FAKE IT ‘TIL YOU MAKE IT: TIPS AND TRICKS FOR IMPROVING INTERFACE RESPONSIVENESS

Why do some native applications seem so fast while others do not? There is an old adage in auto racing. “Speed is money, how much do you want to spend?” It doesn’t take long for iPhone programmers to rub up against a similar problem, one perhaps expressed as “Speed is time, how much do you have left to spend before release?” Given the limitations of processor power, RAM, and network bandwidth, not to mention battery drain, writing iPhone applications that display lots of data is hard. Clever caching, prefetching of data, and optimized drawing is the key to removing the variable response times that make an app consuming nonlocal or large amounts of data seem slow to the user.

How can you avoid a "death by a thousand paper cuts" user experience when you have a lot of data to display? Most of the applications that Apple ships on the iPhone access network services and many of them deal with large data sets. Mail pulls and caches potentially large amounts of data from your mail server, the Maps application loads tiles from Google maps, and the weather application requests the latest weather on demand; even the calendar and contacts applications can sync with data stored on servers hosted by Microsoft, Google, Yahoo, and Apple. Many well-reviewed third party applications also pull large quantities of data from the cloud in one way or another. Facebook, Pandora, AIM, Yahoo Instant Messenger, and many others have developed offerings that are robust and responsive. Writing an application for the iPhone that displays large amounts of potentially nonlocal data is not easy. You've probably experienced an application that seems to start and stop working depending on your network connection or how much information you’ve loaded. Users of native iPhone applications have different expectations with regard to interface responsiveness than they do when browsing the web. It's difficult to satisfy a user who tolerates multiple page loads while using a browser, but who may not tolerate a slow-scrolling table view or a view that takes a few seconds to download data and render in a native application.

In this chapter we're going to build two projects. The first project starts out as an app that displays historical AAPL stock information from Yahoo.com and graphs closing price over time similar to Apple's own stocks application. As we add functionality, we'll discuss some strategies as well as some of the trade-offs involved with various methods of caching information from remote data sources. By the time we’re done, the application will cache and update the stock prices of several stocks while remaining usable and responsive to the user. The second project deals with the calculation and display of large amounts of information in a scroll view that is generated and drawn programatically. In this project, we’ll solve some common performance and user experience problems related to drawing large amounts of data.

Plotting of historical stock prices with AAPLot

Let’s start with a simple application that charts the last few months of Apple, inc. stock prices. <<<LINK TO THE SAMPLE CODE: `01AAPLPlotFirstPlot`>>>

AAPLot uses a simple web service from Yahoo.com to download historical stock data in comma - separated format. Type the following URL into a web browser http://ichart.yahoo.com/

table.csv?s=AAPL&a=3&b=19&c=2009&d=6&e=12&f=2009&g=d&ignore=.csv. You should see text that looks something like this.

```
Date,Open,High,Low,Close,Volume,Adj Close

2009-06-18,136.11,138.00,135.59,135.88,15237600,135.88
2009-06-17,136.67,137.45,134.53,135.58,20377100,135.58
2009-06-16,136.66,138.47,136.10,136.35,18255100,136.35
2009-06-15,136.01,136.93,134.89,136.09,19276800,136.09
2009-06-12,138.81,139.10,136.04,136.97,20098500,136.97
2009-06-11,139.55,141.56,138.55,139.95,18719300,139.95
2009-06-10,142.28,142.35,138.30,140.25,24593700,140.25
2009-06-09,143.81,144.56,140.55,142.72,24152500,142.72
2009-06-08,143.82,144.23,139.43,143.85,33255400,143.85
```

