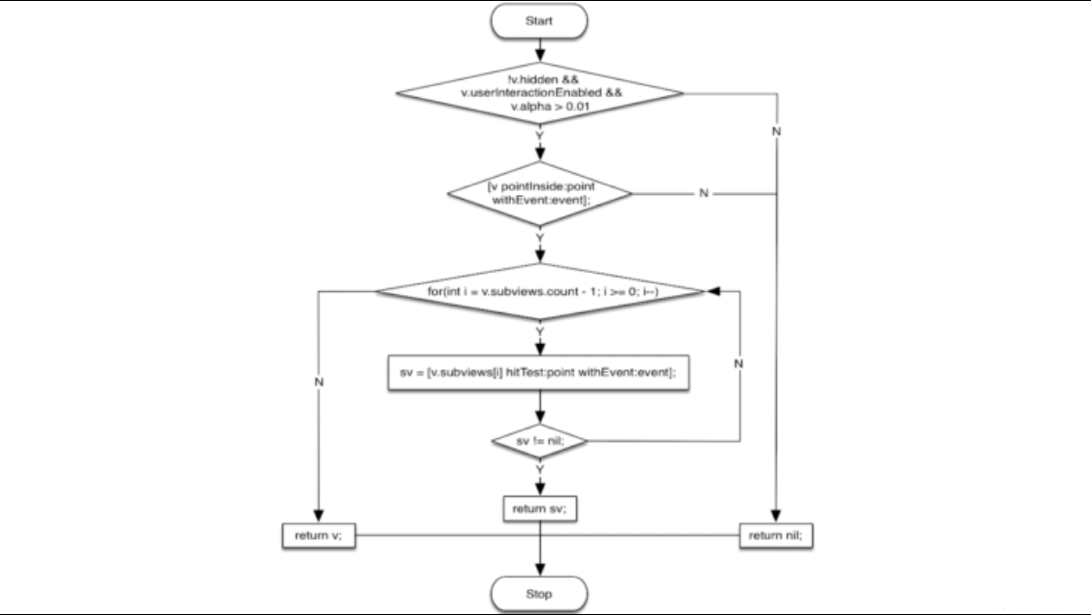
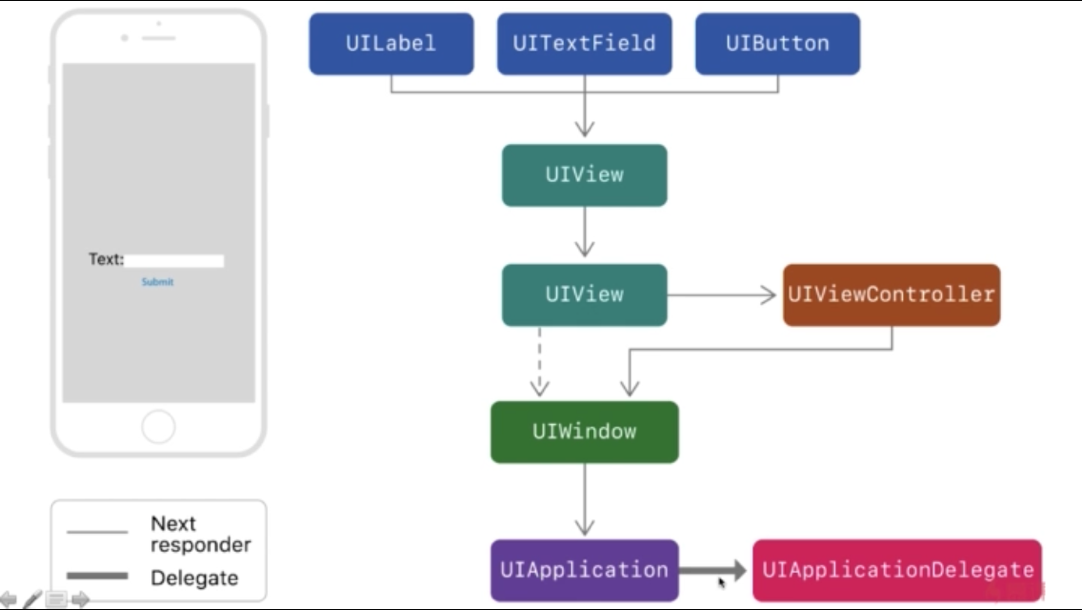
# UI 视图

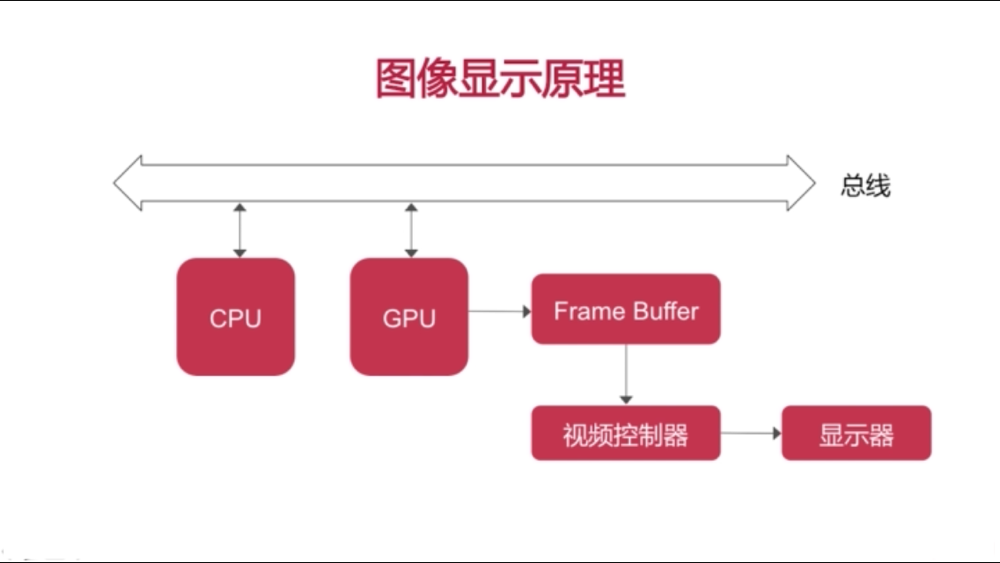
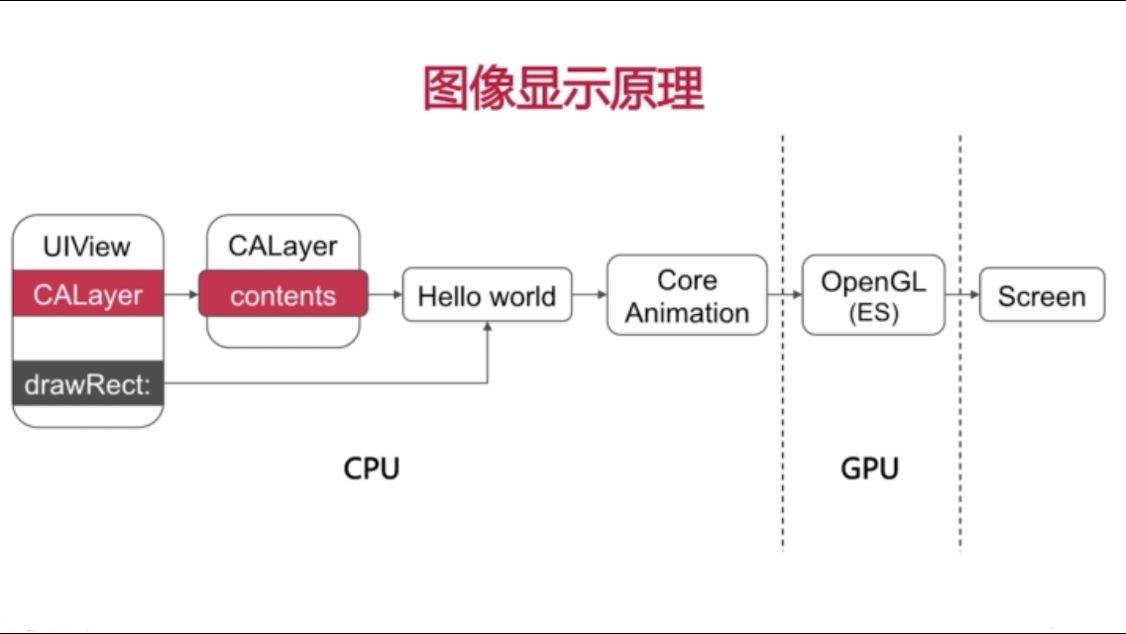
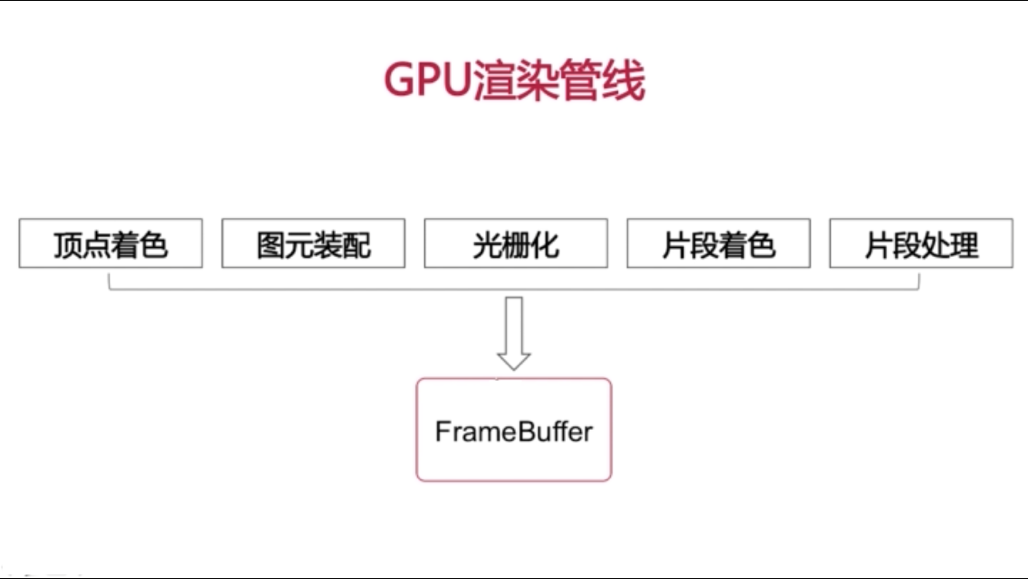
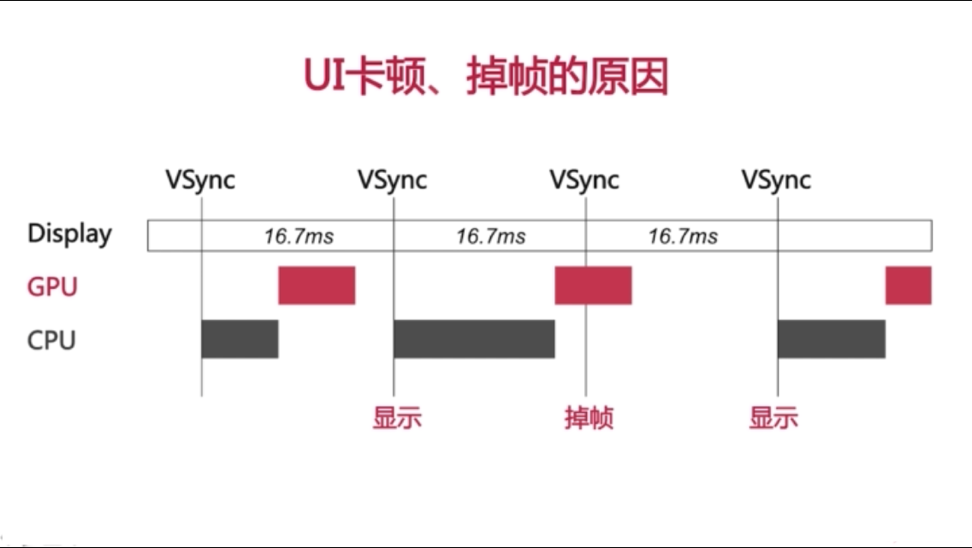
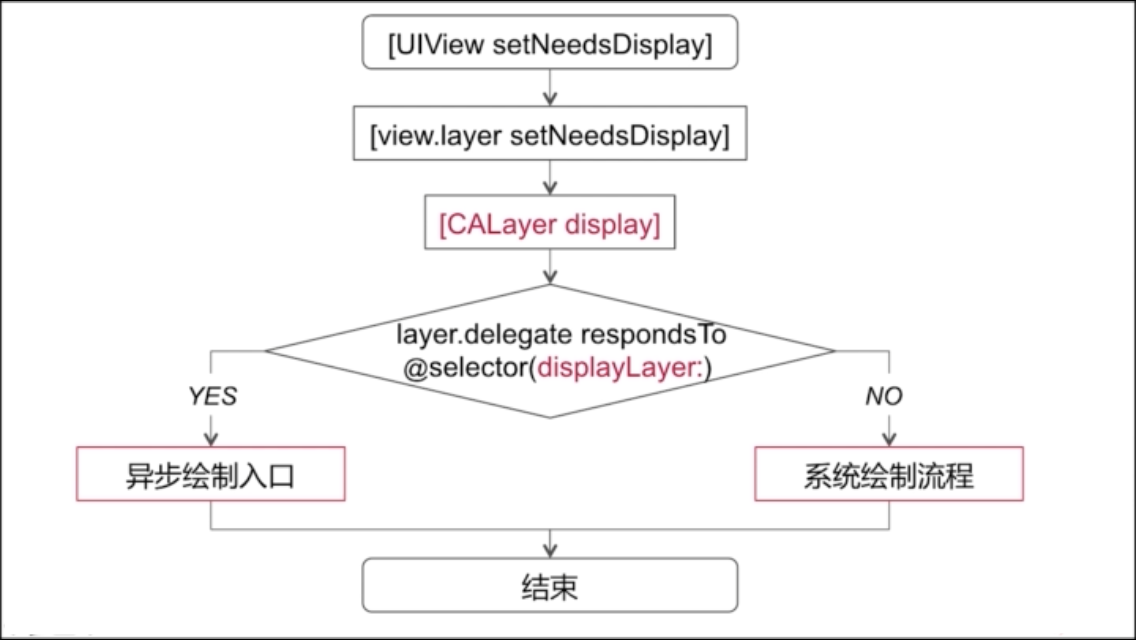
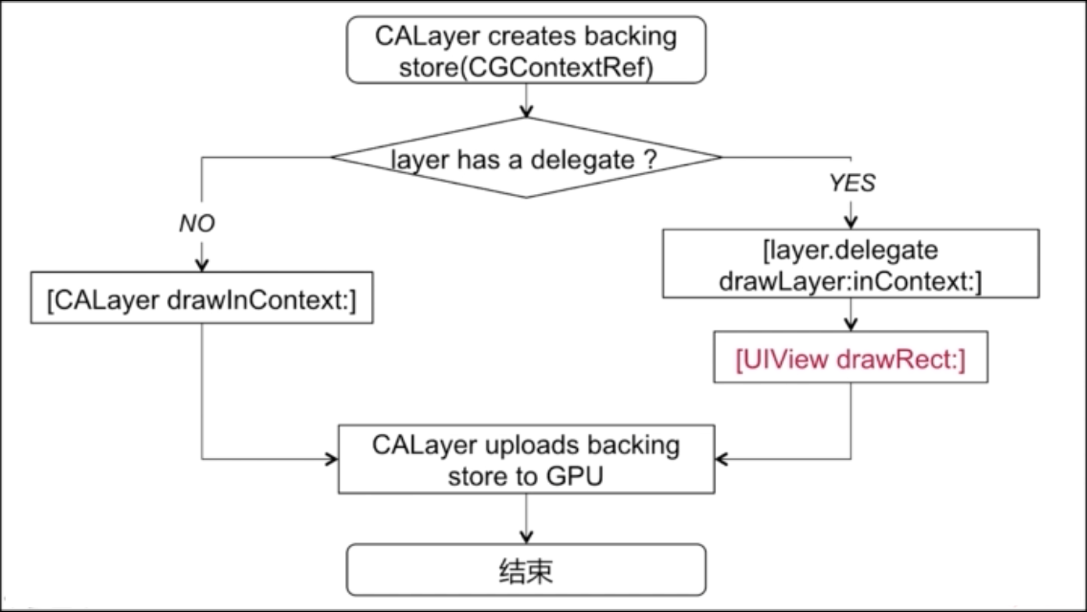
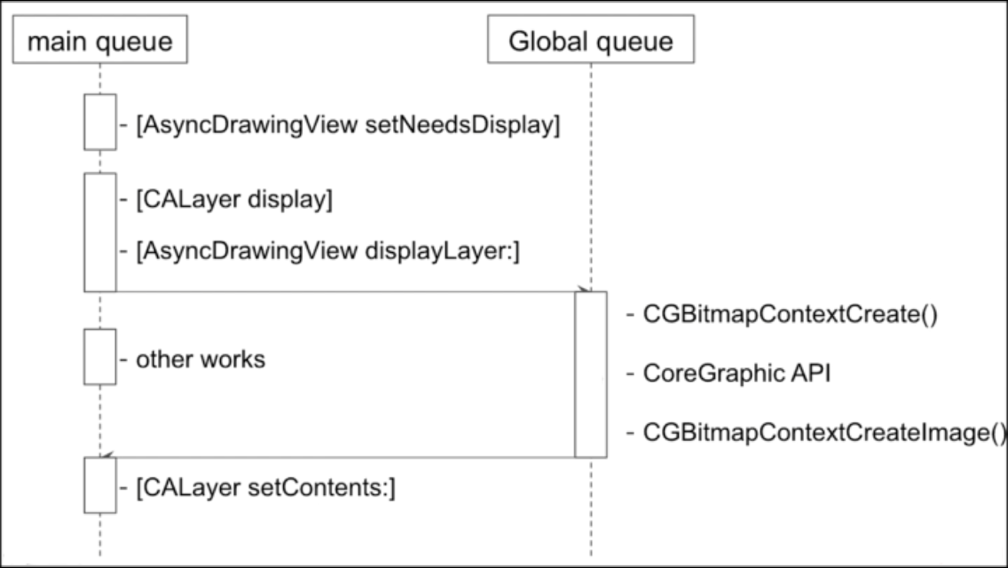
* UITableView cell的重用机制与实现原理
  + 重用机制实现了数据和显示的分离,并不为每个数据创建一个UITableViewCell,我们只创建屏幕可显示的最大的cell个数+1,然后去循环重复使用这些cell,既节省空间,又达到我们需要显示的效果.  
