A Design Pattern for High-Performance Scrolling of Table Views on iOS Using Layout Models

Jason Howlin and Shamal Nikam
July 27, 2018

Abstract

On iOS devices, smooth scrolling lists of content, such as a stream of news articles, is essential to a delightful user experience.

This paper outlines a design pattern for highly optimized and efficient rendering of content in UITableViews and UICollectionViews. We look at how to model layouts as data independent of actual view objects, distribute layout-related tasks across multiple threads, and how to pre-calculate and cache layouts to minimize the work needed at render time.

To support our approach, we'll look at sample code and compare measurements against less-efficient approaches, such as Auto Layout. Our inspiration comes from working on apps like Newsroom, Aol App, and Alto that have long lists of variable height items to display and striving to improve scrolling performance.

Introduction: The Problem

Typical approaches for building lists of content include using Auto Layout or using dummy measuring views to get the heights needed by table view. This requires a lot of layout calculations on main thread, consuming rendering bandwidth and impacting scrolling performance.

Importance

UITableViews and UICollectionViews are the most popular approaches to show long lists of content like news articles, mails, etc, and are an inherent part of all apps. Users expect a smooth scrolling experience to quickly skim through lots of details. So the expectation from app developers is to maintain a 60 FPS (Frames Per Second) rendering speed.

Challenges When Displaying Lists

UITableView and UICollectionView both inherit from UIScrollView, a class which displays content larger than the size of the view itself. In a list of 1000 items, perhaps 5 can fit on a screen at any given time, yet the scroll view needs to know the total content size for all 1000 items.

The scroll view needs the height information to display a set of scroll bars, that indicate the amount of content the scroll view has to display, and the precise location of the user while they are scrolling through the list.

Therefore, one of the earliest tasks to perform before rendering a table view is to calculate the height of each item. And when the height of each item is variable and a function of the content it is displaying, such as the number of lines needed to display a news headline, you need to visit each item in the list and calculate the required height for that item - and the sum of all items equals the total content size.

Common Approach

Auto Layout is a mechanism provided by Apple that calculates size and position of views based on a set of supplied constraints and the contents of a UI element. It is a neat approach and works well with smaller lists where we don't have content with dynamic

height. We can still tweak it to use with the dynamic heights, but that involves creating a “dummy” instance of your view offscreen, configuring it for your data, and then getting the height of the view. Here we are ‘rendering’ content using the main thread that is not visible to the user, just to calculate the height.

We can definitely improve the experience by caching the heights but we still do all this on main thread, where all the UI rendering happens.

UITableView offers an API that works with Auto Layout to calculate variable height items, but our experience has shown it to be slow, as well as visually buggy when adding and removing items in the list.

Our Approach of Using Layout Models

In our approach we propose offloading this work to a background thread, and caching the layout and view models.

A *Layout Model* is simply a representation of your view independent of the actual UI elements, like labels, images, and buttons. This representation consists of all the rectangles representing the position and size of each subview.

View Models

Views display information to the users. The information comes from data model objects inside your application, typically consisting of raw, unformatted types.

The most common approach is to pass the model object to the view, which takes the model values it’s interested in and applies formatting, such as fonts, colors, a localized formatted date, and others. Some values may not be present on an individual model, which means we have to hide or reposition views.

An established but less commonly-used approach is to build an object that represents a formatted view of your data for a particular view, called a *View Model*. The views become more easily configurable and reusable with a view model. To reuse, one can simply map their model to a view model, an operation that keeps the view object agnostic to a particular model object type.

There is a cost involved with applying formatting - the NSAttributedString class performs internal synchronization when applying fonts and colors that has overhead. Storing these values in a view model allows us to cache this information and perform the operation only once.

Here is an example of a View Model:

View Model
NSAttributed Strings for UILabels / UITextViews. Formatted date strings. Image URLs. View background / foreground colors. View shadow / border information. Icon image names (heart, reply)

Layout Models

The layout model contains all of the geometry related information - primarily structs defining the size and position of each user interface element, such as labels, images, and buttons, sized to fit the values in the view model.

Not only does the object store these values, but also contains a function to calculate

them. To do so, it needs information such as a constrained dimension, usually an available width to work in. Also a number of constants around the metrics of the view - the spacing between objects, the margins, and the fixed size of some items, such as images.

What remains are the variable height items, such as labels, that are a function of the formatted text they display.The most important part of this process is calculating the size of the formatted text to be displayed in UILabel or UITextView.