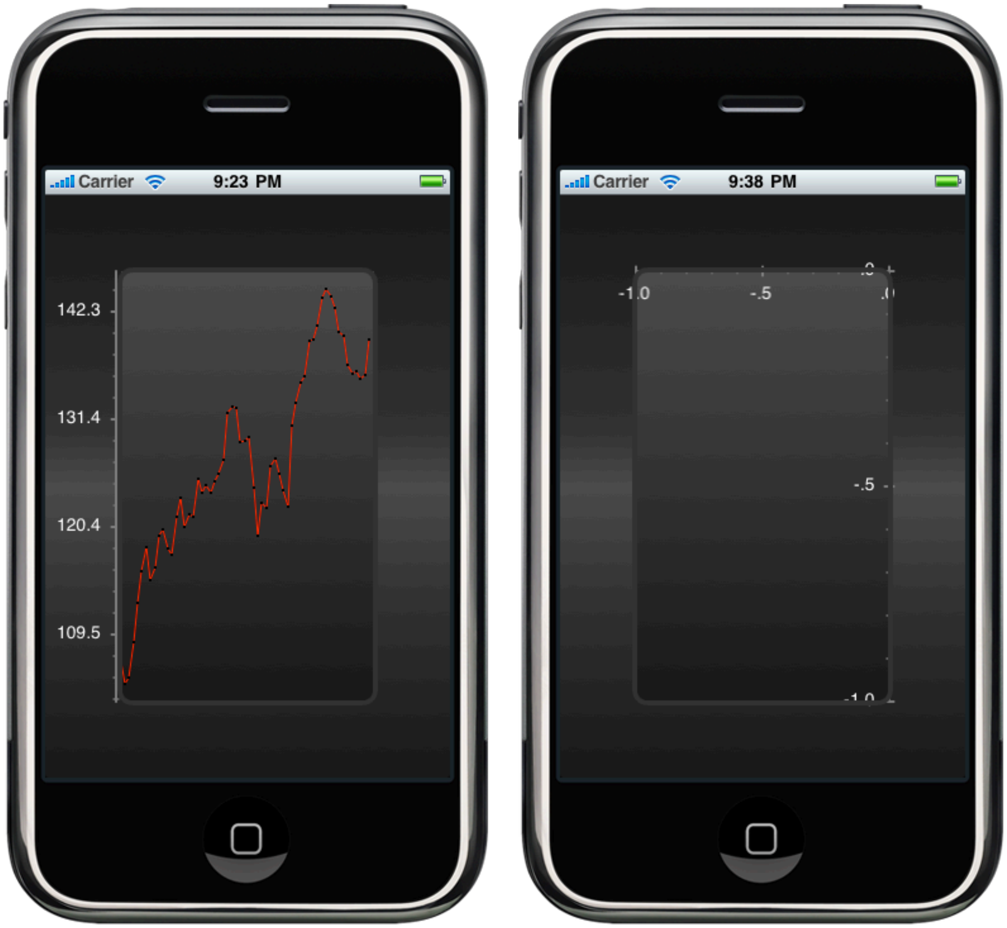
Most of the work for AAPLot is concentrated in two objects, APYahooDataPuller, which downloads, parses, and stores the data from Yahoo.com, and AAPLotViewController, which displays the data in a plot. The method from *APYahooDataPuller* that constructs a URL with a target start and end date is below. On application launch, the *AAPLotViewController* creates an *APYahooDataPuller* instance. It downloads and parses the csv data and then calls the *APYahooDataPullerDelegate* method `dataPullerDidFinishFetch:` of the *AAPLotViewController*. The view controller then draws a plot into a layer of its view.

APYahooDataPuller.m URL string construction

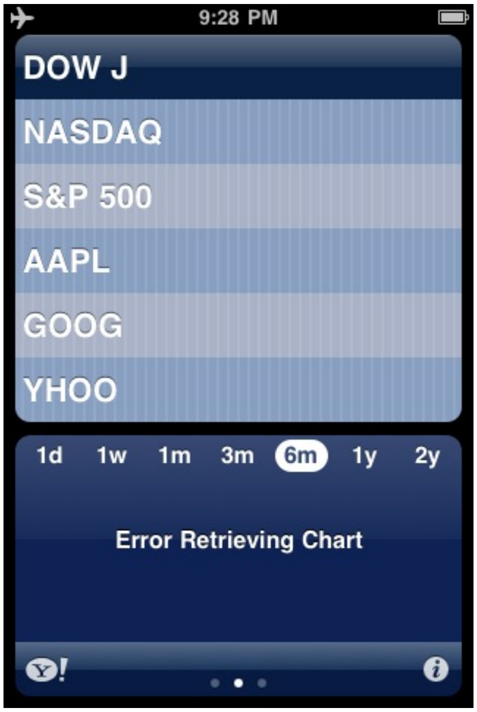
```
-(NSString *)URL;
{

    unsigned int unitFlags = NSMonthCalendarUnit | NSDayCalendarUnit | NSYearCalendarUnit;

    NSCalendar *gregorian = [[NSCalendar alloc] \
                         initWithCalendarIdentifier:NSGregorianCalendar];

    NSDateComponents *compsStart = [gregorian components:unitFlags fromDate:targetStartDate];
    NSDateComponents *compsEnd = [gregorian components:unitFlags fromDate:targetEndDate];

    [gregorian release];

    NSString *url = [NSString stringWithFormat:@"http://ichart.yahoo.com/table.csv?s=%@&”, \
                                                                 [self targetSymbol]];
    url = [url stringByAppendingFormat:@”a=%d&”, [compsStart month]-1];
    url = [url stringByAppendingFormat:@”b=%d&”, [compsStart day]];
    url = [url stringByAppendingFormat:@”c=%d&”, [compsStart year]];

    url = [url stringByAppendingFormat:@”d=%d&”, [compsEnd month]-1];
    url = [url stringByAppendingFormat:@”e=%d&”, [compsEnd day]];
    url = [url stringByAppendingFormat:@”f=%d&”, [compsEnd year]];
    url = [url stringByAppendingString:@”g=d&”];

    url = [url stringByAppendingString:@”ignore=.csv”];
    url = [url stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    return url;
}
```

