to drawRect or not to drawRect (when should one use drawRect/Core Graphics vs subviews/images and why?)

To clarify the purpose of this question: I know HOW to create complicated views with both subviews and using drawRect. I'm trying to fully understand the when's and why's to use one over the other.

I also understand that it doesn't make sense to optimize that much ahead of time, and do something the more difficult way before doing any profiling. Consider that I'm comfortable with both methods, and now really want a deeper understanding.

A lot of my confusion comes from learning how to make table view scroll performance really smooth and fast. Of course the original source of this method is from the author behind twitter for iPhone (formerly tweetie). **Basically it says that to make table scrolling buttery smooth, the secret is to NOT use subviews, but instead do all the drawing in one custom uiview.** Essentially it seems that using lots of subviews slows rendering down because they have lots of overhead, and are constantly re-composited over their parent views.

To be fair, this was written when the 3GS was pretty brand spankin new, and iDevices have gotten much faster since then. Still this method is regularly suggested on the interwebs and elsewhere for high performance tables. In fact it's a suggested method in Apple's Table Sample Code, has been suggested in several WWDC videos (Practical Drawing for iOS Developers), and many iOS programming books.

There are even awesome looking tools to design graphics and generate Core Graphics code for them.

So at first I'm lead to believe "there's a reason why Core Graphics exists. It's FAST!"

But as soon as I think I get the idea "Favor Core Graphics when possible", I start seeing that drawRect is often responsible for poor responsiveness in an app, is extremely expensive memory wise, and really taxes the CPU. Basically, that I should "Avoid overriding drawRect" (WWDC 2012 iOS App Performance: Graphics and Animations)

So I guess, like everything, it's complicated. Maybe you can help myself and others understand the When's and Why's for using drawRect?

I see a couple obvious situations to use Core Graphics:

1. You have dynamic data (Apple's Stock Chart example)
2. You have a flexible UI element that can't be executed with a simple resizable image
3. You are creating a dynamic graphic, that once rendered is used in multiple places

I see situations to avoid Core Graphics:

1. Properties of your view need to be animated separately
2. You have a relatively small view hierarchy, so any perceived extra effort using CG isn't worth the gain
3. You want to update pieces of the view without redrawing the whole thing
4. The layout of your subviews needs to update when the parent view size changes

So bestow your knowledge. In what situations do you reach for drawRect/Core Graphics (that could also be accomplished with subviews)? What factors lead you to that decision? How/Why is drawing in one custom view recommended for buttery smooth table cell scrolling, yet Apple advises drawRect against for performance reasons in general? What about simple background images (when do you create them with CG vs using a resizable png image)?

A deep understanding of this subject may not be needed to make worthwhile apps, but I don't love choosing between techniques without being able to explain why. My brain gets mad at me.

Question Update

Thanks for the information everyone. Some clarifying questions here:

1. If you are drawing something with core graphics, but can accomplish the same thing with UIImageViews and a pre-rendered png, should you always go that route?
2. A similar question: Especially with badass tools like this, when should you consider drawing interface elements in core graphics? (Probably when the display of your element is variable. e.g. a button with 20 different color variations. Any other cases?)
3. Given my understanding in my answer below, could the same performance gains for a table cell possibly be gained by effectively

capturing a snapshot bitmap of your cell after your complex UIView render's itself, and displaying that while scrolling and hiding your complex view? Obviously some pieces would have to be worked out. Just an interesting thought I had.

iphone ios core-graphics

edited Feb 3 '13 at 0:55

asked Feb 2 '13 at 7:03

Bob Spryn

9,712 5 44 73

4 Answers

Stick to UIKit and subviews whenever you can. You can be more productive, and take advantage of all the OO mechanisms that should things easier to maintain. Use Core Graphics when you can't get the performance you need out of UIKit, or you know trying to hack together drawing effects in UIKit would be more complicated.

The general workflow should be to build the tableviews with subviews. Use Instruments to measure the frame rate on the oldest hardware your app will support. If you can't get 60fps, drop down to CoreGraphics. When you've done this for a while, you get a sense for when UIKit is probably a waste of time.

So, why is Core Graphics fast?

CoreGraphics isn't really fast. If it's being used all the time, you're probably going slow. It's a rich drawing API, which requires its work be done on the CPU, as opposed to a lot of UIKit work that is offloaded to the GPU. If you had to animate a ball moving across the screen, it would be a terrible idea to call setNeedsDisplay on a view 60 times per second. So, if you have sub-components of your view that need to be individually animated, each component should be a separate layer.

