知乎 首发于 THINK.DESIGN.CODE

TextureView 的血与泪（完整修订版）

易旭昕
THINK.DESIGN.CODE

写作费时，敬请点赞，关注，收藏三连。

越来越多的应用需要使用自己的绘制引擎进行复杂内容的绘制，比如需要使用 GL 绘制 3D 的内容，或者绘制复杂的文档，图表时不希望阻塞 UI 线程，或者部分内容是通过类似 Flutter 这样的第三方 UI Toolkit 进行绘制。通常这部分内容会通过 SurfaceView 或者 TextureView 呈现在 UI 界面上。

一般来说 SurfaceView 能够提供更好的性能，但是因为 SurfaceView 本身的输出不是通过 Android 的 UI Renderer（HWUI），而是直接走系统的窗口合成器 SurfaceFlinger，所以无法实现对普通 View 的完全兼容。包括不支持 transform 动画，不支持半透明混合，移动，大小改变，隐藏/显示等时机会出现各种瑕疵等等，总的来说 SurfaceView 只适用于有限的场景。

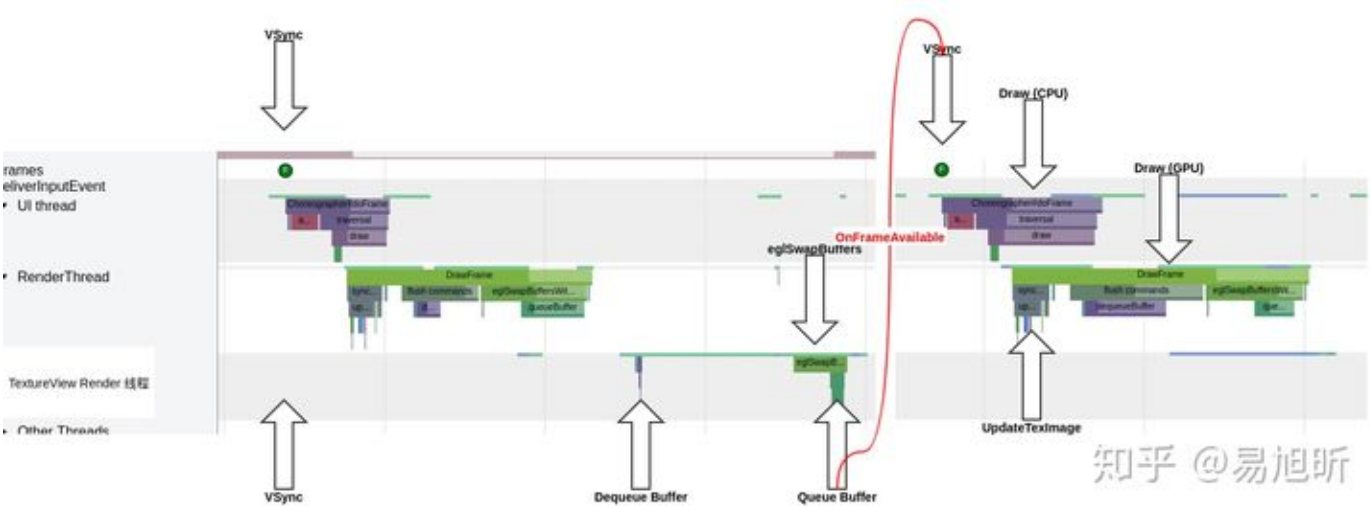
TextureView 正是为了解决 SurfaceView 这些的问题而诞生，在使用上它基本可以无缝替换 SurfaceView，并且因为 TextureView 跟普通 View 一样是通过 UI Renderer 绘制到当前 Activity 的窗口上，所以它跟普通 View 基本上是完全兼容的，不存在 SurfaceView 的种种问题。

但同时正是因为 TextureView 需要通过 UI Renderer 输出，也导致了新的问题的出现。除了性能比较 SurfaceView 会有明显下降外（低端机，高 GPU 负荷场景可能存在 15% 左右的帧率下降），另外因为需要在三个线程之间进行写读同步（包括 CPU 和 GPU 的同步），当同步失调的时候，比较容易出现掉帧或者吞帧导致的卡顿和抖动现象。