We'll be using quite a lot of free and open source code in the examples for this chapter, all of which have licenses that allow for redistribution and commercial use. The plotting library used in AAPLot is from core-plot. It's an impressive new project by a group of developers interested in graphing, charting and plotting for the iPhone and the Mac. During WWDC 2009, Apple sponsored a code-a-thon to jumpstart its development. One of its stated goals is to maintain a tight integration with Apple's core technologies like Core Animation, Core Data, and Cocoa bindings. You can read more and download the latest code at http://code.google.com/p/core-plot/.

Build and run the AAPLot example. Depending on whether you have an internet connection, you should see something that looks like one of the two images below.



It's already a modestly useful application. We might add some text to warn the user if there was a problem while trying to retrieve the graph from the internet, we could also remove the empty graph from the UI when there isn't a connection, call it a day, and release. You certainly wouldn't be the first to be tempted to do that.



Storing data between runs

A simple, and often big, usability win is to cache any downloaded information to disk and present that data to the user as a placeholder before attempting to download any new data. Assuming the user has downloaded the data they wants to see at some point in the past, they can see the version that your application downloaded last time it was run. Also, if your application is host to a type of data that the user would normally want to read while offline, perhaps a document reader like Amazon's amazing Kindle application, it's a great plus to store anything the user has viewed. With data that can get stale fairly quickly, like stock prices, it is still better to show the user something rather than nothing while perhaps signaling an unobtrusive way that the data is a little stale.

To add caching logic to the AAPlot application, we will add a mechanism to save to and load from disk a given set of financial data. On launch, we'll show the cached data, then attempt to download new data. If we are able to get new data, we'll compare it to our old data and we'll overwrite the old data and update the UI only if it's stale.

With the iPhone's NAND flash memory, writing is expensive both in terms of speed and in terms of hardware lifetime. It will eventually wear out with use. Apple recommends that you write to disk only when necessary. Since our application checks to see if the data is stale, it is unlikely to download stock data more than once or twice per day so we can reasonably store it to disk when it arrives. If our data were more often malleable, we might consider storing it only when the application closes or if we were run out of memory. Apple supplies a convenience method in your application's delegate where you can save data before the app closes:

-applicationWillTerminate:

using plists to persist data

Since we are already using an array of *NSDictionary*s to store our data within the *APYahooDataPuller*, it is trivial to persist it because an *NSDictionary* or an *NSArray* can be written to disk as a property list as long as it contains only property list objects (instances of *NSData*, *NSDate*, *NSNumber*, *NSString*, *NSArray*, or *NSDictionary*). The *NSDecimalNumber*s we are using are subclasses of *NSNumber*, so we can store those with one caveat: they're going to get converted to floating point first, which will reduce their precision. For demonstration purposes we'll just round them when reading them back in. The precision we lose might cause a graph line to move by a pixel, which isn't a big deal for this application. Let's add some caching methods to *APYahooDataPuller*.