The other problem is that when you don't do custom drawing with drawRect, UIKit can optimize stock views so drawRect is a no-op, or it can take shortcuts with compositing. When you override drawRect, UIKit has to take the slow path because it has no idea what you're doing.

These two problems can be outweighed by benefits in the case of table view cells. After drawRect is called when a view first appears on screen, the contents are cached, and the scrolling is a simple translation performed by the GPU. Because you're dealing with a single view, rather than a complex hierarchy, UIKit's drawRect optimizations become less important. So the bottleneck becomes how much you can optimize your Core Graphics drawing.

Whenever you can, use UIKit. Do the simplest implementation that works. Profile. When there's an incentive, optimize.

answered Feb 2 '13 at 18:32

Ben Sandofsky

491 4 2

I'm going to try and keep a summary of what I'm extrapolating from other's answers here, and ask clarifying questions in an update to the original question. But I encourage others to keep answers coming and vote up those who have provided good information.

General Approach

It's quite clear that the general approach, as Ben Sandofsky mentioned in his answer, should be **"Whenever you can, use UIKit. Do the simplest implementation that works. Profile. When there's an incentive, optimize."**

The Why

1. There are two main possible bottlenecks in an iDevice, **the CPU and GPU**
2. CPU is responsible for the initial drawing/rendering of a view

3. GPU is responsible for a majority of animation (Core Animation), layer effects, compositing, etc.
4. UIView has a lot of optimizations, caching, etc, built in for handling complex view hierarchies
5. When overriding drawRect you miss out on a lot of the benefits UIView's provide, and it's generally slower than letting UIView handle the rendering.

Drawing cells contents in one flat UIView can greatly improve your FPS on scrolling tables.

Like I said above, CPU and GPU are two possible bottlenecks. Since they generally handle different things, you have to pay attention to which bottleneck you are running up against. **In the case of scrolling tables, it's not that Core Graphics is drawing faster, and that's why it can greatly improve your FPS.**

In fact, Core Graphics may very well be slower than a nested UIView hierarchy for the initial render. However, it seems the typical reason for choppy scrolling is you are bottlenecking the GPU, so you need to address that.

Why overriding drawRect (using core graphics) can help table scrolling:

From what I understand, the GPU is not responsible for the initial rendering of the views, but is instead handed textures, or bitmaps, sometimes with some layer properties, after they have been rendered. It is then responsible for compositing the bitmaps, rendering all those layer affects, and the majority of animation (Core Animation).

In the case of table view cells, the GPU can be bottlenecked with complex view hierarchies, because instead of animating one bitmap, it is animating the parent view, and doing subview layout calculations, rendering layer effects, and compositing all the subviews. So instead of animating one bitmap, it is responsible for the relationship of bunch of bitmaps, and how they interact, for the same pixel area.

So in summary, the reason drawing your cell in one view with core graphics can speed up your table scrolling is NOT because it's drawing faster, but because it is reducing the load on the GPU, which is the bottleneck giving you trouble in that particular scenario.

edited Dec 2 '13 at 6:23

answered Feb 3 '13 at 0:43

Bob Spryn
9,712 5 44 73

1 Careful though. GPUs effectively re-composite the whole layer tree for each frame. So moving a layer is effectively zero cost. If you create one large layer, only moving that one layer is faster than moving several. Moving something *inside* this layer suddenly involves the CPU and has a cost. Since cell contents usually don't change much (only in response to comparatively rare user actions), using fewer views speeds up things. But making one large view with all cells would be a Bad Idea™. – uliwitness Oct 26 '14 at 2:22

The difference is that UIView and CALayer essentially deal in fixed images. These images are uploaded to the graphics card (if you know OpenGL, think of an image as a texture, and a UIView/CALayer as a polygon showing such a texture). Once an image is on the GPU, it can be drawn very quickly, and even several times, and (with a slight performance penalty) even with varying levels of alpha transparency on top of other images.

CoreGraphics/Quartz is an API for *generating* images. It takes a pixel buffer (again, think OpenGL texture) and changes individual pixels inside it. This all happens in RAM and on the CPU, and only once Quartz is done, does the image get "flushed" back to the GPU. This round-trip of getting an image from the GPU, changing it, then uploading the whole image (or at least a comparatively large chunk of it) back to the GPU is rather slow. Also, the actual drawing that Quartz does, while really fast for what you are doing, is way slower than what the GPU does.