TextureView 绘制和输出

知乎 首发于 THINK.DESIGN.CODE

1. 我们自己创建的用于绘制 TextureView 的线程（当然实际上是绘制 TextureView 创建的 SurfaceTexture，通过 Surface 接口），在这里我们称之为 TextureView Render 线程（实际的名字取决于应用的绘制引擎）；
2. 系统创建的用于操作 View 的 UI 线程，也是应用的主线程；
3. 系统创建的用于绘制所有 View 的内容到当前 Activity 窗口的 Android Render 线程；



一般情况下，TextureView Render 和 UI 线程都是由 VSync 信号驱动的（Choreographer 的回调），而 Android Render 线程是由 UI 线程驱动的。

TextureView 本质上是 SurfaceTexture 的 View 封装，而 SurfaceTexture 本质上是一个 Buffer Queue。当我们使用 GL 绘制 SurfaceTexture 时（SurfaceTexture 包装成 Surface 作为当前上下文的 Window Surface）：

1. 新的一帧的第一个 GL Draw Call 会触发一个 Dequeue Buffer 的操作；
2. 后续的 GL Draw Calls 都绘制到这个 Buffer 上；
3. 调用 eglSwapBuffers 会触发一个 Queue Buffer 的操作；
4. Queue Buffer 会导致 SurfaceTexture 发送一个 OnFrameAvailable 的消息回 UI 线程；

我们再来看 TextureView 输出的流程：

```
private final SurfaceTexture.OnFrameAvailableListener mUpdateListener =
        new SurfaceTexture.OnFrameAvailableListener() {
    @Override
    public void onFrameAvailable(SurfaceTexture surfaceTexture) {
        updateLayer();
        invalidate();
```

```
private void updateLayer() {
    synchronized (mLock) {
        mUpdateLayer = true;
    }
}

@Override
public final void draw(Canvas canvas) {
    ...

    if (canvas.isHardwareAccelerated()) {
        DisplayListCanvas displayListCanvas = (DisplayListCanvas) canvas;

        TextureLayer layer = getTextureLayer();
        if (layer != null) {
            applyUpdate();
            applyTransformMatrix();

            mLayer.setLayerPaint(mLayerPaint); // ensure layer paint is up
            displayListCanvas.drawTextureLayer(layer);
        }
    }
}
```

1. 如上面的 TextureView 代码所示，TextureView 通过回调在 UI 线程收到 SurfaceTexture 的 OnFrameAvailable 消息后，它会先 UpdateLayer 然后再 Invalidate 自己；
2. UpdateLayer 只是设置一个 mUpdateLayer 的标记，Invalidate 会触发 UI 绘制下一帧；
3. UI 线程在绘制新的一帧的过程，图中的Draw (CPU)，实际上就是调用被 Invalidated 的 View 的 draw 方法；
4. TextureView 的 draw 主要做两件事情，第一是 applyUpdate，applyUpdate 实际上是生成一个 UpdateSurfaceTexture 的任务等待 Android Render 线程执行，第二是生成一个 DrawTextureLayer 的指令到自己的 DisplayList，这里的 TextureLayer 实际上又是对 SurfaceTexture 的封装；
5. 当 UI 线程的 Draw (CPU) 完成后，意味着更新完所有需要更新的 View 的 DisplayList，此时它会请求 Android Render 线程开始绘制这些 DisplayList；

在 Android Render 线程：

1. 在 Draw (GPU) 的过程中，会先执行 UpdateSurfaceTexture 任务，它实际上是调用 SurfaceTexture 的 UpdateTexImage；

释放旧的 Buffer 回 Queue；
3. 接下来就是生成每个 DisplayList 里面的绘制指令对应的 GL 指令进行绘制；
4. DrawTextureLayer 指令其实就是 TextureLayer 关联的 SurfaceTexture 的绘制，使用跟 SurfaceTexture 关联的 Texture ID 作为纹理绘制一个多边形，通过这种方式将 2 中的 Buffer 输出到当前 Activity 的 Window 上；

上面就是完整的 TextureView 一帧的绘制和输出的全流程，需要经过三个不同的线程。从这个流程我们可以看到 TextureView 除了增加了额外的 GPU 开销导致性能低于 SurfaceView 以外，因为绘制完成的 Buffer 需要经过 Android 的 UI Renderer 输出，所以也增加了一个 VSync 周期的输出延迟，对游戏来说，输出延迟的增加通常是不可接受的，当然正常情况下，游戏不需要也不会使用 TextureView。但是对一般应用来说，输出延迟的增加也不见得完全是坏事，一来一般应用对输出延迟要求没那么高，二来虽然增加了输出延迟，但是这也意味着渲染流水线更长了，而更长的流水线意味着更高的吞吐量和宽容度，让我们有机会可以让帧率更平稳。

