浅谈iOS视图的性能优化

教师端iOS开发 孙裔

如何找出影响视图的因素?

帧率	iOS设备上越接近60fps越好
绘制过程中CPU/GPU受限	利用率越低越好越省电
多余的CPU渲染工作	正确使用CPU参与渲染工作
太多的离屏渲染	代价太昂贵，少而精
太多的图层混合	GPU渲染不透明的图层效率更高
奇怪的图片格式或尺寸大小	即时转换、调整大小也会耗费资源
代价昂贵的视图或动画	合理使用高耗性能的视图或动画
较复杂的视图层级	你真正需要的视图层级是什么样的?

帧率

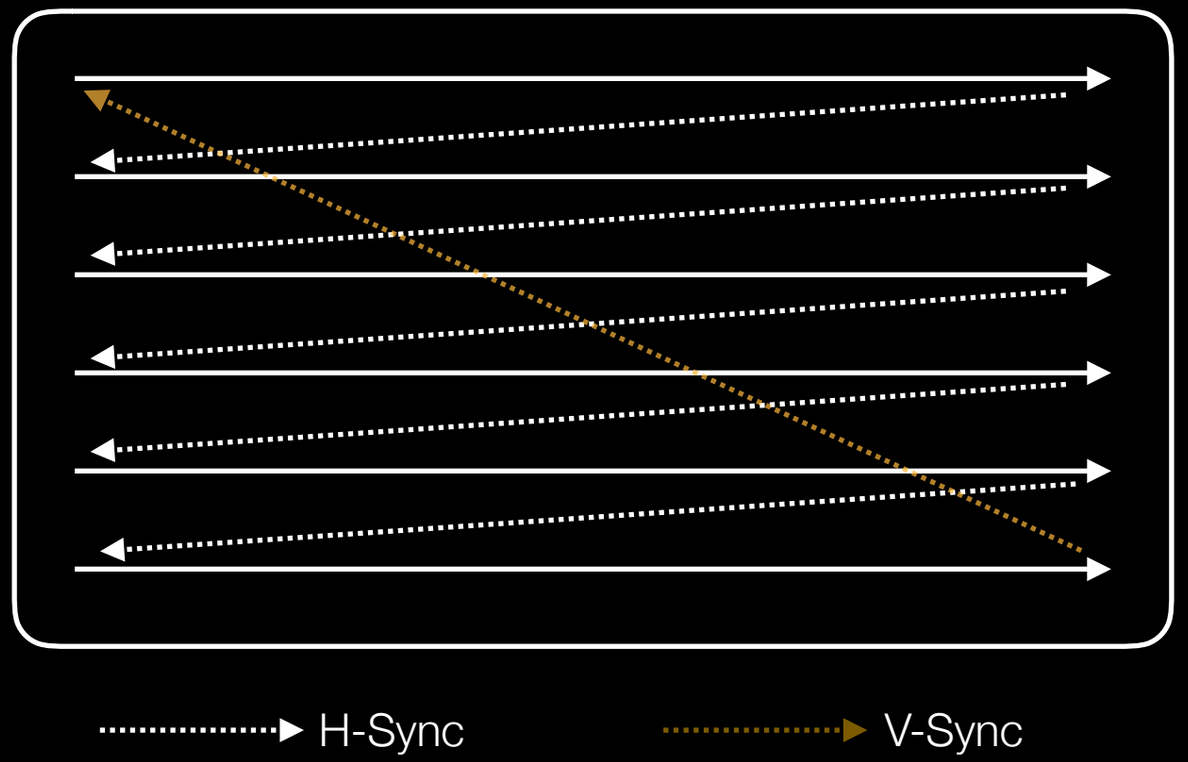
为什么是60fps（frame-per-second）

- 12 fps：由于人类眼睛的特殊生理结构，如果所看画面之帧率高于每秒约10-12帧的时候，就会认为是连贯的
- 24 fps：有声电影的拍摄及播放帧率均为每秒24帧，对一般人而言已算可接受
- 30 fps：早期的高动态电子游戏，帧率少于每秒30帧的话就会显得不连贯，这是因为没有动态模糊使流畅度降低
- 60 fps：在实际体验中，60帧相对于30帧有着更好的体验
- 85 fps：一般而言，大脑处理视频的极限

帧率

为什么是60fps（frame-per-second）

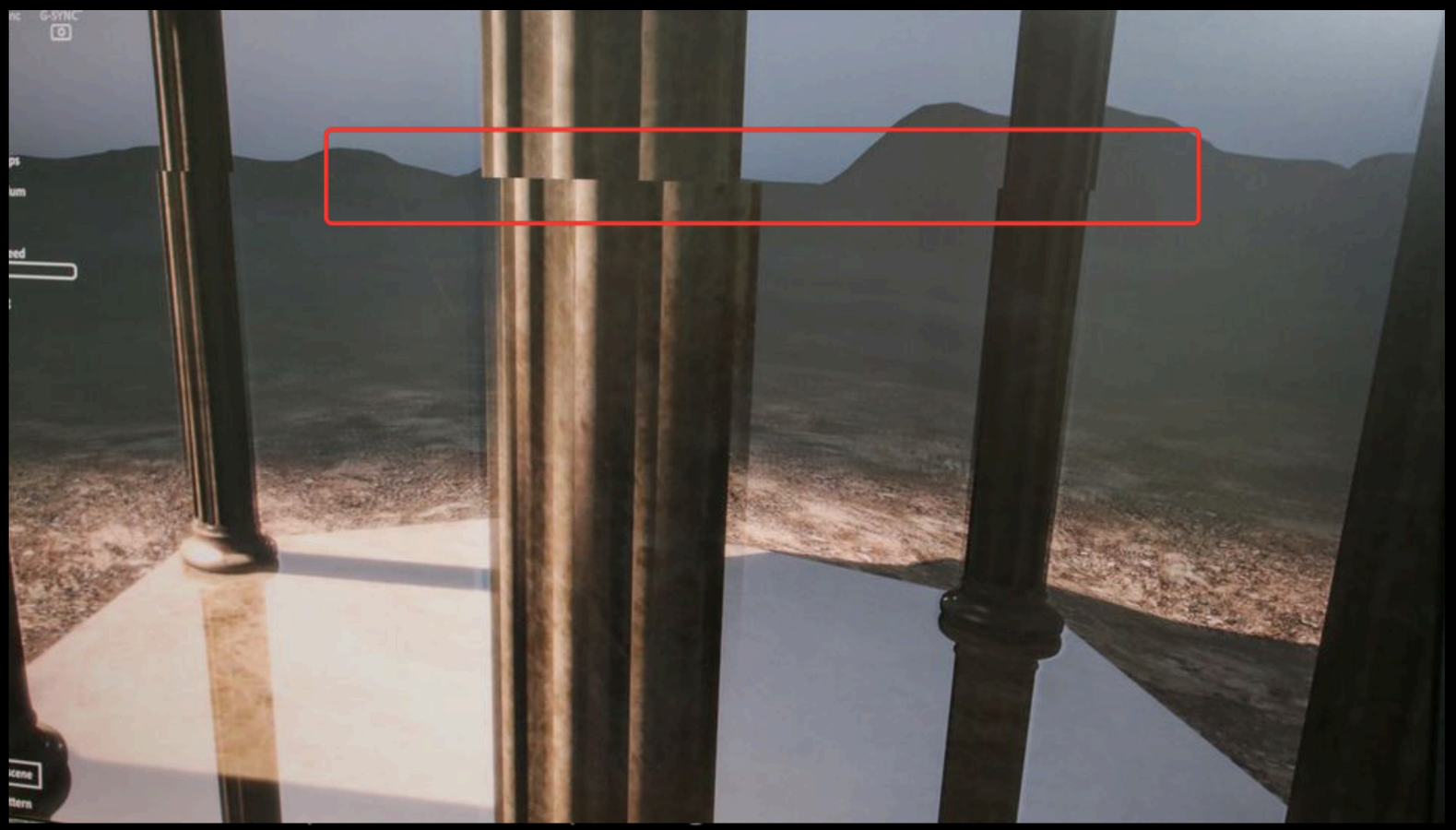
- 显示器
  刷新频率是固定的，常见的30Hz、60Hz、120Hz 或者 144Hz 的频率进行刷新。而其中最常见的刷新频率是60Hz。
- 显卡
  刷新频率是变化的。