First we'll add a method called `plistRep` that returns a dictionary representation of the *APYahooDataPuller*'s data. Then we'll add a method that writes that dictionary to a file, calling the *NSDictionary* `writeToFile:atomically:` method. We should also take this opportunity to further modify *APYahooDataPuller* to better model our new strategy. Since we are caching the startDate and endDate values to disk and will need them for comparison later, we will want to add a few instance variables to track the dates we want from the server and also the symbol we're looking for, which may be different from those we're loading from the cache, and change our designated initializer accordingly. We should also change the behavior with respect to notifying our delegate. Since we are caching financial data, it's possible that our target startDate, endDate, and symbol will match that which is already cached. If that is the case, we won't need to reload the graph and we should probably not even notify our delegate. We'll change the interface with our delegate so that we only notify when the financial data changes as a result of a fetch.

APYahooDataPuller.m persistence methods

```
- (NSDictionary *)plistRep
{
    NSMutableDictionary *rep = [NSMutableDictionary dictionaryWithCapacity:7];
    [rep setObject:[self symbol] forKey:@"symbol"];
    [rep setObject:[self startDate] forKey:@"startDate"];
    [rep setObject:[self endDate] forKey:@"endDate"];
    [rep setObject:[self overallHigh] forKey:@"overallHigh"];
    [rep setObject:[self overallLow] forKey:@"overallLow"];
    [rep setObject:[self financialData] forKey:@"financalData"];
    return [NSDictionary dictionaryWithDictionary:rep];
}

- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag;
{
    NSLog(@"writeToFile:%@", path);
    BOOL success = [[self plistRep] writeToFile:path atomically:flag];
    return success;
}
```

Where are we saving our data to, exactly? <<TODO: Picture of the directory structure?>>

When your application is installed on the iPhone or the iPhone simulator, its application bundle includes a sandboxed subdirectory for storing user data. You can get the path to that bundle like this:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

We append AAPL.plist to the path in *documentsDirectory* when we store the plist data file.

Rename the dataPullerDidFinishFetch delegate method in AAPLotViewController.m

Now we need to modify the AAPLotViewController to use our new delegate method.

Replace this:

```
-(void)dataPullerDidFinishFetch:(APYahooDataPuller *)dataPuller;
```

With the more accurately named:

```
-(void)dataPullerFinancialDataDidChange:(APYahooDataPuller *)dataPuller;
```

Build and run the application while connected to the internet. It should look about the same as before. Now disable your internet connection and run the application again. The graph should render just as it did before using the data that was cached to disk on the first run. If your application stores more critical data, perhaps business documents, your users will appreciate having their content available to them anywhere. <<<Examples/02AAPLPlotCachePlist>>>

> If the graph does not draw when you run the application without an internet connection, you're likely re-installing the application and overwriting the Documents folder with an empty one each time you install it. Instead of running the app using Xcode's build and run, try running the application by touching or clicking on it on the phone or in the simulator without reinstalling.

Shipping AAPLot with placeholder data

You never get a second chance to make a first impression. If a user downloads your application on the App store and then finds himself without an internet connection the first time they use it, having nothing to look at can be disappointing. That user may never run your application again. Many applications would benefit from having some kind of default local data, even if it is just something to show the user what it will look like when they are able to get fresh data. To ship a default version of the AAPL.plist with our application, we will first need to retrieve one from the simulator.

iPhone simulator spelunking

The iPhone simulator loads its library of applications and data from your home directory in ~/*Library/Application Support/iPhone Simulator/User/Applications/.* Each application is housed in a directory named with a UUID. The easiest way to find our AAPL.plist is to empty this directory, build and run our application, and then retrieve it from the newly created directory. Open a

Terminal and issue the following command; be aware that this will delete all applications from the simulator and all of their user data:

```
rm -rfv ~/Library/Application\ Support/iPhone\ Simulator/User/Applications/*
```

Making sure your internet connection is live, build and run the application. You'll find the AAPL. plist in the *~/Library/Application\ Support/iPhone\ Simulator/User/Applications/SOMELONGUUID/ Documents/* directory. Copy it into the AAPLot code directory. Now add it as a resource in Xcode. *Reference Type:* can be set to *Default*. Make sure that *Add To Target* is also checked so Xcode knows to copy it during the build. <<<PICTURE SHOWING THE ABOVE>>>