    这种机制下系统默认有一个可变数组NSMutableArray\* visiableCells,用来保存当前显示的cell.一个可变字典NSMutableDictnery\* reusableTableCells,用来保存可重复利用的cell.(之所以用字典是因为可重用的cell有不止一种样式,我们需要根据它的reuseIdentifier,也就是所谓的重用标示符来查找是否有可重用的该样式的cell).
* 如何对UITableView的滚动加载进行优化，防止卡顿？
  + UITableView的滚动优化主要在于以下两个方面：
    - 减少cellForRowAtIndexPath代理中的计算量（cell的内容计算）
      * 首先要提前计算每个cell中需要的一些基本数据，代理调用的时候直接取出；
      * 图片要异步加载，加载完成后再根据cell内部UIImageView的引用设置图片；
      * 图片数量多时，图片的尺寸要跟据需要提前经过transform矩阵变换压缩好（直接设置图片的contentMode让其自行压缩仍然会影响滚动效率），必要的时候要准备好预览图和高清图，需要时再加载高清图。
      * 图片的‘懒加载'方法，即延迟加载，当滚动速度很快时避免频繁请求服务器数据。
      * 尽量手动Drawing视图提升流畅性，而不是直接子类化UITableViewCell，然后覆盖drawRect方法，因为cell中不是只有一个contentview。绘制cell不建议使用UIView，建议使用CALayer。原因要参考UIView和CALayer的区别和联系。