帧率

为什么是60fps（frame-per-second）

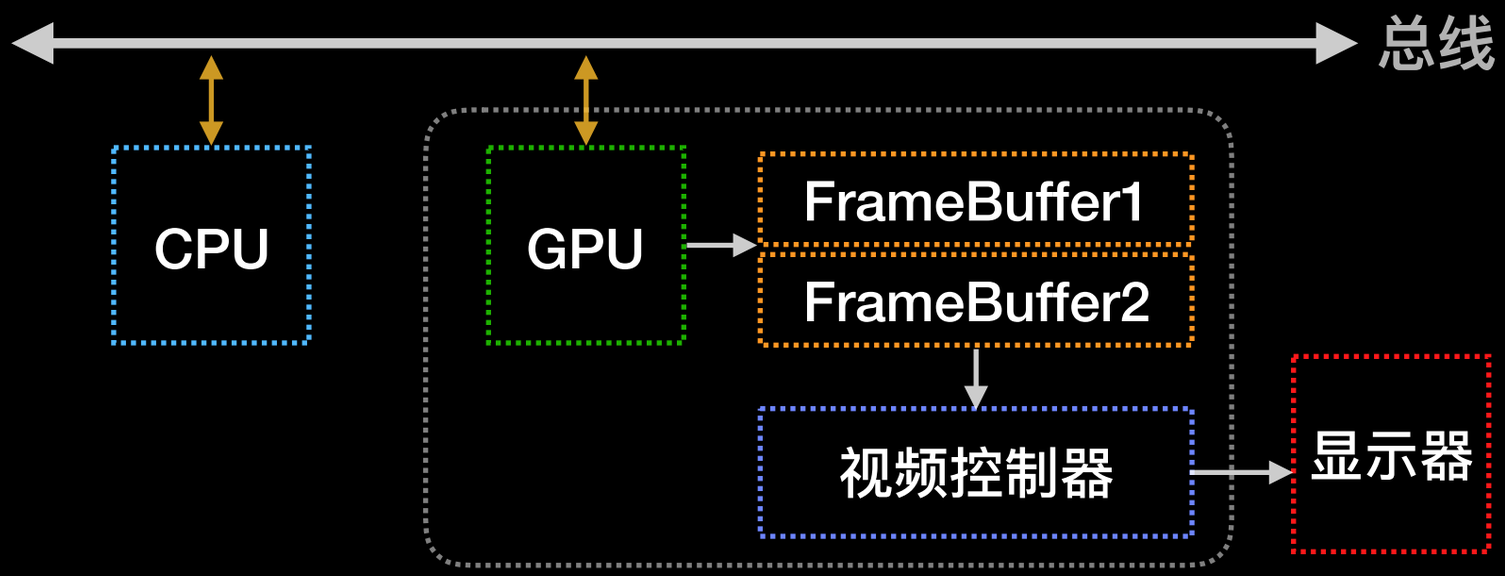
- 屏幕撕裂



帧率

为什么是60fps（frame-per-second）

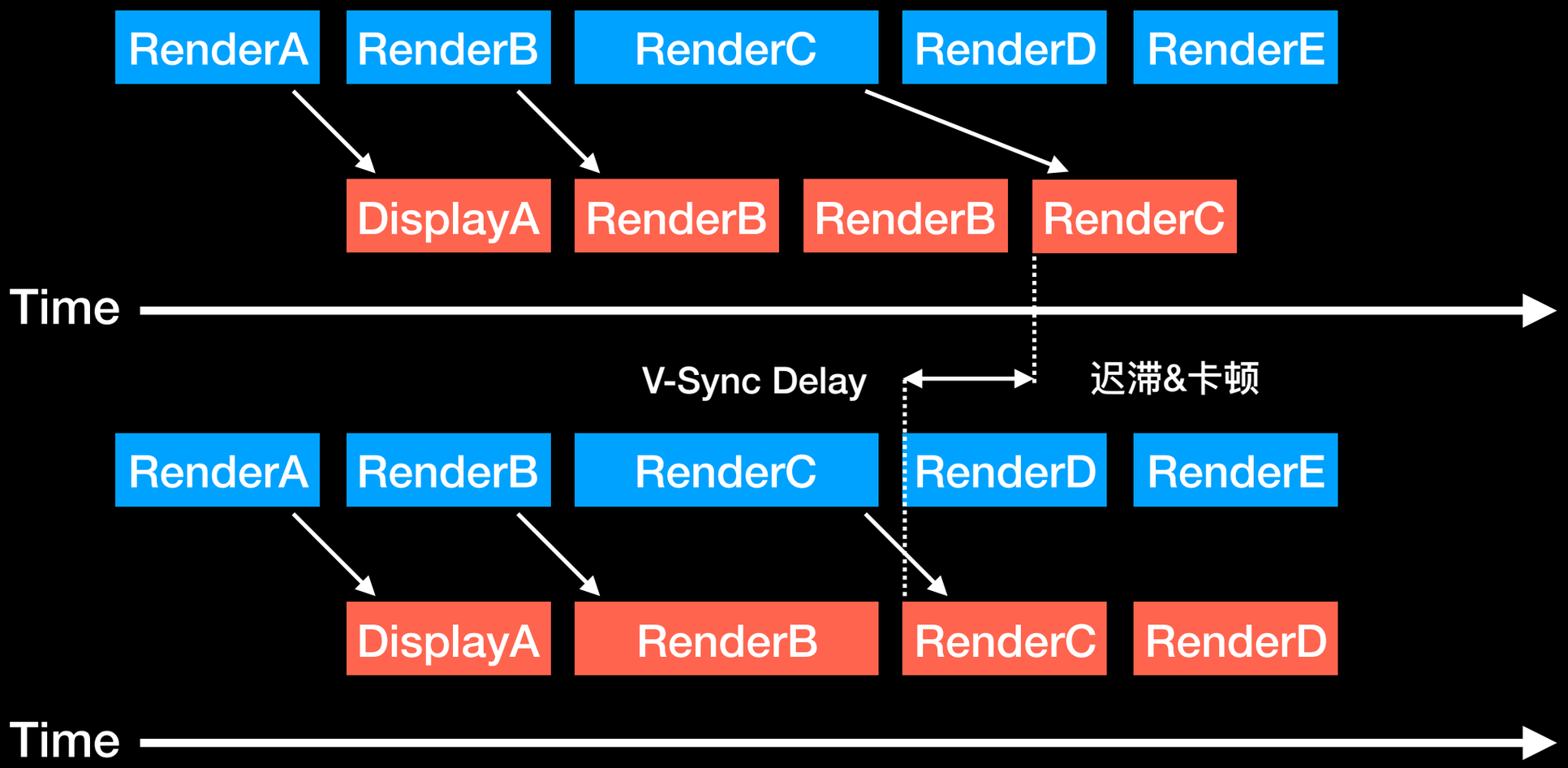
- 屏幕撕裂的解决方案：双缓存+垂直同步（V-Sync）



帧率

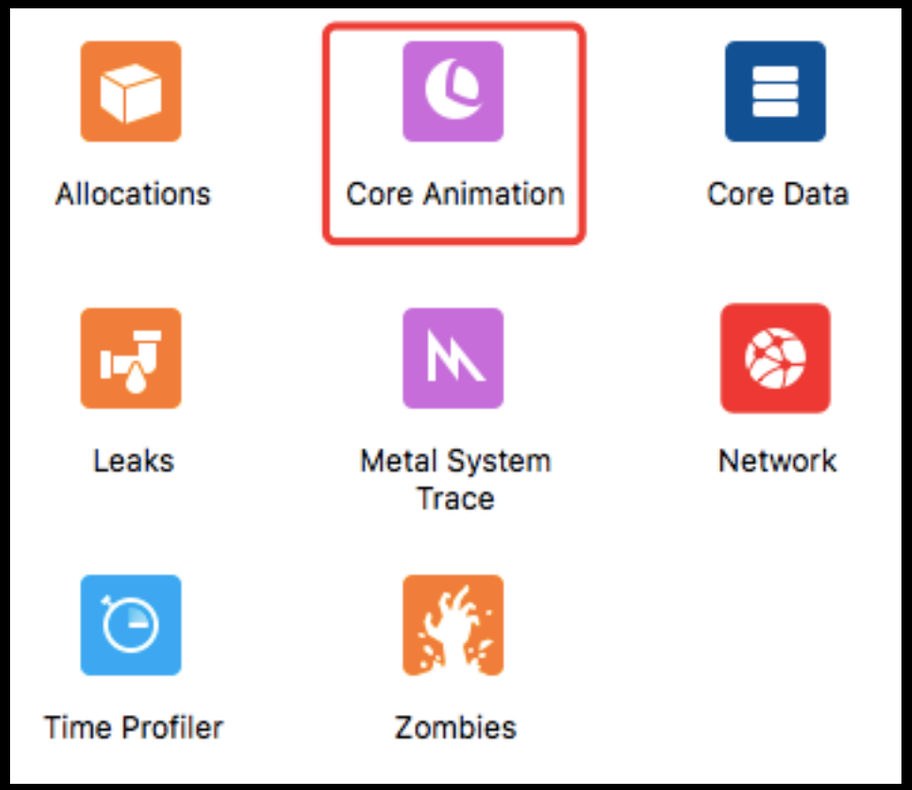
为什么是60fps（frame-per-second）

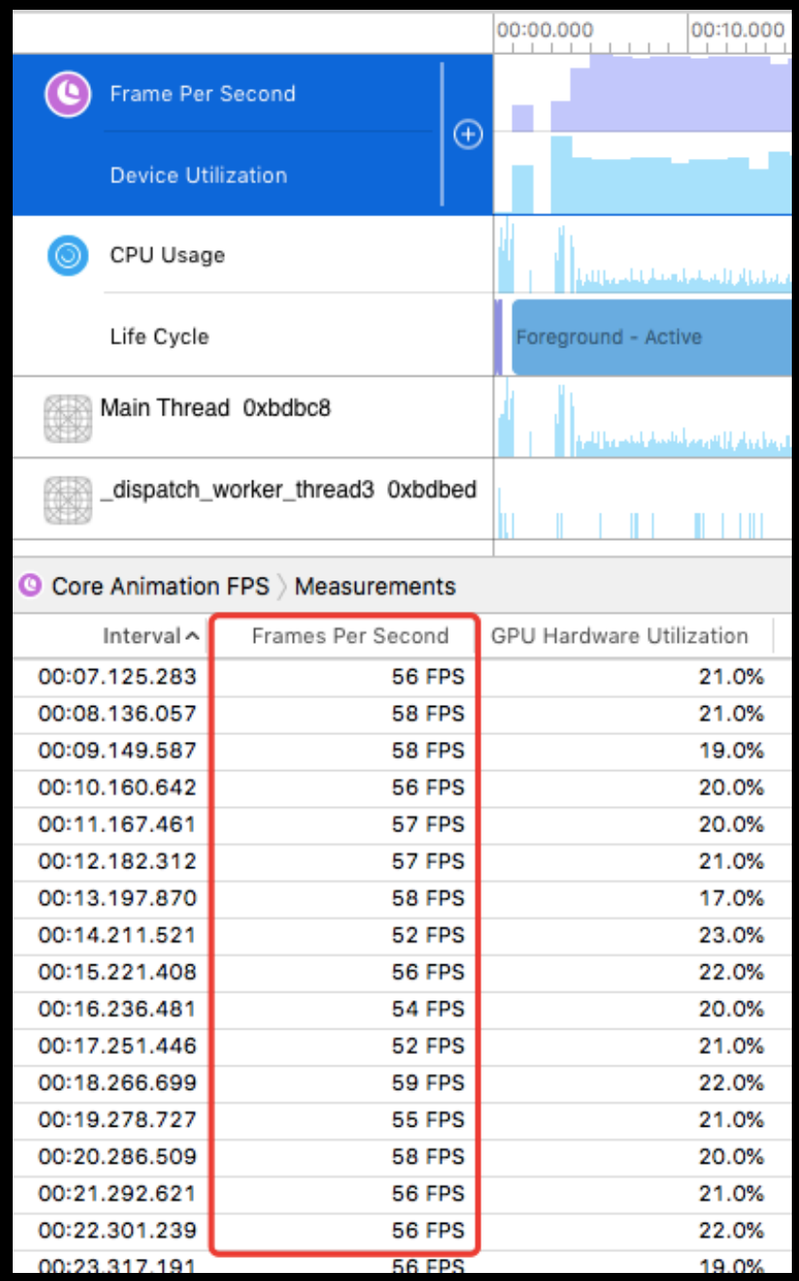
- 迟滞&卡顿



帧率

如何使用Instrument查看帧率?