Now we need to write a method that checks to see if AAPL.plist is in the Documents directory and, if it is not, we should instead load the plot from the application's resources folder.

```
-(NSString *)faultTolerantPathForSymbol:(NSString *)aSymbol
{
    NSString *docPath = [self pathForSymbol:aSymbol];;
    if (![[NSFileManager defaultManager] fileExistsAtPath:docPath]) {
        //if there isn't one in the user's documents directory, see if we ship with this data
        docPath = [[[NSBundle mainBundle] resourcePath] \
                        stringByAppendingPathComponent:[NSString stringWithFormat:@"%@.plist", aSymbol]];
    }
    return docPath;
}

-(NSDictionary *)dictionaryForSymbol:(NSString *)aSymbol
{
    NSString *path = [self faultTolerantPathForSymbol:aSymbol];
    NSMutableDictionary *localPlistDict = [NSMutableDictionary dictionaryWithContentsOfFile:path];
    return localPlistDict;
}
```

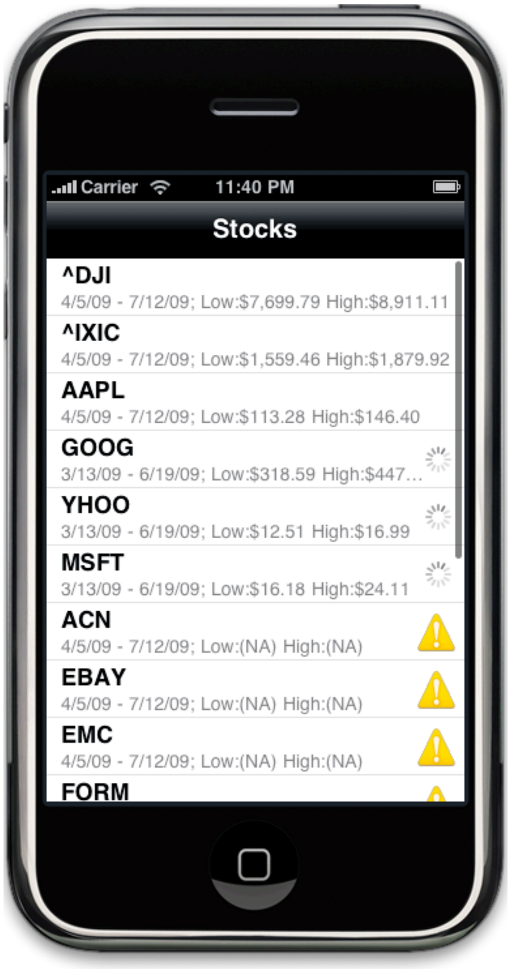
Once again in Terminal remove all applications from the simulator so we can see how the application behaves as it will when it is used for the first time:

```
rm -rfv ~/Library/Application\ Support/iPhone\ Simulator/User/Applications/*
```

Now disable your internet connection again. Build and run. Our default AAPL.plist should load even though the application is freshly installed with no previously-fetched data. The version of AAPLot that includes all of these caching changes can be found at <<<Examples/ 03AAPLPlotDefaultData>>>

> In a shipping application, indicating to the user that the data they're seeing is stale and warning them that the application would really benefit from an internet connection is a good idea. See Apple's Reachability sample code for information on how to test for the availability of a server on the internet. See also the Human Interface Guidelines for the iPhone... <<TODO: LINKS TO Reachability SAMPLE CODE AND HIG>>

Extending the app for multiple stock graphs: StockPlot



^DJI
4/5/09 - 7/12/09; Low:$7,699.79 High:$8,911.11
^IXIC
4/5/09 - 7/12/09; Low:$1,559.46 High:$1,879.92
AAPL
4/5/09 - 7/12/09; Low:$113.28 High:$146.40
GOOG
3/13/09 - 6/19/09; Low:$318.59 High:$447…
YHOO
3/13/09 - 6/19/09; Low:$12.51 High:$16.99
MSFT
3/13/09 - 6/19/09; Low:$16.18 High:$24.11
ACN
4/5/09 - 7/12/09; Low:(NA) High:(NA)
EBAY
4/5/09 - 7/12/09; Low:(NA) High:(NA)
EMC
4/5/09 - 7/12/09; Low:(NA) High:(NA)
FORM