TextureView 渲染流水线的问题

我们通常计算帧率的方式是通过一段时间统计绘制的帧数，计算出一秒内绘制了多少帧。TextureView 的渲染可能会碰到下面的问题，当我们对 TextureView 的绘制做帧率计算时，会得到一个基本满帧的比较理想的结果，但是却发现实际的输出动画很容易出现卡顿和抖动。这一章节就是为了结合上面对 TextureView 渲染流水线的说明来解释问题可能发生的原因和一些补救的办法。

> 对一个以 VSync 信号驱动的绘制流程来说，所谓满帧就是帧率达到了屏幕的刷新率。假设屏幕刷新率是 60hz，那系统发送 VSync 信号的频率也同样是 60hz，满帧就是 60 帧，而一个 VSync 周期就是 1/60 = 16.7毫秒。

卡顿问题

卡顿通常是绘制时间过长导致的，一般绘制时间受绘制内容和 CPU 调度的影响，通常不是恒定而是上下波动的。所以虽然计算出来的平均帧率是满帧，但实际上很有可能个别帧的绘制时间超过了一个 VSync 周期，这在 TextureView 的渲染流水线上会导致丢帧（如果使用 SurfaceView，由于 Triple Buffers 的缘故不一定会丢帧）。甚至某一帧的绘制时间很接近但是没有超过一个 VSync 周期，比如说 16ms，在 TextureView 的渲染流水线上也有一定概率会丢帧。

如果 TextureView 在第 N 个 VSync 周期，绘制第 N 帧的时间超过一个 VSync 周期，比如说 17ms，也就是说 TextureView Render 线程完成 eglSwapBuffers，发送 OnFrameAvailable 消息这个时间达到了 17ms，此时 UI 线程已经进入第 N + 1 个 VSync 周期，开始处理下一个 VSync 信号（Choreographer#doFrame），所以这个 OnFrameAvailable 要等 UI 线程处理完 VSync，退出当前消息循环后才有机会被处理。因为没有 OnFrameAvailable 触发 updateLayer 和 invalidat
UI 线程在 VSync N + 1 信号的处理过程中不会去调用 TextureView 的 draw 方法，自然也不会

而不是新的 Buffer。**也就是说，上述情况下 TextureView 的第 N 帧绘制赶不上 UI Renderer 第 N + 1 帧的输出，UI Renderer 输出第 N + 1 帧时还是用的 TextureView 在第 N - 1 帧绘制的 Buffer。**

甚至我们假设 TextureView Render 线程完成 eglSwapBuffers 并且 Queue 新的 Buffer 到 SurfaceTexture 的队列里面的时候只用了 16ms，还没有超过一个 VSync 周期，但是由于下面的一些原因，仍然有可能出现上述的情况：

1. TextureView Render 线程 Queue Buffer 之后没有获得 CPU 时间，当它重新获得调度发出 OnFrameAvailable 消息时已经超过了一个 VSync 周期；
2. TextureView Render 线程在一个 VSync 周期内完成了绘制并发送 OnFrameAvailable 消息，但是 UI 线程在处理别的消息，然后马上进入下一个 VSync 信号的处理，OnFrameAvailable 消息仍然在 VSync 后面被处理；

但是如前所述，TextureView 的第 N 帧绘制需要在 UI Renderer 第 N + 1 帧输出，虽然这带来了一个 VSync 周期的输出延迟，但是更长的流水线深度也给我们一些机会来减少丢帧的概率。从之前的 TextureView 的渲染流水线分析我们可以看到，TextureView 绘制跟输出真正的同步点并不是在 UI 线程的 Draw (CPU)，而是在 Android Render 线程的 Draw (GPU)。但是我们又必须先要触发在 Draw (CPU) 中调用 TextureView.draw 方法，并且 mUpdateLayer 被提前设置为 true，这样才会产生新的 UpdateSurfaceTexture 任务在 Android Render 线程的 Draw (GPU) 中调用，这样 Android UI Renderer 才会从 SurfaceTexture 取出新的 Buffer 更新关联的 Texture。