Interval	Frames Per Second	GPU Hardware Utilization
00:07.125.283	56 FPS	21.0%
00:08.136.057	58 FPS	21.0%
00:09.149.587	58 FPS	19.0%
00:10.160.642	56 FPS	20.0%
00:11.167.461	57 FPS	20.0%
00:12.182.312	57 FPS	21.0%
00:13.197.870	58 FPS	17.0%
00:14.211.521	52 FPS	23.0%
00:15.221.408	56 FPS	22.0%
00:16.236.481	54 FPS	20.0%
00:17.251.446	52 FPS	21.0%
00:18.266.699	59 FPS	22.0%
00:19.278.727	55 FPS	21.0%
00:20.286.509	58 FPS	20.0%
00:21.292.621	56 FPS	21.0%
00:22.301.239	56 FPS	22.0%
00:23.317.191	56 FPS	19.0%

帧率

总结：

- 画面帧率越高，体验越好，但由于受到硬件设备的限制，目前显示性能的极限是60fps。
- 在iOS开发过程中，可以使用Instrument来查看App运行过程中的帧率变化，帧率越接近60越好。

绘制过程中CPU/GPU受限

CPU受限（CPU Bound）

因为CPU执行渲染任务耗时太久而导致帧渲染时间过长，我们称其为CPU受限。

GPU受限（GPU Bound）

因为GPU执行渲染任务耗时太久而导致帧渲染时间过长，我们称其为GPU受限。

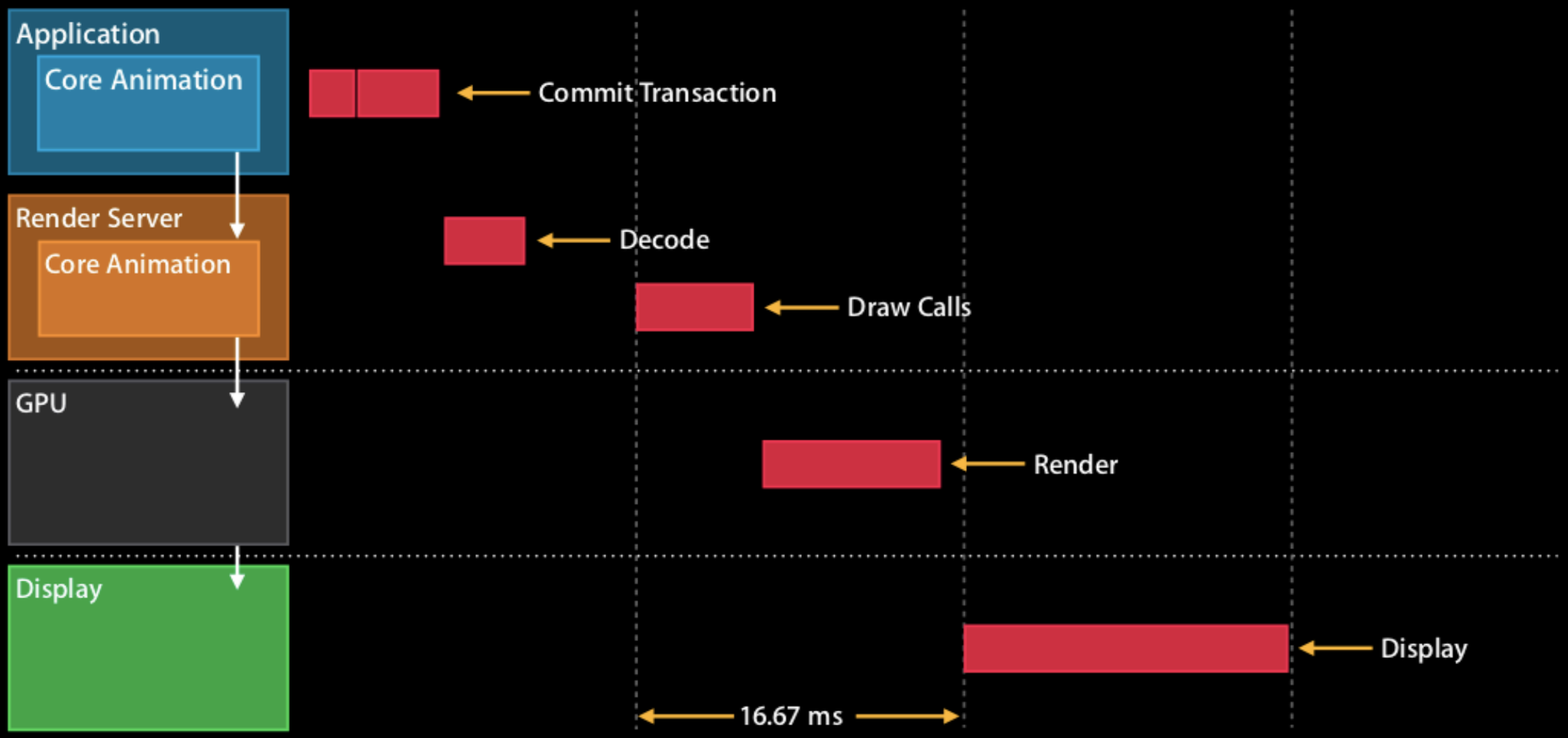
绘制过程中CPU/GPU受限

哪些工作可能会造成GPU受限?

- 纹理的渲染
- 视图的混合
- 图形的生成（边框、圆角、阴影、遮罩等）
- 过大的图片
- ...

绘制过程中CPU/GPU受限

CoreAnimation流水线



绘制过程中CPU/GPU受限

CoreAnimation流水线

提交事务（Commit Transaction）

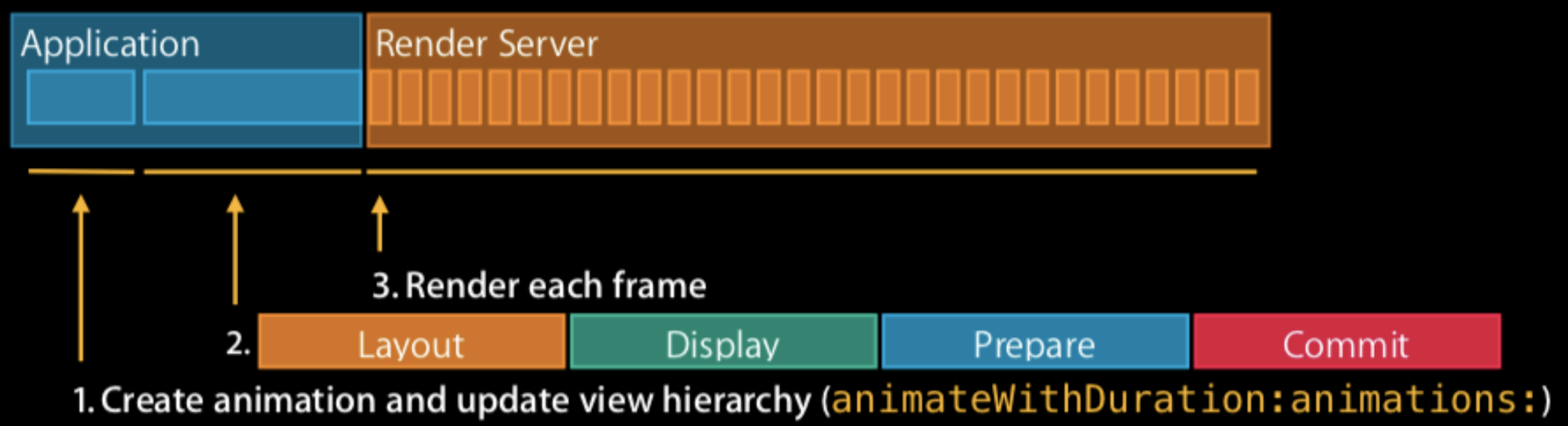
Layout	Display	Prepare	Commit
Set up the views • Layout subviews • View creation • Add subview • Populate content • Database lookups • CPU/IO Bound	**Draw the views** • drawRect • String drawing • CPU/Memory Bound	**Additional Core Animation work** • Image decoding • Image conversion	**Package up layers and send them to render server** • Recursive • Expensive is layer tree is complex