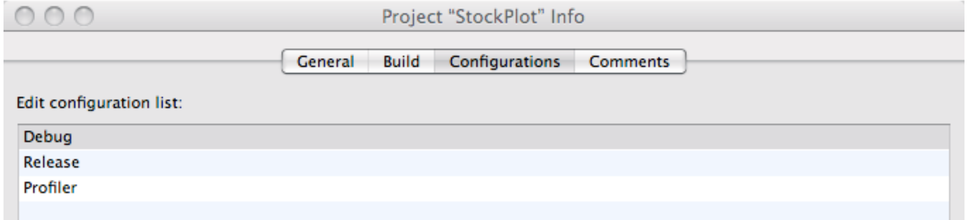
Now we are going to reuse some of the objects from the AAPLot application in an app called StockPlot that loads a whole bunch of stocks into a table that the user can select to push a graph onto the screen. Things get rather more complicated when there is a lot of data to download and store. Our earlier strategy of download, then parse, then cache, then display from the AAPLot application might not hold-up when we try it with a lot of stocks at the same time. Let's also see what happens when we try to load graphs in response to user input. StockPlot is in <<<Examples/04StockPlotConcurrentDownloads>>>

StockPlot will ship with Yahoo, Microsoft, Google, and Apple stock data and will attempt to download a little more than a dozen other technology companies' prices on launch. The *RootViewController* of the project handles table view loading and *APYahooDataPullerDelegate* duties. It loads a summary of whatever data it can find in its array of *APYahooDataPuller* objects, which it creates at launch, each of which act just like they did in AAPLot by loading from disk, downloading, and notifying of changed data. The *RootViewController* object also has a small amount of code to limit the number of concurrent downloads to 3 connections at a time. Build and run it on the simulator. If you're online (and you don't blink), you will see the little exclamation point cautionary icons in the table cells replaced by progress indicators while the corresponding *APYahooDataPuller* object downloads, then they disappear once fully loaded. If you click on a table cell, the now-familiar graph is rendered and animated on screen through the canonical *UINavigationController* viewcontroller-pushing methods.

Now install and run it on your device. It seems like it's pretty slow to download, huh? It's nothing like the simulator experience. The user interface even freezes in fits and starts while the data comes in; you can't even scroll the table view most of the time. Once everything is downloaded, the interface is really responsive. We could chose to download only the data we need for the table view on launch, but that would only push our lack of responsiveness somewhere else, which would bring us dangerously close to death by a thousand paper cuts; It would probably take quite a while to load the data for a given graph on demand. You should also try to build and run in release configuration to see if perhaps the sluggish UI has anything to do with a certain lack of compiler optimizations. Nope. Let's profile this in Shark to see what's going on.

Shark

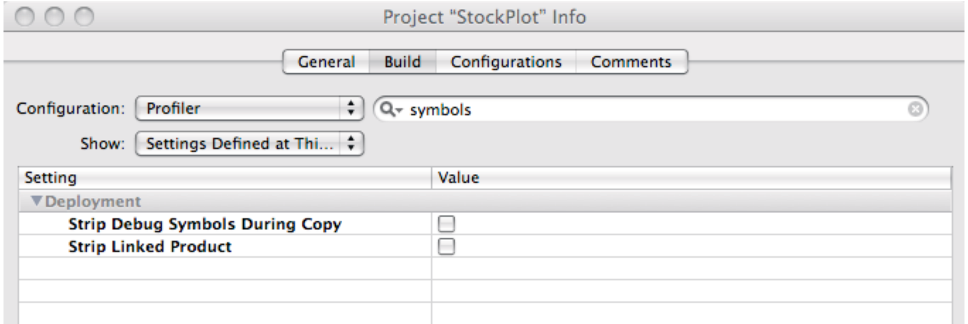
Shark is Apple's profiler. Attach it to a running process and it takes a sort of snapshot of what portion of your binary is running at regular intervals. Shark shows you a sort of weighted statistical table of how many times through a given method or line of code it counted. The more times it sampled your code in a given area, the more time your code spends in that area. You should always run Shark on a release build of your application because you will want to profile the compiler-optimized code with which your application will ship. There is one problem. The default settings for release also strip debugging symbols from your binary, which makes Shark look more like a hexadecimal puzzle game for those who can solve rubik's cube, of which I am certainly not one. Copy the release build configuration into one called Profiler by opening the Project info menu in Xcode duplicating the Release configuration.



Edit configuration list:

Debug
Release
Profiler

Then in the Build tab, uncheck the boxes for stripping debug symbols.