我们再重新复述一遍：

1. TextureView 绘制跟输出真正的同步点在 Android Render 线程的 Draw (GPU)；
2. TextureView 第 N 帧的绘制，eglSwapBuffers 只要在 Android UI Renderer 第 N + 1 帧的 Draw (GPU) 开始之前完成，理论上这一帧不会丢失；
3. Draw (GPU) 的开始距离 VSync 的开始通常有一定时间，UI 线程需先完成事件处理，重布局，其它 VSync 回调的处理，Draw (CPU) 等等，这段时间一般会有 2 ～ 3ms；
4. 所以 TextureView 第 N 帧的绘制实际上会有一个 VSync 周期加上 2 ～ 3ms 的宽裕时间；
5. 如果 TextureView 第 N + 1 帧的绘制耗时比较短，补上第 N 帧超出 VSync 周期的部分，第 N + 1 帧也不会丢失；
6. 但是要触发 Draw (GPU) 调用 SurfaceTexture.updateTexImage 去同步，必须在 UI 线程开始第 N + 1 帧的 VSync 处理之前触发 TextureView 的 updateLayer 和 invalidate 方法；
7. Android 目前触发的方式是通过发送 OnFrameAvailable 消息给 TextureView 在 UI 线程处理；
8. OnFrameAvailable 消息发送或者被处理的时机过晚，就会导致丢帧，即使我们满足了条件 2；

从上面的叙述我们可以看到，TextureView 确实会出现 **即使我们满足了条件 2，理论上不会丢帧，但是因为 8，在实际中又导致了丢帧**这种诡异的现象。

MyOnFrameAvailable 的处理中一样去调用 TextureView 的 updateLayer 和 invalidate 方法。

发送 MyOnFrameAvailable 消息的时机需要选择比较适当，需要减少过早发送而实际上我们没有满足条件 2，导致 UI Renderer 绘制了多余的 N + 1 帧这种情况发生的概率（虽然无法完全避免）。我自己的选择是在调用 eglSwapBuffers 之前，一般对比系统的 OnFrameAvailable 消息提前 2 ~ 3ms。在测试中部分实际场景下，低端机的动画帧率可以提升 5 ~ 10% 左右，流畅度有较为明显的提升。

发送自己的 MyOnFrameAvailable 消息的另外一个好处是 TextureView 有时会中断自己的 OnFrameAvailable 消息的发送，虽然概率不高，不过的确有时会发生。

> TextureView.updateLayer 是私有 API，一般需要通过反射调用，并且在 Android 9+ 需要规避系统对私有 API 的限制。不过也有一种比较简单的方法可以达到同样的效果，就是调用两次 setOpaque，先用跟当前相反的值调用一次，再用原值调用多一次，这样既没有改变 Opaque 属性，又触发了 updateLayer。

```
public void setOpaque(boolean opaque) {
        if (opaque != mOpaque) {
            mOpaque = opaque;
            if (mLayer != null) {
                updateLayerAndInvalidate();
            }
        }
    }
```

抖动问题

如果能够理解上面的卡顿问题，要理解抖动问题也就比较容易了。抖动通常是吞帧导致的，出现的原因一般是 TextureView 绘制的速度比较快，而 UI 线程则有严重的阻塞现象。我们来看下面一个吞帧的现象：

1. TextureView 绘制第 N - 1 帧（VSync N - 1）；
2. UI Renderer 输出第 N 帧，对应 TextureView 第 N - 1 帧的内容，TextureView 同一时间绘制了第 N 帧（VSync N）；
3. UI Renderer 开始输出第 N + 1 帧，但是 UI 线程阻塞严重，到了 Draw (GPU) 时已经消耗了较长的时间，而 TextureView 在 Draw (GPU) 之前就已经完成第 N + 1 帧的绘制（VSync N + 1）；
4. UI Renderer 第 N + 1 帧 Draw (GPU) 在调用 SurfaceTexture.updateTexImage 更新到的 Buffer 实际上是 TextureView 绘制的第 N + 1 帧的 Buffer，所以 UI Renderer 输出第 N + 1 帧对应的是 TextureView 第 N + 1 帧的内容（VSync N + 1）；

2）；

从上面的描述我们就可以了解到，在这个帧序列中，TextureView 绘制的第 N 帧的内容实际上是被吞掉了，**它还没来得及输出就被第 N + 1 帧绘制的内容给覆盖了**，并且第 N + 1 帧连续输出了两次，分别在 UI Renderer 的第 N + 1 和 N + 2 帧。如果在动画过程中频繁出现这种吞帧的现象，动画就会出现比较明显的抖动，看起来时快时慢，步调明显不协调。