绘制过程中CPU/GPU受限

CoreAnimation流水线

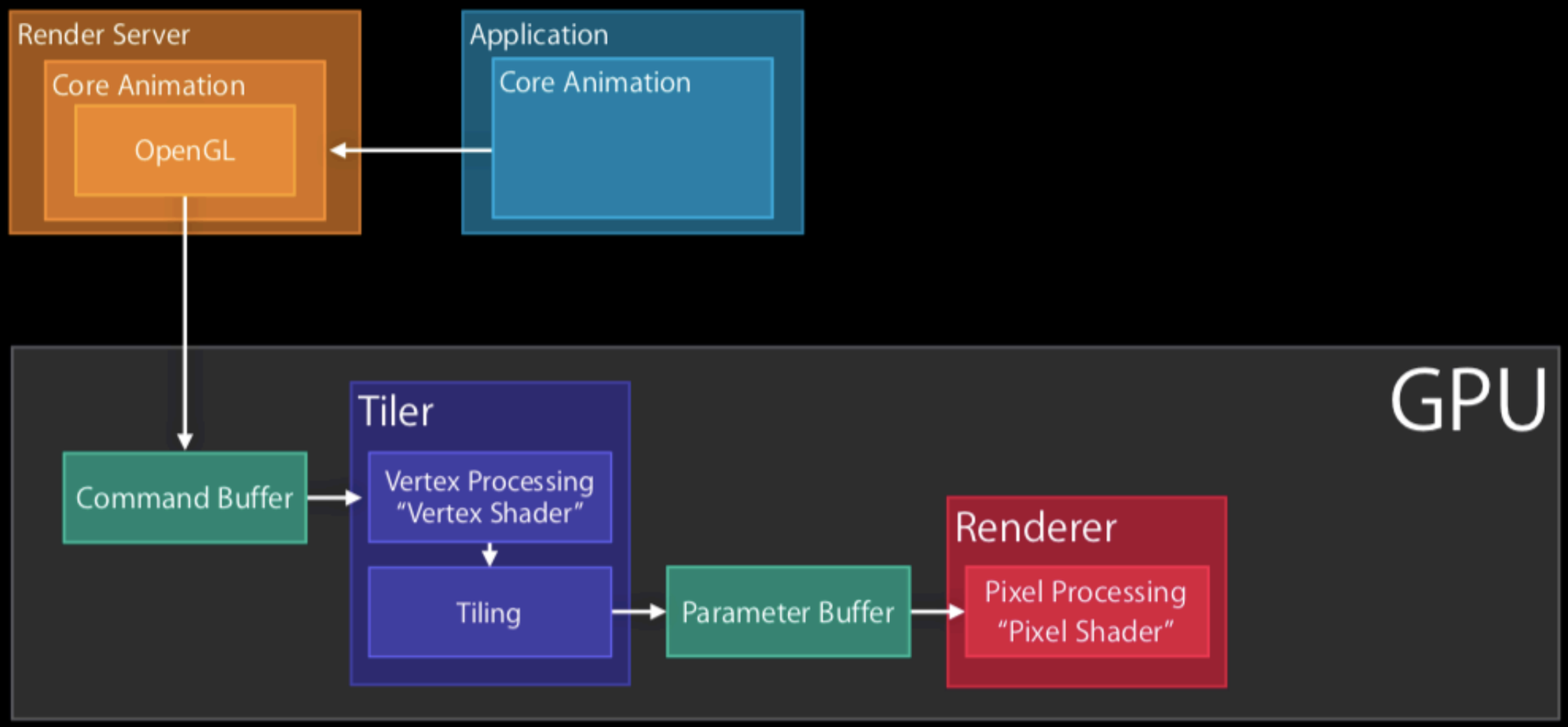
提交到Render Server后，Render Server会做以下两步工作：

- 对所有的图层属性计算中间值，设置OpenGL几何形状（纹理化的三角形）来执行渲染；
- 在屏幕上渲染可见的三角形。



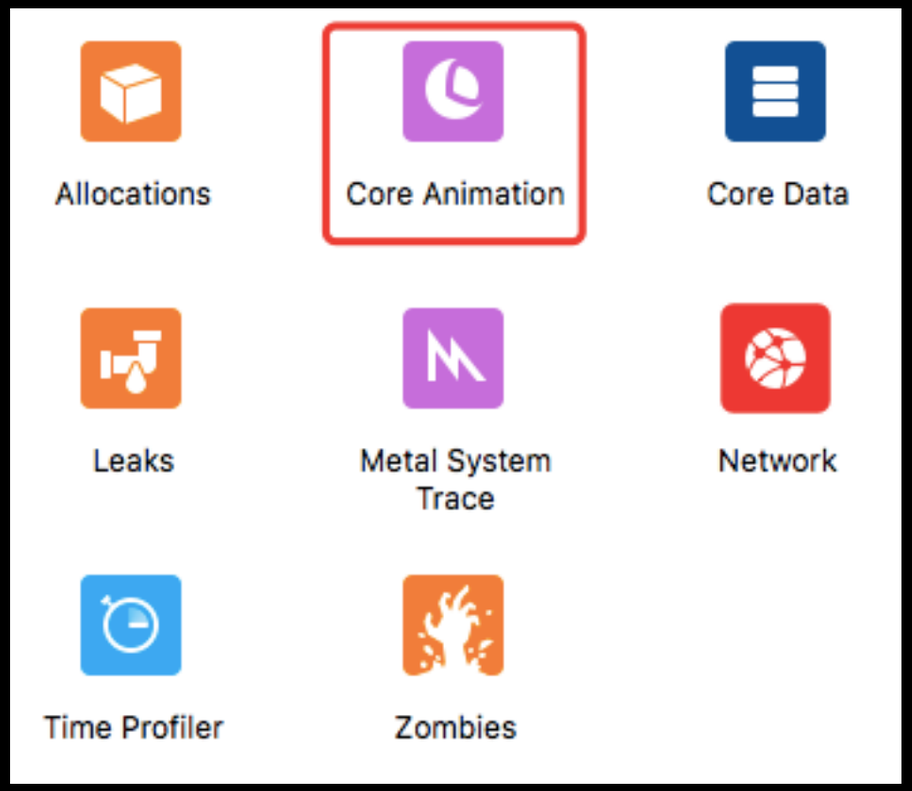
绘制过程中CPU/GPU受限

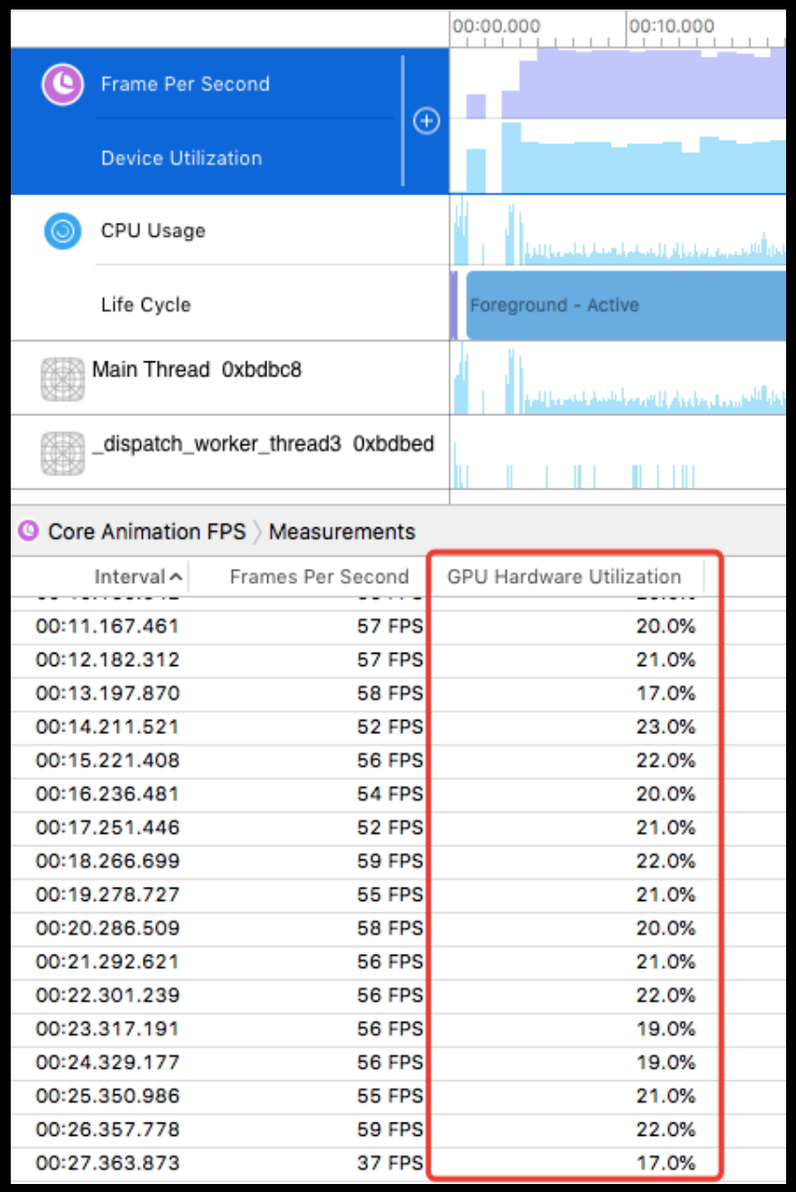
基于Tile的GPU渲染过程



绘制过程中CPU/GPU受限

如何使用Instrument查看GPU利用率





Interval	Frames Per Second	GPU Hardware Utilization
00:11.167.461	57 FPS	20.0%
00:12.182.312	57 FPS	21.0%
00:13.197.870	58 FPS	17.0%
00:14.211.521	52 FPS	23.0%
00:15.221.408	56 FPS	22.0%
00:16.236.481	54 FPS	20.0%
00:17.251.446	52 FPS	21.0%
00:18.266.699	59 FPS	22.0%
00:19.278.727	55 FPS	21.0%
00:20.286.509	58 FPS	20.0%
00:21.292.621	56 FPS	21.0%
00:22.301.239	56 FPS	22.0%
00:23.317.191	56 FPS	19.0%
00:24.329.177	56 FPS	19.0%
00:25.350.986	55 FPS	21.0%
00:26.357.778	59 FPS	22.0%
00:27.363.873	37 FPS	17.0%

绘制过程中CPU/GPU受限

总结

在开发过程中，我们需要知道应用的瓶颈是CPU受限还是GPU受限，为了流畅性和省电，我们当然希望CPU和GPU的利用率越低越好。

多余的CPU渲染工作

哪些工作会可能让CPU将你的界面变的“迟钝”?