    - 减少heightForRowAtIndexPath代理中的计算量（cell的高度计算）
      * 由于每次TableView进行update更新都会对每一个cell调用heightForRowAtIndexPath代理取得最新的height，会大大增加计算时间。如果表格的所有cell高度都是固定的，那么去掉heightForRowAtIndexPath代理，直接设置TableView的rowHeight属性为固定的高度；
      * 如果高度不固定，应尽量将cell的高度数据计算好并储存起来，代理调用的时候直接取，即将height的计算时间复杂度降到O(1)。例如：在异步请求服务器数据时，提前将cell高度计算好并作为dataSource的一个数据存到数据库供随时取用。
    - 异步绘制cell 内容
    - 滚动很快时，只加载目标范围内的Cell，这样按需加载，极大的提高流畅度。
  + 总结：
    - 提前计算并缓存好高度，因为heightForRow最频繁的调用。
    - 异步绘制，遇到复杂界面，性能瓶颈时，可能是突破口。
    - 滑动时按需加载，这个在大量图片展示，网络加载时，很管用。（sdwebimage已经实现异步加载）。
    - 重用cells。
    - 如果cell内显示得内容来自web，使用异步加载，缓存结果请求。
    - 少用或不用透明图层，使用不透明视图。
    - 尽量使所有的view opaque，包括cell本身。
    - 减少subViews
    - 少用addView给cell动态添加view，可以初始化的时候就添加，然后通过hide控制是否显示。
* UITableView 数据源同步
  + 并发访问，数据拷贝（弊端：内存耗费大）
  + 串行访问（无论在主线程还是子线程，对于数据源的操作都在串行队列上实现，保证了数据源的同步，避免UI刷新错乱；弊端：子线程处理耗时任务时，另一个比如删除操作就会有延时）
* 事件传递与响应
  + UIView 为CALayer提供内容，以及负责处理触摸等事件，参与响应链
  + CALayer 负责显示内容
    - 单一职责设计模式（UIView和CaLyer各有分工）
  + 事件传递流程：
    - ？为什么点击按钮会调两次hitTest:withEvent:
    - 覆盖了按钮的touchesBegan:withEvent:等方法后，需要显示调用[super touchesBegan:withEvent:],不然按钮的targetAction不能被调用





