发生偶尔丢帧的现象。

4.4 当渲染时间超过16ms时就意味着卡顿了吗

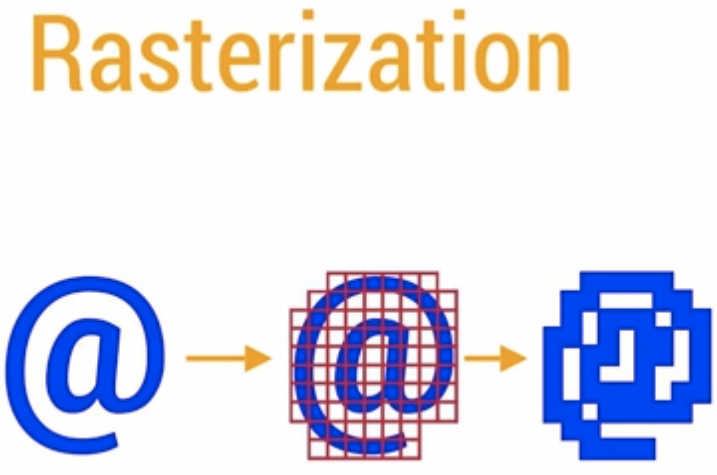
通过上面对三种颜色线条的深度理解，这个问题应该不难回答了。整体时间小于16ms时界面是平滑显示的，蓝色部分普遍小于16ms的情况下应该也是不错的。当红色部分很高的时候也就意味这界面比较复杂，但是也要看这个界面的设计，例如可能只是某个图片的渲染时间太长了导致的，而图片可以已模糊的形式先呈现，那么就不会出现卡顿。当然也有可能是真的出现卡顿的，就要去调查界面的复杂度是否可以精简。当黄色部分很高时，通常就是GPU过于繁忙了，这种情况下就很可能出现丢帧的情况。

4.5 不止三种颜色

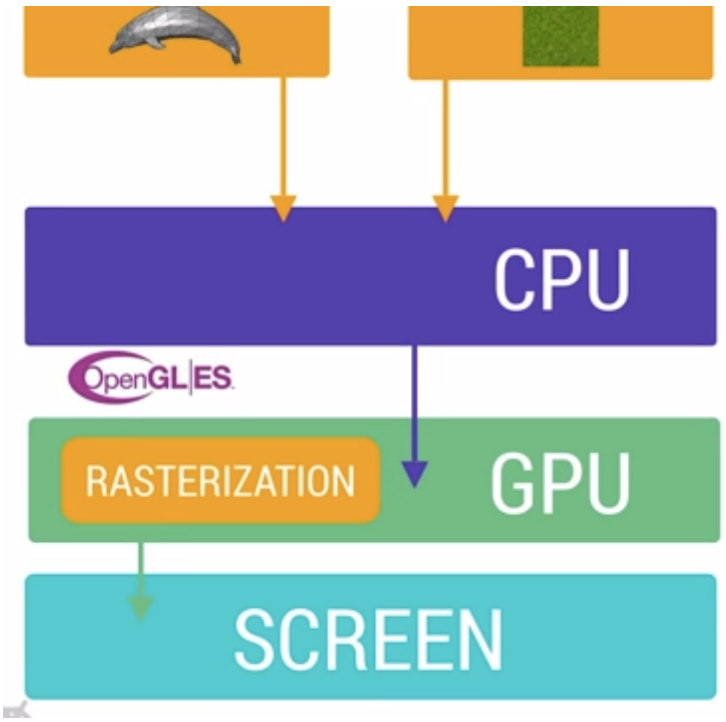
在Android6.0上的截图看到了不止三种颜色，还有绿色，浅蓝，粉色。这些颜色的含义还暂未找到资料。

5. Android, UI and the GPU

这一节中原文中资料简单介绍了Android界面绘制的原理以及Android是如何利用GPU进行渲染的。基本的界面元素例如Button，Image，Text等都是通过Resterization栅格化操作拆分到屏幕上的每个像素点上去的。