- 对象创建
- 对象调整
- 对象销毁
- 布局计算
- Autolayout
- 文本计算&文本渲染
- 图片的解码及渲染大图
- 图像的绘制，例：-drawRect:和-drawLayer:inContext:

多余的CPU渲染工作

对象创建

- CPU会为对象的创建分配内存、调整属性，甚至还有I/O操作，这些工作都会消耗CPU的资源。

建议

- 如果不能减少对象的创建，尽量选择轻量级的对象
- 尽量推迟对象的创建时间，并把对象的创建分散到多个任务中
- 尽可能地复用对象

多余的CPU渲染工作

对象调整

- CPU会为对象的调整重新计算对象的相关属性，消耗的资源并不少。

建议

- 尽量减少不必要的属性修改
- 尽量避免调整视图层级、添加和移除视图

多余的CPU渲染工作

对象销毁

- 积少成多，大量的对象销毁会明显的消耗CPU的资源

建议

- 尽量将对象的销毁放在后台线程中

多余的CPU渲染工作

布局计算

- 太多的几何结构会导致CPU耗费大量资源去计算，在显示之前通过IPC发送到渲染服务的时候会引发CPU的瓶颈

建议

- 尽量提前计算好视图布局
- 尽量避免多次、频繁调整视图布局

多余的CPU渲染工作

Autolayout

- 随着视图数量和层级复杂度的增长，Autolayout带来的CPU消耗会呈指数级上升

建议

- 尽可能手动设置布局
- 使用AsyncDisplayKit等框架

多余的CPU渲染工作

文本计算&文本渲染

- 大量的文本宽高的计算会消耗不少的资源

建议

- 如果文本过多，可使用TextKit或CoreText对文本进行异步绘制，
  CoreText对象创建好后，能直接获取文本的宽高等信息

多余的CPU渲染工作

图片的解码及渲染大图

- 图片数据一般会在被提交到GPU前才会在主线程中进行解码操作
- 如果视图超出GPU支持的2048 * 2048 或4096 * 4096的尺寸的纹理，就必要要用CPU在图层每次显示之前对图片进行预处理

建议

- 如果文本过多，可使用TextKit或CoreText对本文进行异步绘制，CoreText对象创建好后，能直接获取文本的宽高等信息

多余的CPU渲染工作

图像的绘制

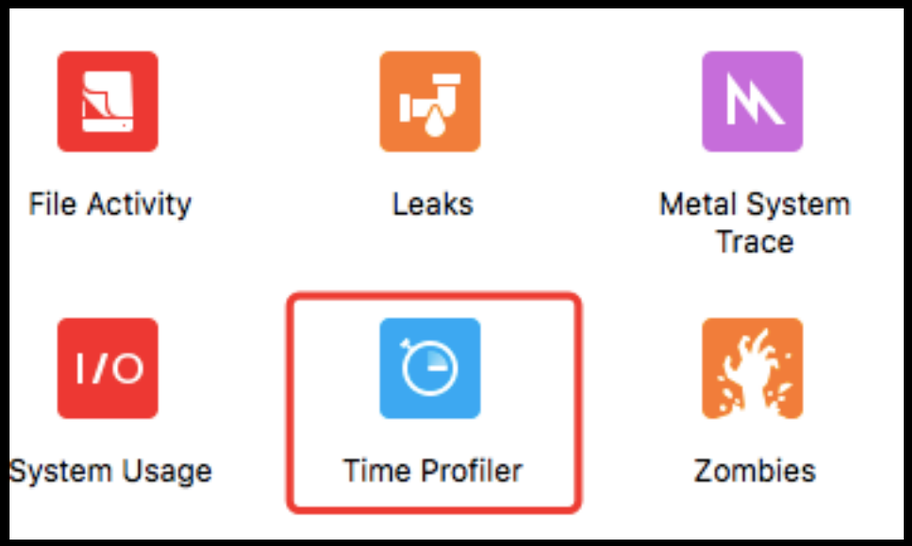
- 使用drawRect:或drawLayer: inContext:会触发离屏渲染，占用CPU大量的资源

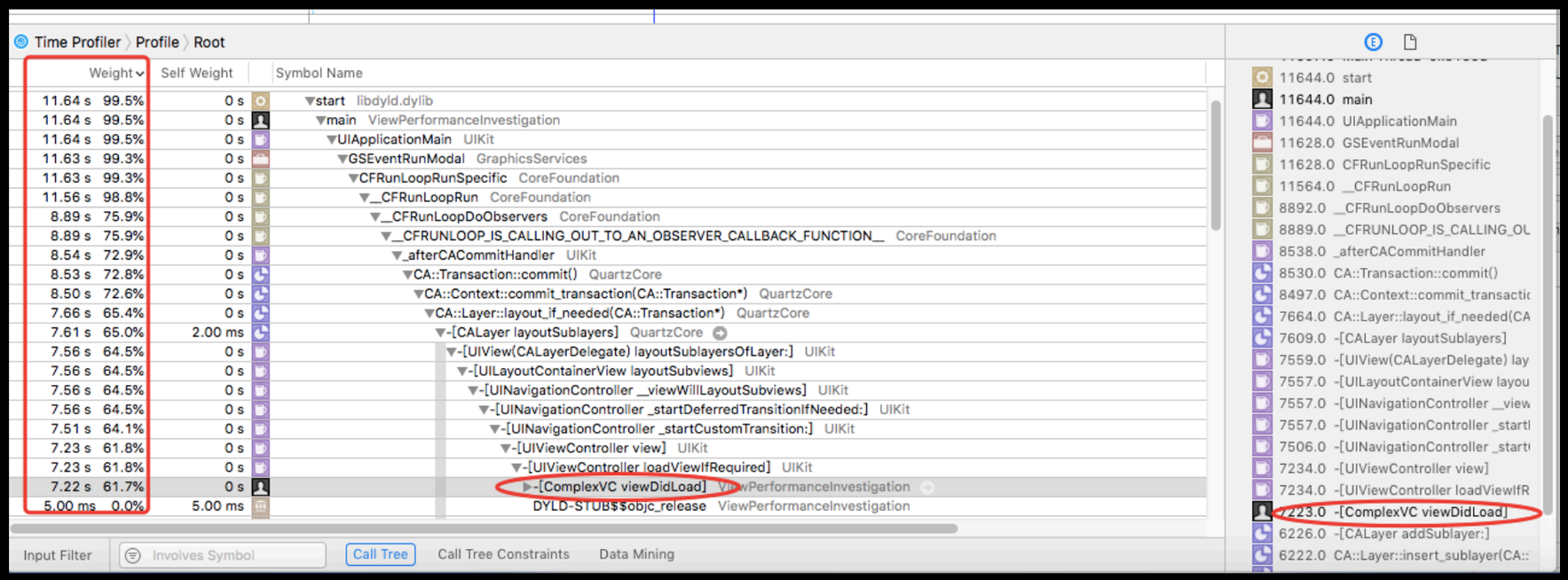
建议

- 尽可能少的使用CPU进行绘制工作，如果实在不行，可以放在后台线程中。（CoreGraphic是线程安全的）

多余的CPU渲染工作

如何使用Instrument查看CPU利用率





多余的CPU渲染工作

总结

- CPU“喜欢”静而不“喜欢”动
- CPU“喜欢”少而不“喜欢”多
- 在渲染工作中，GPU比CPU更专业

太多的离屏渲染

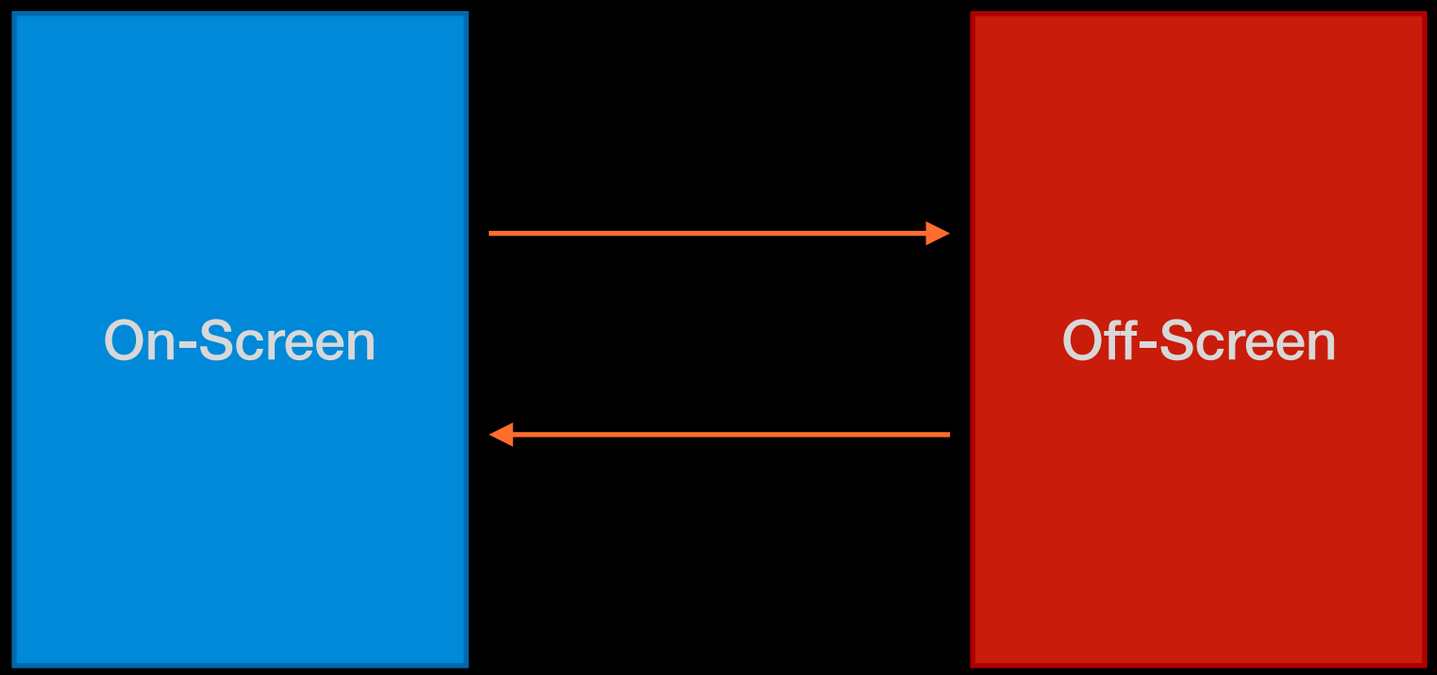
OpenGL中，GPU的渲染方式

- 屏幕渲染（On-Screen Rendering），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（Off-Screen Rendering），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。