要避免吞帧的现象，最简单的办法就是避免 UI 线程的阻塞，保证 Draw (GPU) 被及时调用。我们需要检查是否在 UI 线程的 VSync 处理期间做了太多事情：

1. 应用是否在自己的 VSync 回调花费了太长的时间；
2. 应用是否同时有其它 View 在不断触发重绘，并且绘制的耗时较长；

另外也可以提高 UI 线程的优先级来获得更多的 CPU 时间，特别是避免被其它后台线程抢占太多的 CPU 时间。Android 8 开始系统已经把 UI 线程从 THREAD_PRIORITY_DEFAULT 提高到 THREAD_PRIORITY_VIDEO 的优先级，这是一个相当高的优先级，基本上除了少数特殊的系统线程外，其它线程都不会高于这个优先级（包括优先级为 THREAD_PRIORITY_DISPLAY 的 Android Render 线程），所以我们也可以在 Android 7 以下的版本提高 UI 线程的优先级，参考下面的代码。

```
private void checkUIThreadPriority() {
        // On Android 8+, UI thread's priority already increase from 0 to -10(
        // higher than URGENT_DISPLAY (-8), we at least adjust to URGENT_DISPLA
        // and it will help to improve TextureView performance
        try {
            if (Process.getThreadPriority(0) > Process.THREAD_PRIORITY_URGENT_I
                Log.e(TAG, "UI thread priority=" +
                        Process.getThreadPriority(0) + ", need to raise!");
                if (ANDROID_8_OR_ABOVE)
                    Process.setThreadPriority(Process.THREAD_PRIORITY_VIDEO);
                else
                    Process.setThreadPriority(Process.THREAD_PRIORITY_URGENT_DI
            }
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
```

最坏的情况我们还可以人为阻塞 TextureView Render 线程，增加 TextureView 绘制每一帧的耗时，不过这种方法一般不推荐使用。

总的来说，TextureView 是一个不得已的选择，它有明显的性能缺陷，还有同步机制导致的一些问题，了解它的渲染流水线有助于帮助我们规避一些比较明显的问题，希望这篇文章对读者解决 TextureView 的问题能够有所帮助。

发布于 2020-06-10

Android 开发

文章被以下专栏收录

THINK.DESIGN.CODE
关注浏览器渲染和Web技术演进

推荐阅读

TextureView 的血与泪（一）

越来越多的应用需要使用自己的绘制引擎进行复杂内容的绘制，比如需要使用 GL 绘制 3D 的内容，或者绘制复杂的文档，图表时不希望阻塞 UI 线程，或者部分内容是通过类似 Flutter 这样的第三...

易旭昕 发表于THINK...

不曾熟悉过的odex（编译过程）

Kelvi... 发表于Andro...

Android系统中动画浅析

Kelvi... 发表于

易旭昕 (作者) 回复 **龙泉寺扫地僧** 2020-06-10

SV只能用在特殊场景，大部分时候还是要用TV

1

fred 郑 06-09

TextureView为啥不能用Triple Buffers呢?

赞

fred 郑 06-09

SurfaceView也可以lockHardwareCanvas用硬件加速吧，这种情况下也是三个线程

赞

hiten 05-24

大佬，小弟有一事不明，为啥TextureView Render 是由 VSync 信号驱动的

赞

易旭昕 (作者) 回复 **hiten** 05-24

理论上这是自己的线程，用什么驱动都可以，但是一般还是用 vsync 驱动，调度上比较合理

赞

小朱 01-14

最近在做一个直播应用，用的MediaCodec+TextureView，直播帧率设为30fps时，通过PerfDog检测手机的刷新还是很稳定一直都是30fps左右，但是当直播帧率设为60fps时，手机的刷新就很不稳定了在30-50之间波动，一直不知道怎么解决，打日志每秒收到的帧数都是60，每解码也是60，但是工具抓到的手机渲染帧率就是不稳定30-50波动。

赞

龙泉寺扫地僧 2020-06-10

易大终于更新了！

赞

易旭昕 (作者) 回复 **龙泉寺扫地僧** 2020-06-10

哈，好久没写一篇完整的了

赞