* + 事件响应流程：（与事件传递流程相反）



* UI图像显示原理
  + CPU： 输出位图，经由总线，传给GPU；
  + GPU： GUP拿到位图后，做图层渲染（包括纹理合成），然后放到帧缓冲区，由视频控制器根据vsync信号在指定时间之前去提取对应帧缓冲区中的屏幕内容，显示到手机屏幕上；
  + CPU和GPU是通过总线连接起来的。
  + 
  + 
  + CUP承担
    - Layout：UI布局（frame设置等）；文本计算（文字size计算等）
    - Display：绘制（drawRect）
    - Prepare：图片编解码（UIImage设置时）
    - Commit：coreAnimation提交位图
  + GPU渲染管线
    - 
* UI卡顿&掉帧
  + 
  + 滑动优化方案：
    - CPU
      * 对象创建、调整、销毁 放到子线程
      * 预排版（布局计算、文本计算） 放到子线程
      * 预渲染（文本等异步绘制，图片编解码等）
    - GPU
      * 纹理渲染层面优化
        + 避免离屏渲染
      * 视图混合层面优化
* UI绘制原理&异步绘制
  + 
  + 系统绘制
    - 
  + 异步绘制 [layer.delegate displayLayer:]
    - 代理负责生成对应的bitmap
    - 设置该bitmap作为layer.contents属性的值
    - 
* UI离屏渲染
  + 在屏渲染：GPU的渲染操作是在当前用于显示的屏幕缓冲区中进行
  + 离屏渲染：GPU在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作
  + 什么是离屏渲染？当我们制定了UI视图的某些属性标记为它在未预合成之前不能用于当前屏幕上面进行直接显示的时候就会触发离屏渲染，而离屏渲染起源于GPU层面，指的是在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作。
  + 什么情况会触发离屏渲染？
    - 圆角（当和maskToBounds一起使用时）
    - 图层蒙版
    - 阴影
    - 光栅化
  + 为何要避免离屏渲染？（卡顿，掉帧）
    - 离屏渲染使GPU层面上触发了OpenGL的多通道渲染管线，产生了额外的开销（合成多通道渲染结果），所以我们需要避免离屏渲染。
    - 触发离屏渲染时会增加GPU的工作量，而增加工作量很有可能导致GPU和CPU的总耗时超过16.7ms，可能会导致UI的卡顿和掉帧