Configuration: Profiler

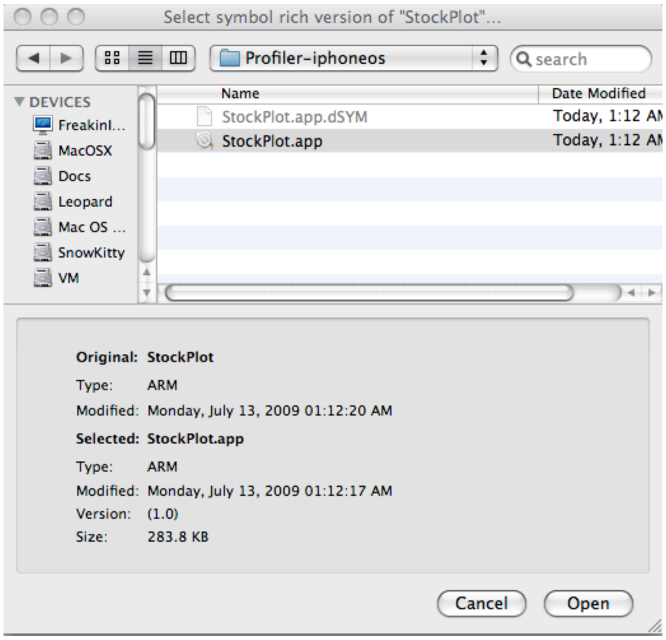
Show: Settings Defined at Thi...

symbols

Setting	Value
▼Deployment	
Strip Debug Symbols During Copy	☐
Strip Linked Product	☐

We have one thing left to do before we can test our downloading problem. Once your application is running, it takes a little while to attach the profiler. In order to test the problematical code, we need a way of attaching shark at the very start of the application run. While it might seem easy to drop a breakpoint in gdb in your main() function, I have had some trouble getting shark to connect while gdb is also attached. Instead, we'll drop a 10 second *sleep()* call in `applicationDidFinishLaunching`. That should give us enough time to attach Shark.

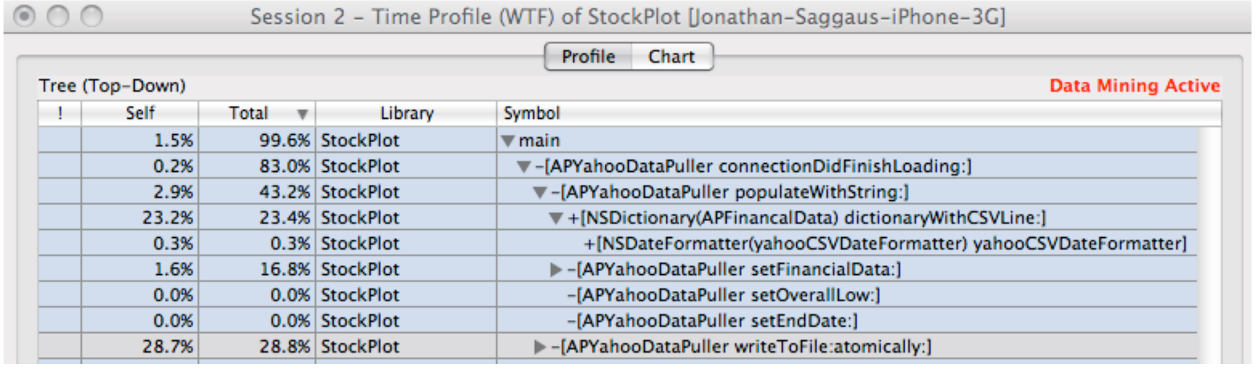
You can usually find Shark.app in in /Developer/Applications/Performance Tools/. Run it and select Network/iPhone Profiling from the Sampling menu. Delete the copy of StockPlot with cached data from your phone by using the Xcode organizer or directly on the iPhone. Connect your iPhone to your computer, build, and run the application. Once it's running (and sleeping), you can select the check mark in the menu next to the name of your iPhone, and select TimeProfile (WTF) from the Config dropdown, and Select StockPlot from the Target dropdown. As soon as you see log messages indicating download activity, hit the Start button; once the messages stop, hit the stop button. If you are a coffee drinker, now is a good time to go make a cup. This part takes a little time.because a lot of the processing that Shark needs to figure out what happened during the profiling run is actually performed on the device itself.