That's obvious, considering the GPU is mostly moving around unchanged pixels in big chunks. Quartz does random-access of pixels and shares the CPU with networking, audio etc. Also, if you have several elements that you draw using Quartz at the same time, you have to re-draw all of them when one changes, then upload the whole chunk, while if you change one image and then let UIViews or CALayers paste it onto your other images, you can get away with uploading much smaller amounts of data to the GPU.

When you don't implement -drawRect:, most views can just be optimized away. They don't contain any pixels, so can't draw anything. Other views, like UIImageView, only draw a UIImage (which, again, is essentially a reference to a texture, which has probably already been loaded onto the GPU). So if you draw the same UIImage 5 times using a UIImageView, it

is only uploaded to the GPU once, and then drawn to the display in 5 different locations, saving us time and CPU.

When you implement -drawRect:, this causes a new image to be created. You then draw into that on the CPU using Quartz. If you draw a UIImage in your drawRect, it likely downloads the image from the GPU, copies it into the image you're drawing to, and once you're done, uploads this *second copy* of the image back to the graphics card. So you're using twice the GPU memory on the device.

So the fastest way to draw is usually to keep static content separated from changing content (in separate UIViews/UIView subclasses/CALayers). Load static content as a UIImage and draw it using a UIImageView and put content generated dynamically at runtime in a drawRect. If you have content that gets drawn repeatedly, but by itself doesn't change (I.e. 3 icons that get shown in the same slot to indicate some status) use UIImageView as well.

One caveat: There is such a thing as having too many UIViews. Particularly transparent areas take a bigger toll on the GPU to draw, because they need to be mixed with other pixels behind them when displayed. This is why you can mark a UIView as "opaque", to indicate to the GPU that it can just obliterate everything behind that image.

If you have content that is generated dynamically at runtime but stays the same for the duration of the application's lifetime (e.g. a label containing the user name) it may actually make sense to just draw the whole thing once using Quartz, with the text, the button border etc., as part of the background. But that's usually an optimization that's not needed unless the Instruments app tells you differently.

edited Oct 26 '14 at 2:36

answered Apr 10 '14 at 10:23

uliwitness

3,113 12 28

Thanks for the detailed answer. Hopefully more people scroll down and vote this up as well. – Bob Spryn Apr 21 '14 at 2:35

Wish I could upvote this twice. Thanks for the great writeup! – Sea Coast of Tibet Jan 30 at 13:22

I am a game developer, and I was asking the same questions when my friend told me that my UIImageView based view hierarchy was going to slow down my game and make it terrible. I then proceeded to research everything I could find about whether to use UIViews, CoreGraphics, OpenGL or something 3rd party like Cocos2D. The consistent answer I got from friends, teachers, and Apple engineers at WWDC was that there won't be much of a difference in the end because *at some level they are all doing the same thing*. Higher-level options like UIViews rely on the lower level options like CoreGraphics and OpenGL, just they are wrapped in code to make it easier for you to use.

Don't use CoreGraphics if you are just going to end up re-writing the UIView. However, you can gain some speed from using CoreGraphics, as long as you do all your drawing in one view, but is it really *worth* it? The answer I have found is usually no. When I first started my game, I was working with the iPhone 3G. As my game grew in complexity, I began to see some lag, but with the newer devices it was completely unnoticeable. Now I have plenty of action going on, and the only lag seems to be a drop in 1-3 fps when playing in the most complex level on an iPhone 4.

Still I decided to use Instruments to find the functions that were taking up the most time. I found that the problems *were not related to my use of UIViews*. Instead, it was repeatedly calling CGRectMake for certain collision sensing calculations and loading image and audio files separately for certain classes that use the same images, rather than having them draw from one central storage class.

So in the end, you might be able to achieve a slight gain from using CoreGraphics, but usually it will not be worth it or may not have any effect at all. The only time I use CoreGraphics is when drawing geometric shapes rather than text and images.

answered Feb 2 '13 at 7:52

WolfLink

2,137 1 14 31

5 I think you may be confusing Core Animation with Core Graphics. Core Graphics is really slow, software based drawing and is not used for drawing regular UIViews unless you override the drawRect method. Core Animation is hardware accelerated, fast and responsible for most of what you see on screen. – Nick Lockwood Sep 11 '13 at 21:58