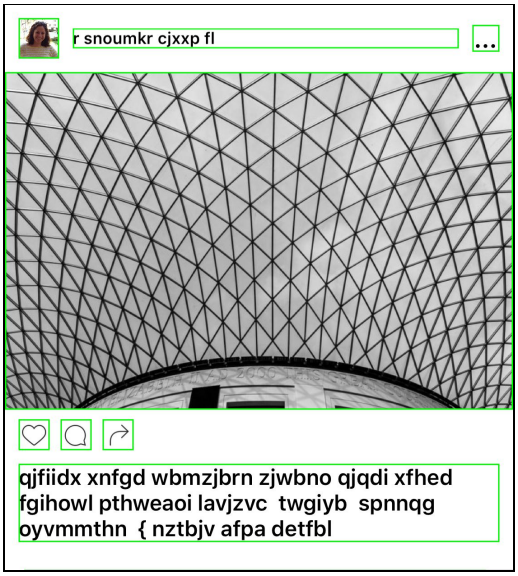
The system frameworks provide a number of ways to calculate the height required to draw text without actually drawing the text. These include methods on NSAttributedString and within TextKit framework.

What's nice about these methods are they do not need to be performed on the main thread. It allows us to offload this work to a background thread and keep the main thread free, which is where all of our stutters occur.

And finally, the layout model will also provide the overall height of the view. That way it is easily accessible to the tableview delegates that need to know the height of every item in the list.

Once all the calculations are done, the frames can be set in the 'layoutSubviews' method as needed, such as when a cell appears on screen. It is also worth noting that at this point we can determine if the frame has changed and set it only if the existing view frame is different than the one provided by the layout model.

In the image below, green rectangles show the frames calculated by the Layout Model.



Layout Model
Frames for all UI elements. Margins / Padding. Constant Heights. Size of the entire View.
Function that accepts width and calculates frames for all subviews.

Offloading Layout Calculation to Background Thread

It is quite common that displaying a list of content requires some asynchronous work - whether it's making a network request to fetch items or querying from a database.

The UI for a list of content is typically built to display a 'Loading' state of some kind, such as a spinner. To give the user an experience of infinite scrolling, apps usually show a spinner to indicate they are getting more data as well.

The common approach is to perform this fetching off of the main thread, and once the data is ready for the UI, bounce back and

set the new data. But with the new data, the scroll view's height will change. And so the heights need to be recalculated all over again, which will happen on the main thread.

If you recall, we've previously established that heights can all be calculated off the main thread, so we can add this calculation (the creation of our view models and layout models) as *another step in our asynchronous data retrieval process*. Not only can it be performed asynchronously, the individual calculations can be performed in parallel, as each item's calculation has no dependency on another. Caching the results and then bouncing back to the main thread allows the updated height calculation process to be a very fast cache lookup.

Downloading and Rendering Images in Table Views

Any discussion of scrolling performance would be incomplete without addressing the downloading and rendering of images. To stand a chance of achieving 60 FPS scrolling, best practices must be followed to asynchronously download images over the network *and* decode and resize them in the background.

We've included our custom implementation of a high-performance image downloading system in our sample code.

It also supports working with UITableView's prefetch API to fetch images about to appear at a lower priority, as well as cancelling fetches.

It's important that this be implemented efficiently, as an incorrect implementation uses main thread CPU time, which would have an impact on any layout-related work.
https://developer.apple.com/videos/play/wwdc2018/219/

Related Work

UICollectionViewLayout

Some of Apple's own frameworks have taken a similar approach of separating the layout code from the actual view rendering.

The 'UICollectionViewLayout' expects an implementation to provide objects that represent the position and size of all cells in the collection view before any rendering. These objects, the 'UICollectionViewLayoutAttributes' class, have a one-to-one relationship with each cell. It has just a few simple properties, most importantly a frame.

The typical approach when creating your own layout is to calculate this information, and then cache them in the layout class for use when it comes time to render the cell on screen.

TextKit Framework

For any given collection of text, which contains the string to render as well as the font, size, paragraph spacing, and a width constraint, TextKit can provide a series of frames that each glyph will be rendered in, without any actual view to render in.

These calculations can safely be performed off the main thread.

On the left is the information that TextKit's NSLayoutManager provides as a collection of frames independent of a label (we've colored them in to

illustrate their position). On the right is a UILabel laid on top - you can see how the frames match perfectly.