太多的离屏渲染

离屏渲染的代价是昂贵的

- 创建新的缓冲区
- 上下文环境切换



太多的离屏渲染

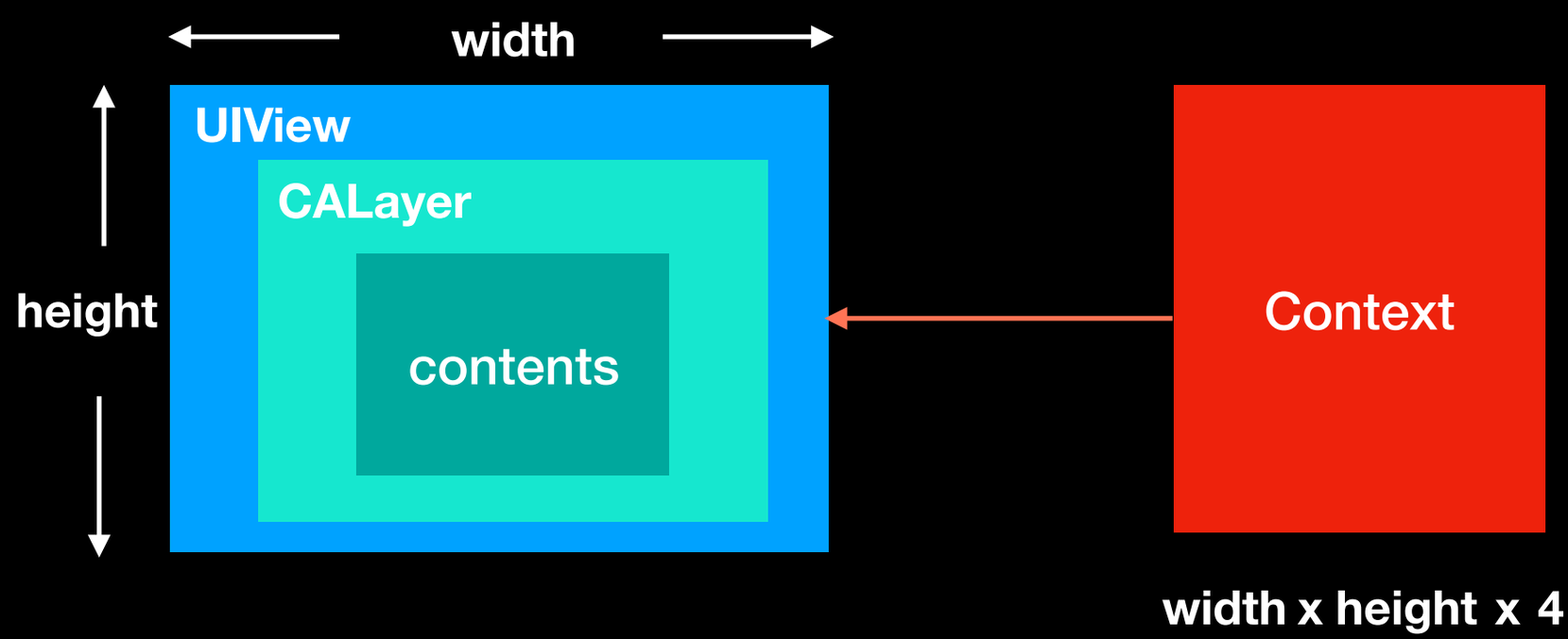
当设置以下属性时，会触发GPU的“离屏渲染”

- layer.mask（遮罩）
- layer.masksToBounds（截取）
- layer.shadow（阴影）
- layer.allowsGroupOpacity（不透明）
- layer.shouldRasterize（光栅化）

太多的离屏渲染

以下操作，会触发特殊的“离屏渲染”：CPU渲染

- -drawRect:
- -drawLayer: inContext:
- 使用Core Graphics的技术进行绘制操作



太多的离屏渲染

Mask（遮罩）和MasksToBounds（截取）

- 虽然MasksToBounds是使用Mask来实现的，不过使用Mask导致离屏渲染的代价比masksToBounds还要大。
- Mask无法取消离屏渲染。

建议

- 使用图片通过blending实现遮罩及圆角
- 重绘圆角

太多的离屏渲染

Shadow（阴影）

- 可以通过设定与视图边界相同的指定路径来避免layer.shadow产生离屏渲染。

建议

- 使用shadowPath来实现

太多的离屏渲染

Group Opacity（不透明）

- 离屏渲染触发条件：layer.opacity != 1.0，并且有子layer或者背景图。
- 开启 GroupOpacity 后，子 layer 在视觉上的透明度的上限是其父 layer 的opacity。

建议

- 如果不需要，尽量关闭allowsGroupOpacity。

太多的离屏渲染

Rasterization（光栅化）

- 使用GPU合成图像，缓存一次可多次使用
- 开启shouldRasterize后，CALayer会被光栅化
- 更新已光栅化的layer，会造成大量的离屏渲染
- 不要过度使用，系统限制了缓存的大小为2.5 x ScreenSize
- 被光栅化的图片如果超过100ms没有被使用，会被从缓存中移除

建议

- 当需要避免静态内容的复杂特效的重绘时，建议使用光栅化
- 当需要避免多个view嵌套的复杂层级视图的重绘时，建议使用光栅化

太多的离屏渲染

特殊的“离屏渲染” — CPU渲染

- 触发条件:重写drawRect或drawLayer: inContext:

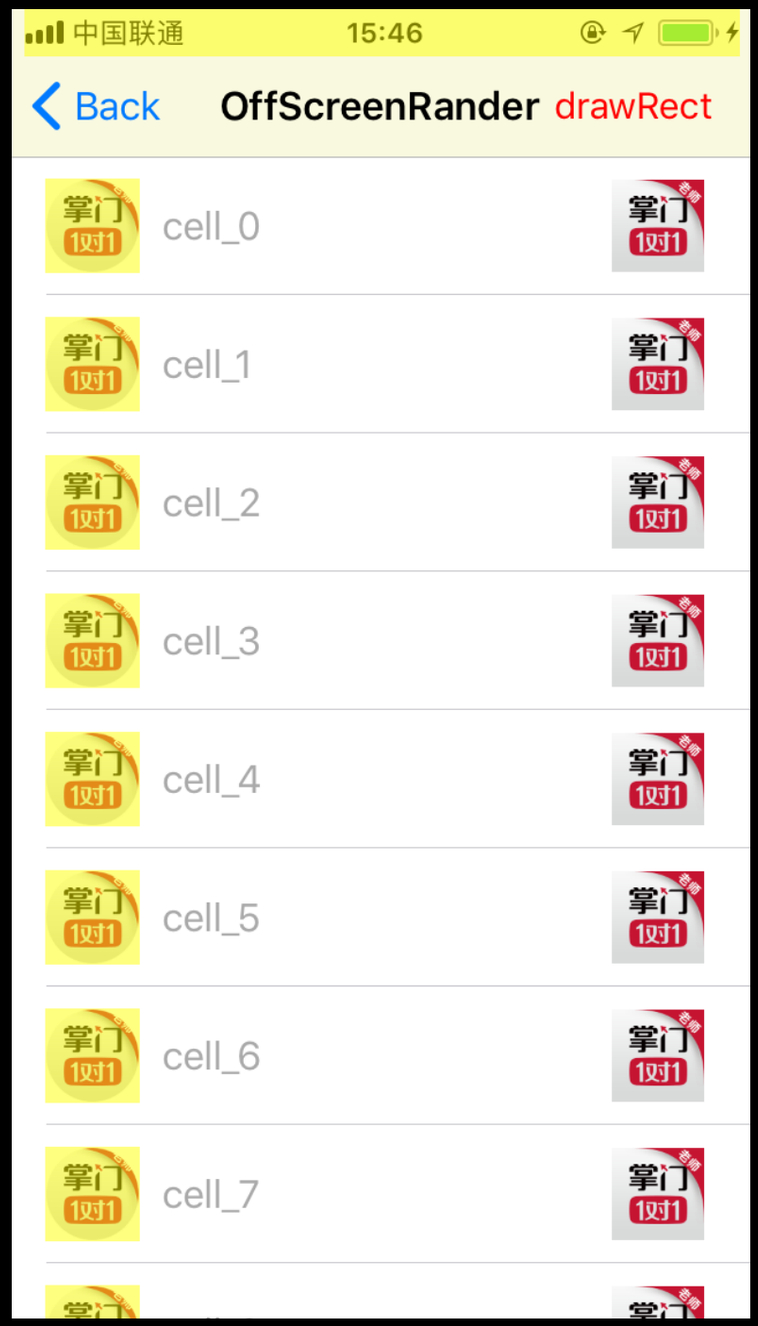
建议

- 最好的绘制就是不要绘制

太多的离屏渲染

如何检测离屏渲染?

- 真机：Xcode -> Debug -> View Debugging -> Rendering -> Color Offscreen-Rendered Yellow
- 模拟器：Simulator -> Debug -> Color Offscreen-Rendered
- 触发离屏渲染表现：黄色