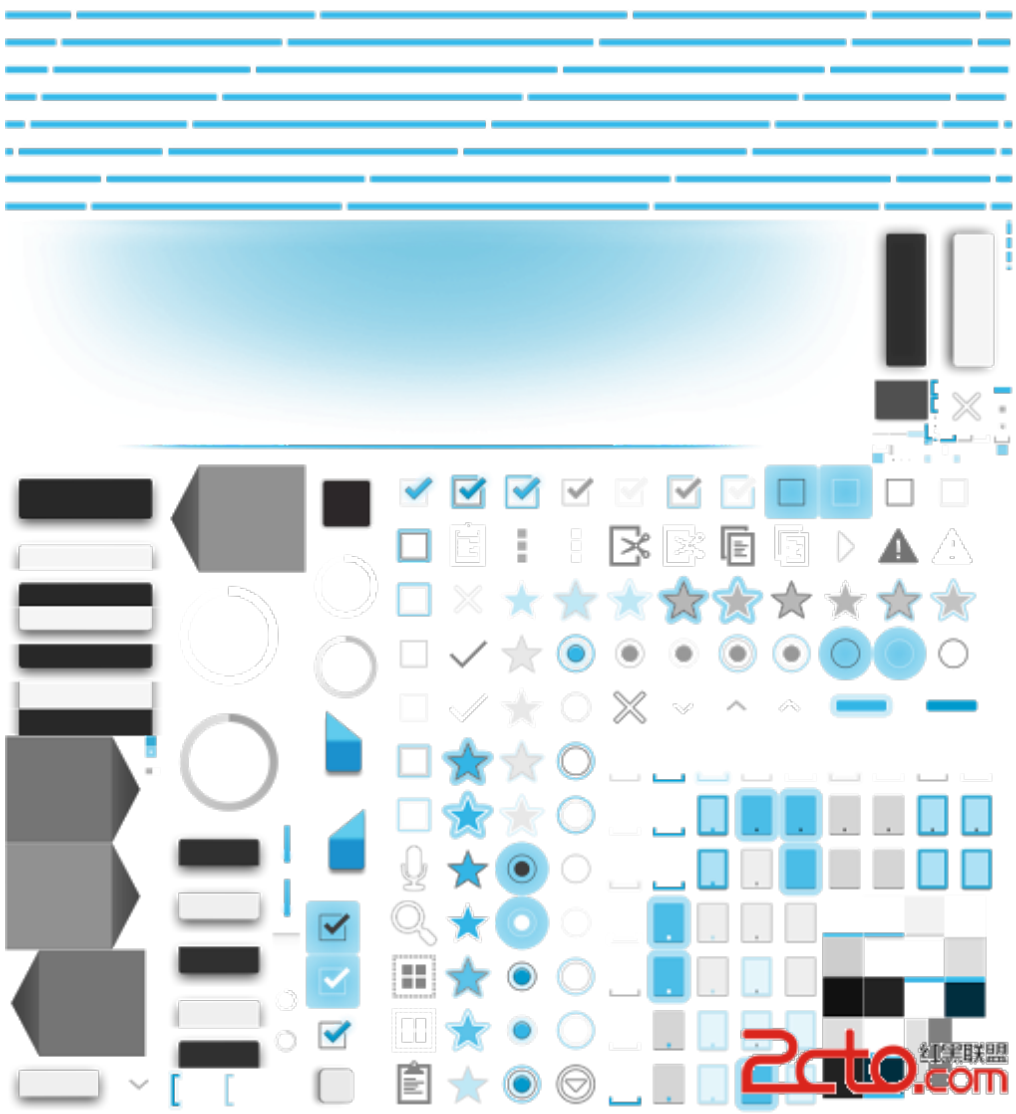


而栅格化操作是非常耗时的一个动作，所以一个频繁出现的概念“硬件加速”指的就是引入GPU来做这个栅格化的操作，它的性能要比CPU处理更优。CPU负责把UI组件计算成Polygons，Texture纹理，然后交给GPU进行栅格化渲染。



而对GPU来说有个很重要的指标就是显存(在配PC时会关注的比较多吧)，显存和GPU的性能联系在哪儿呢？当出现视图更新时，CPU会重新计算需要渲染的纹理，而从CPU将数据转移到GPU中是非常麻烦的，但是如果纹理可以直接从GPU的memory中读取出来，那么效率就高了很多。所幸的是OpenGL ES可以将纹理放在GPU的memory中，当下次需要时直接读取出来。所以显存和绘制性能的关系也就很明朗了。

在Android里面那些由主题所提供的资源，例如Bitmaps，Drawables都是一起打包到统一的Texture纹理当中，然后再传递到GPU里面例如以下创建的纹理集。而文字的栅格化会更加复杂，需要使用skia字库或者是第三方字库来做栅格化



关于硬件加速的的一些简单资料可以参考：Android HWUI硬件加速模块浅析 (http://www.2cto.com/kf/201507/425856.html)和android硬件加速总结 (http://blog.csdn.net/xu_fu/article/details/48208795)

6. Update & Performance

Android界面更新时会导致View的重绘，这个会导致如下的流程，CPU计算重新生成DisplayList，交给GPU执行渲染命令。对于一个View的整体来说，如果在渲染完之后后续操作中只是其位置发生了变化，则GPU只需要重新执行一次渲染命令即可。但是如果View中的控件发生变化，这个变化可以是位置或者尺寸

等，例如一个Button的大小发生了改变，那么需要重新计算这个Button在View中的位置，和其他子View的相对位置，其他子View的位置等信息。那么就需要重新生成DisplayList，然后GPU再重新渲染，所以当布局复杂时，频繁的更新也可能会导致性能问题。

在开发只模式下，有一个“Show GPU view updates”选项，开启之后在GPU渲染时会显示渲染的区域(也就是说用到硬件加速的应用)，此工具被网友称为鸡肋工具，自己使用中也还未发现具体有何作用。附上一篇介绍代码实现的文章Show GPU View Update实现原理 (http://www.wtoutiao.com/p/1bfiv8u.html)。

PREVIOUS
一场ANDROID PERFORMANCE的追根溯源之旅
(/2017/03/08/ANDROID_PER_PATTERNS_OVERVIEW/)

NEXT
ANDROID PERFORMANCE PATTERNS——MEMORY PERFORMANCE **(/2017/03/15/ANDROID_PERF_PATTERNS_MEMORY/)**

FEATURED TAGS (/tags/)

前端 (/tags/#前端) Android (/tags/#Android) frameworks (/tags/#frameworks) AlarmManager (/tags/#AlarmManager) Performance (/tags/#Performance) systrace (/tags/#systrace) PowerManager (/tags/#PowerManager) Wakelock (/tags/#Wakelock) Guitar (/tags/#Guitar) 民谣 (/tags/#民谣) 赵雷 (/tags/#赵雷) Doze (/tags/#Doze) Android Performance Patterns (/tags/#Android Performance Patterns)

FRIENDS

待遇见志同道合的你 (https://github.com)　小明 (http://www.betterming.cn)

(https://twitter.com/chendongqi)
(https://www.zhihu.com/people/chendongqi)
(http://weibo.com/chendongqi) (https://www.facebook.com/chendongqi)
(https://github.com/chendongqi)
(https://www.linkedin.com/in/firstname-lastname-idxxxx)