Texture Framework (formerly AsyncDisplayKit)

Facebook and Pinterest have worked on the Texture framework, which focuses on rendering off the main thread for smooth scrolling. Their sample code works well, but it introduces heavy dependencies: it requires that all views inherit from a base class, it is a large and complex framework that could be difficult to debug, has a steep learning curve to understand how things work together, and puts your application at their mercy for fixing bugs or updating it to support new iOS versions.

Our approach and these have similarities only in the sense that they take advantage of doing work asynchronously. There are only so many UI-related tasks that can be performed async - namely image downloading and decompression, and text measurement.

The other component of Texture is the notion of performing work for cell rendering lazily, as cells are about to appear on screen. However, this was before Apple provided public API to be notified of upcoming cells to be rendered. So the ability to prefetch image downloads for cells appearing on screen can be performed without the Texture framework.

Benchmarks and Comparisons

Text only view measurements on iPhone 6 - iOS 12

{jiy bo rae zljw njtnd icjxwu k yztxp rntqz a{sm

	Auto Layout	Our Approach
Time to dequeue and configure a UITableViewCell	4.5 ms	0.9 ms
FPS with very fast scrolling	57	60
Time to rotate (avg value)	480 ms	480 ms

A more complex view measurements on iPhone 6 - iOS 12

	Auto Layout	Our Approach
Time to dequeue and configure a UITableViewCell	12.5 ms	2.8 ms
FPS with very fast scrolling	57	60
Time to rotate (avg value)	600 ms	550 ms

Considering on iOS that a frame needs to render every 8 to 16 ms, being able to return a cell in 3 ms vs. 12 ms is valuable. Dropping even a single frame will cause a stutter. We consistently achieved a solid 60 fps under very fast scrolling, versus 57 fps using the Auto Layout approach.

Drawbacks

Care must be taken to ensure the layout object stays in sync with both the view model, and the higher-level layout environment, such as the device orientation or the constraints of a parent view.

Any time these values change, they may impact the layout object. If label frames have been calculated with a fixed width of the device in a portrait orientation, the cached values will no longer be valid when device is rotated.

To resolve this, a higher level object, such as a view controller, needs to invalidate and recalculate layout objects on bounds change. Resetting the layout object on a view, if there are changes, should invalidate the existing layout and trigger a repositioning of views.

Another potential issue is the methods used to measure text are unaware of a specific UI element. A 'UILabel' may provide some additional margins and insets which need to be factored in when measuring text.

Conclusion

IOS provides a very efficient rendering system. In most cases, abstracting the layout is overkill - it doesn't solve performance issue that needs solving. In fact, it increases complexity.

However, when dealing with large lists containing views of varying heights and sizes, this approach attempts to do the required work the least amount of times, and with a number of optimizations.
Our sample code provides a complete implementation of our approach versus using Auto Layout. It also includes our image downloading implementation.

Pattern Summary

1. Asynchronously create a view model for each cell that contains the formatted data to display, such as 'NSAttributedStrings', formatted dates, colors. Cache this model as to calculate only once.

2. Asynchronously create a view layout object that, given a view model and constrained width, measures text, uses margins and spacing, and provides position and sizes for each subview in your cell. Cache this model as to calculate only once.

3. Perform the above steps in a background queue as an additional step in fetching your content from the network or database (which is already asynchronous).

4. For each cell about to be rendered, apply the values from these two objects if changed - set attributed text on labels and set frames of views.

5. Use the UITableView prefetching API to begin / cancel image downloading, and be sure your image work decompresses and resizes images properly.

The above steps are a focused approach when high performance is necessary.

Future Work

Further additions would be, building a complex hierarchy of views to see at what point Auto Layout performs poorly and compare it against our layout approach. Introduce our design pattern in a real world complex application that performs a wide variety of operations like, handling push notification, authentication updates, showing live user presence while scrolling, and measure how this approach performs.

Addendum

References

1. Sample Implementation of Layout Model Approach vs. Auto Layout: https://git.ouroath.com/jhowlin/LayoutModelFeed.git
2. WWDC 2018 Image Best Practices: https://developer.apple.com/videos/wwdc2018/
3. UICollectionViewLayout: https://developer.apple.com/documentation/uikit/uicollectionviewlayout
4. TextKit https://developer.apple.com/documentation/appkit/textkit
5. Texture https://github.com/TextureGroup/Texture