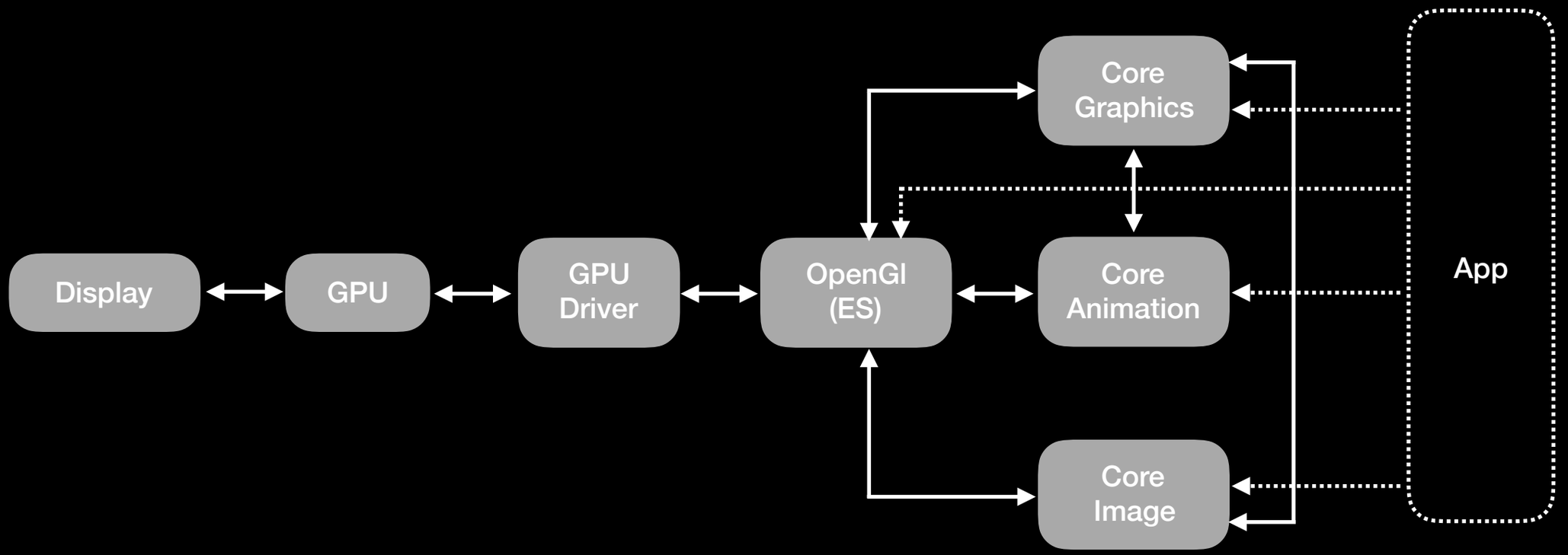
太多的离屏渲染

总结

- 尽量避免触发离屏渲染，如果必须使用，可以通过CAShapeLayer，layer.contents或者shadowPath来实现，可以较少地影响性能。

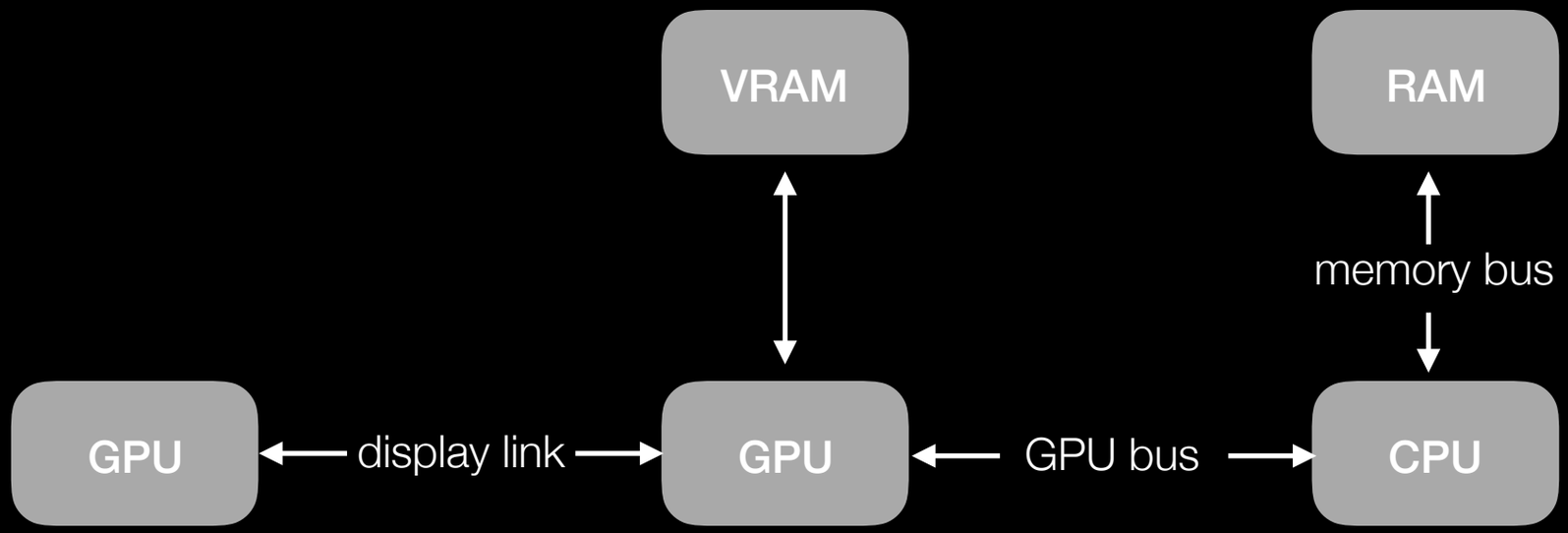
太多的图层混合

一个像素是如何绘制到屏幕上的?



太多的图层混合

一个像素是如何绘制到屏幕上的?



太多的图层混合

一个像素是如何绘制到屏幕上的?

- 屏幕上是像素是由红、绿、蓝三种颜色的组件构成的
- 在基于tile的渲染中，屏幕被分割成NxN个像素的titles
- 每个tile可以适配SoC(System on a chip)的缓存
- 几何信息划分进tile bucket中
- 光栅化在所有几何信息提交后才开始

太多的图层混合

当像素对齐的时候，像素混合的公式：

$$R = S + D * (1 - Sa)$$

混合色彩 = 源色彩(顶端纹理)+目标颜色(低一层的纹理)*(1-源颜色的透明度)

那么，什么时候像素不对齐?

- 缩放
- 当纹理的起点不在一个像素的边界上

扩展：

更多关于透明合成的基础公式可访问*: https://en.wikipedia.org/wiki/Alpha_compositing*

太多的图层混合

导致混合的原因

- layer(UIView)的Alpha < 1当纹理的起点不在一个像素的边界上
- UIImgaeView的image含有Alpha channel

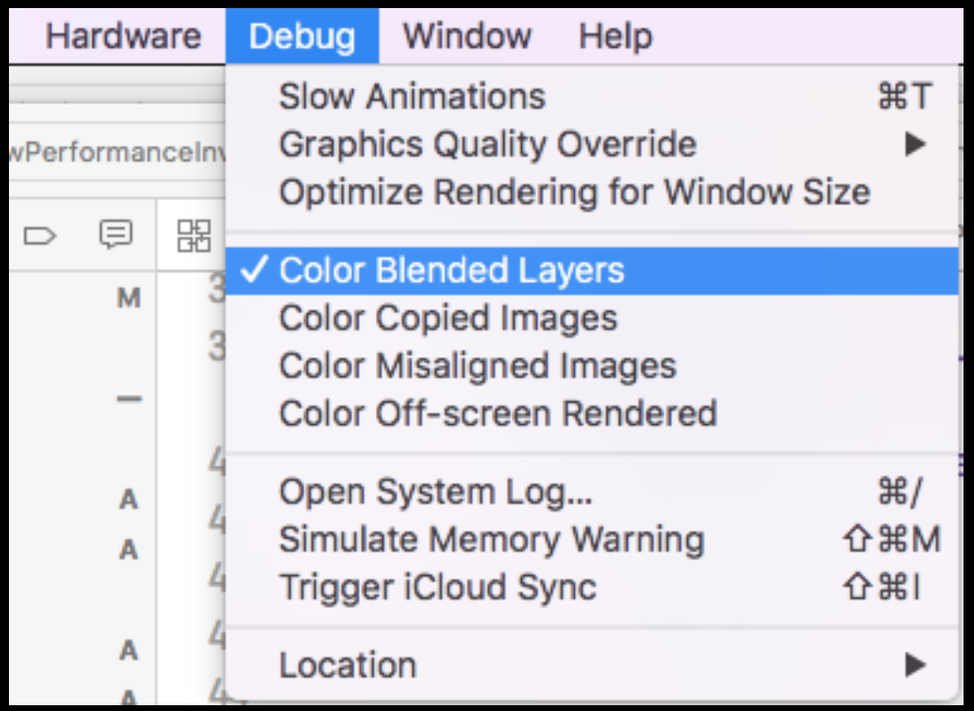
优化

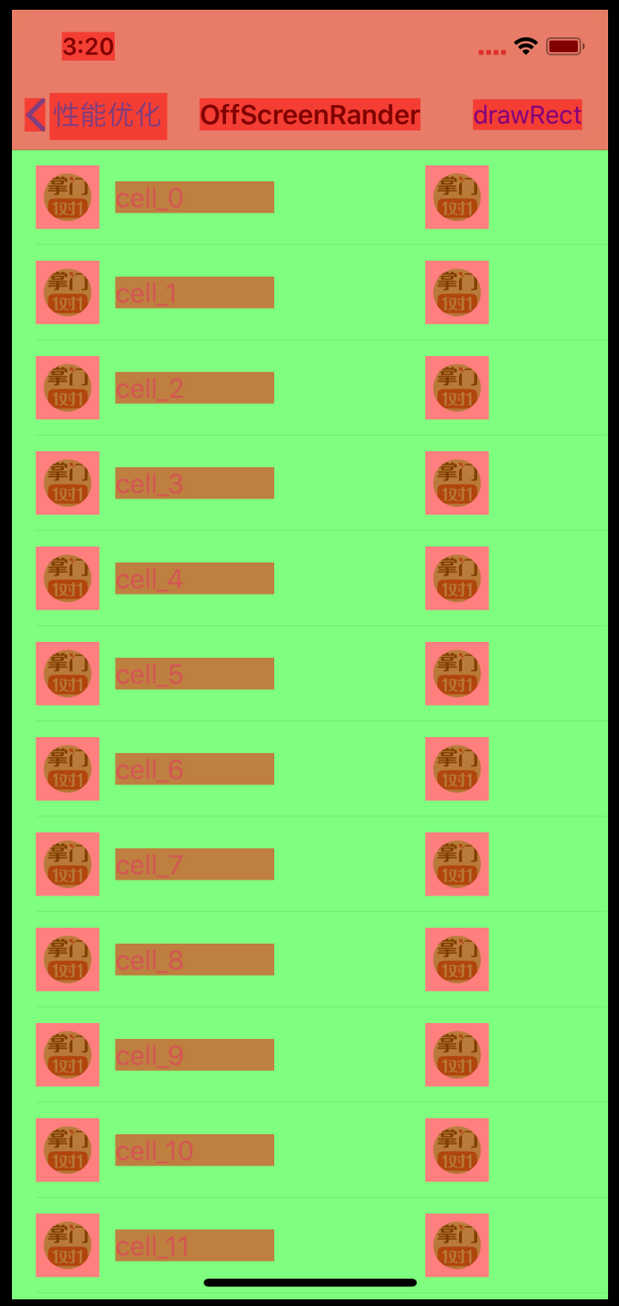
- 尽可能使用不透明图层，图片中去掉alpha通道

太多的图层混合

如何在Instrument中检测图层混合?

- 绿色：无像素混合
- 红色：有像素混合





奇怪的图片格式或尺寸大小

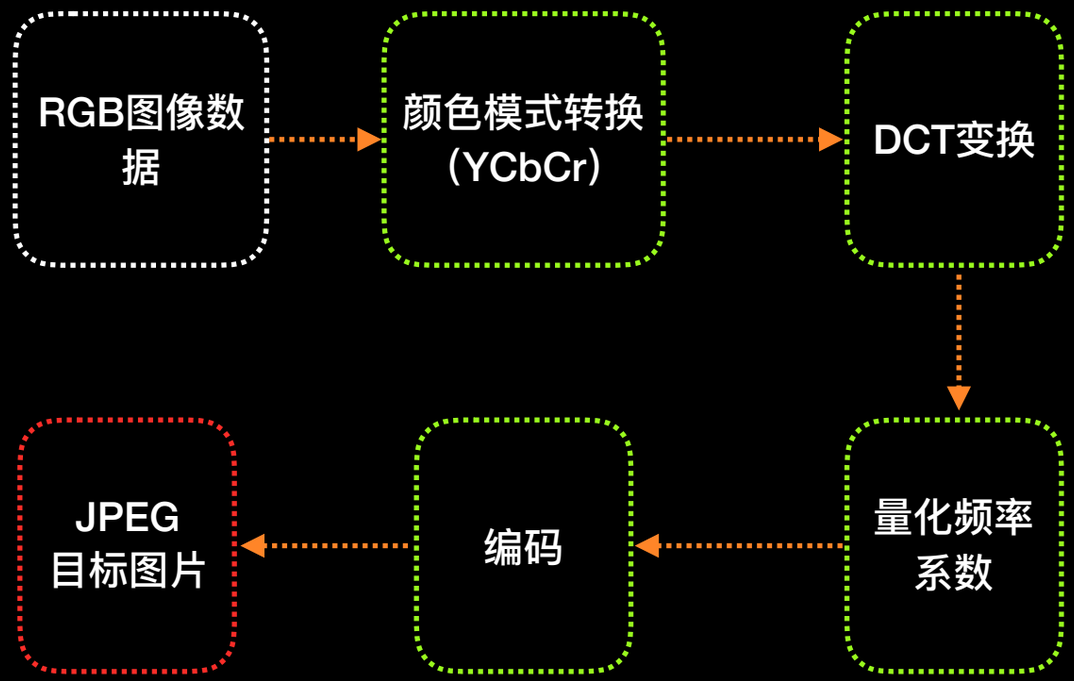
像素布局

32bits-per-pixel(bpp) & 8bits-per-componet(bpc)

奇怪的图片格式或尺寸大小

JPEG

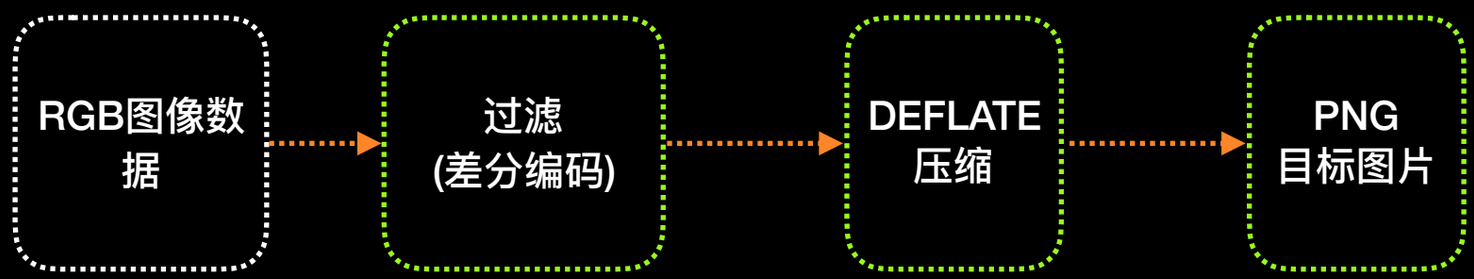
jpeg的压缩（有损）



奇怪的图片格式或尺寸大小

PNG

png的压缩（无损）



奇怪的图片格式或尺寸大小

PVRTC

- 利
  - ★ 不需提前解压便可直接绘制到屏幕上
  - ★ 加载的时候消耗更少的RAM
- 弊
  - ★ 文件尺寸比较大
  - ★ 必须是二维正方形，可强制拉伸或填充空白空间
  - ★ 质量并不是很好，尤其是透明图片
  - ★ 不能用Core Graphics绘制，也不能在普通的ImageView中展示，也不能直接做为CALayer的内容。必须要用作OpenGL纹理加载PVRTC图片映射到一对三角板来在CAEAGLLayer或者GLKView中显示。
  - ★ 创建一个OpenGl纹理来绘制PVRTC图片的开销相当昂贵，除了把所有图片都绘制到一个相同的上下文。
  - ★ PVRTC使用了一个不对称的压缩算法，压缩过程相当漫长。

扩展：

$:texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-4 -o target.pvrtc source.png

GLView: *https://github.com/nicklockwood/GLView*

奇怪的图片格式或尺寸大小

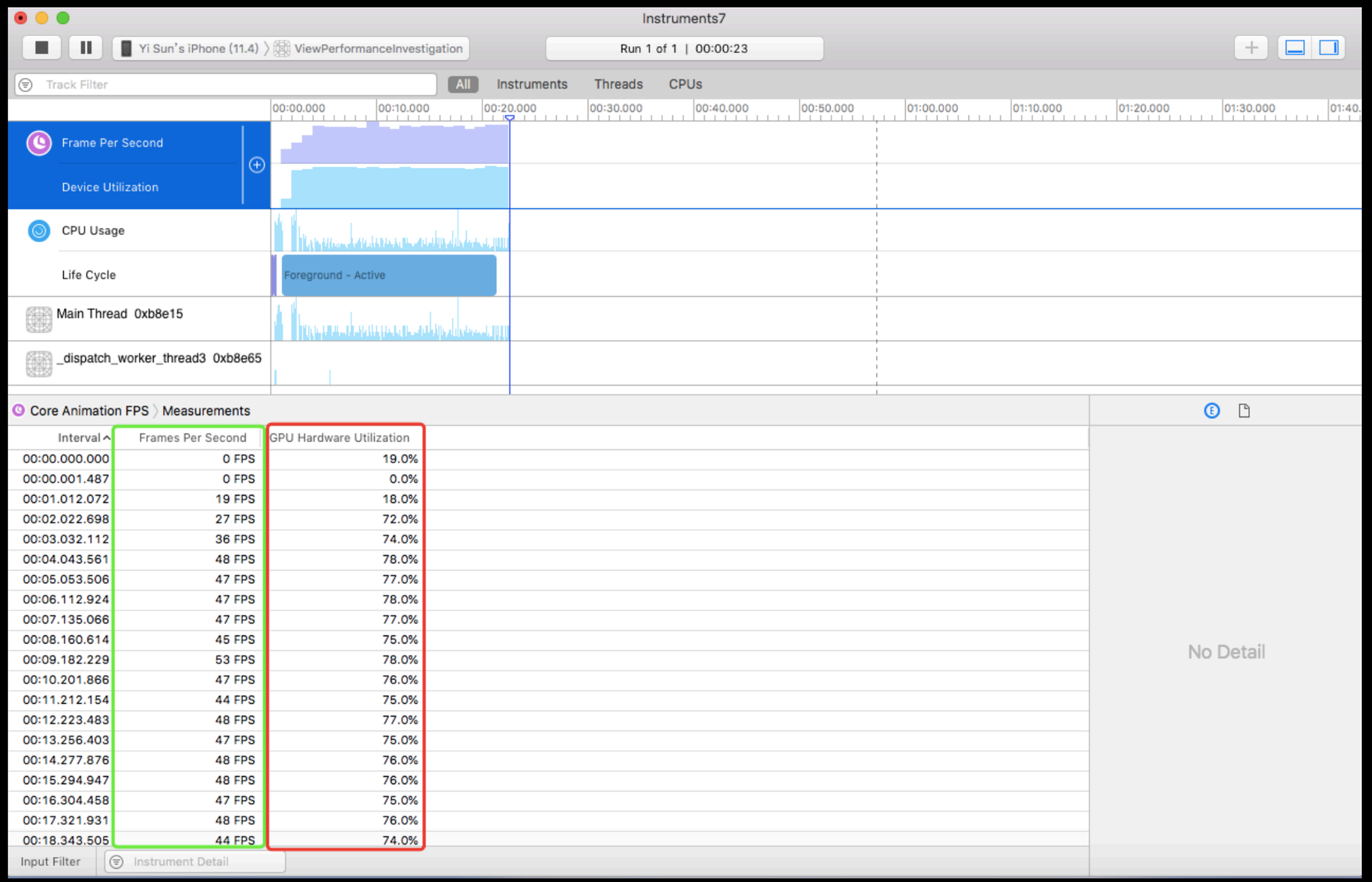
合适的尺寸和大小是一个不错的选择

- 图片的缩放也会占用一定的时间
- GPU不支持的颜色格式图片，CPU需要进行格式转换

代价昂贵的视图或动画

合理地使用性能消耗大的视图或动画效果

e.g. UIVisualEffectView with UIBlurEffect



Interval	Frames Per Second	GPU Hardware Utilization
00:00.000.000	0 FPS	19.0%
00:00.001.487	0 FPS	0.0%
00:01.012.072	19 FPS	18.0%
00:02.022.698	27 FPS	72.0%
00:03.032.112	36 FPS	74.0%
00:04.043.561	48 FPS	78.0%
00:05.053.506	47 FPS	77.0%
00:06.112.924	47 FPS	78.0%
00:07.135.066	47 FPS	77.0%
00:08.160.614	45 FPS	75.0%
00:09.182.229	53 FPS	78.0%
00:10.201.866	47 FPS	76.0%
00:11.212.154	44 FPS	75.0%
00:12.223.483	48 FPS	77.0%
00:13.256.403	47 FPS	75.0%
00:14.277.876	48 FPS	76.0%
00:15.294.947	48 FPS	76.0%
00:16.304.458	47 FPS	75.0%
00:17.321.931	48 FPS	76.0%
00:18.343.505	44 FPS	74.0%

较复杂的视图层级

太多的图层会引起CPU的瓶颈

- 初始化图层，处理图层、打包通过IPC发给渲染引擎，转化成OpenGL几何图形，都占用资源开销
- 确切的限制数量取决于iOS设备，图层类型，图层内容和属性等

问题与补充