Name	Date Modified
StockPlot.app.dSYM	Today, 1:12 A
StockPlot.app	Today, 1:12 A

Original: **StockPlot**
Type: ARM
Modified: Monday, July 13, 2009 01:12:20 AM
Selected: **StockPlot.app**
Type: ARM
Modified: Monday, July 13, 2009 01:12:17 AM
Version: (1.0)
Size: 283.8 KB

Once Shark and your iPhone are finished, you might see a window with all kinds of hexadecimal jibberish that I promised wouldn't happen. If so, you will need to symbolicate the time profile by telling shark where the symbol-rich binary is located on your filesystem. Click on File->Symbolicate, then navigate to the iPhoneos build directory corresponding to your Profiler build settings. Make sure you see type: ARM on the window when you select it. Now shark should have familiar method names. Poking around in the trace we see that most of the work is being done in parsing the comma-separated strings and writing the plists to disk. That makes sense, we're using an asynchronous download, so that shouldn't freeze our UI, but the string parsing and caching to disk is blocking the main thread.


Session 2 - Time Profile (WTF) of StockPlot [Jonathan-Saggaus-iPhone-3G]
Profile
Chart
Tree (Top-Down)
Data Mining Active

!	Self	Total ▼	Library	Symbol
	1.5%	99.6%	StockPlot	▼main
	0.2%	83.0%	StockPlot	▼-[APYahooDataPuller connectionDidFinishLoading:]
	2.9%	43.2%	StockPlot	▼-[APYahooDataPuller populateWithString:]
	23.2%	23.4%	StockPlot	▼+[NSDictionary(APFinancalData) dictionaryWithCSVLine:]
	0.3%	0.3%	StockPlot	+[NSDateFormatter(yahooCSVDateFormatter) yahooCSVDateFormatter]
	1.6%	16.8%	StockPlot	▶-[APYahooDataPuller setFinancialData:]
	0.0%	0.0%	StockPlot	-[APYahooDataPuller setOverallLow:]
	0.0%	0.0%	StockPlot	-[APYahooDataPuller setEndDate:]
	28.7%	28.8%	StockPlot	▶-[APYahooDataPuller writeToFile:atomically:]

Even if we ship with 20 plists (which we probably would), they'll definitely be stale once the application gets into the users' hands. We don't want our application to be this unresponsive the first time it is run. What can we do about this? We have several options. At the moment, we're downloading and parsing all three months worth of data from Yahoo, because that's the easiest thing to do. We could figure out how much data we already have on disk and only download the missing data. We are also spending a fair amount of time converting *NSNumbers* to the *NSDecimalNumbers* we need for core-plot. We could change core-plot to accept *NSNumbers* or we could change our storage to CoreData, which retrieves *NSDecimalNumbers* without conversion.

The problem with the above optimizations, some of which we may chose to do before shipping, is that they will all incur unpredictable amounts of overhead on the main thread, thus we would have to test a lot of use cases. It may also prove difficult to predict just how much data we'll need. If your user uses your application often enough to pull down small chunks (in this case, fewer days) of data, which is not guaranteed, we might do well to avoid downloading duplicate data. We might also like to allow the user to add stocks to plot, which would definitely require a lot of parsing the first time the stock data is downloaded and it adds yet another stock to our queue on application launch. Perhaps we would do well to try to pull the processing off of the main thread so we can unblock the user interface once and for all, thus freeing us from all of these problems at once.

Concurrency

Cause it's gonna be the future soon. And I won't always be this way. When the things that make me weak and strange get engineered away.

-Jonathan Coulton

lyrics for *The Future Soon*

NSOperation NSOperationQueue, and Blocks

Wait a tick. Did I just suggest multi - threading?

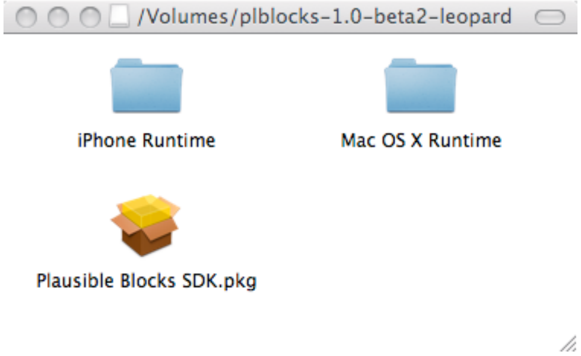
Okay. Threading is hard, but the engineers at Apple and elsewhere keep making it easier for us. We have all of these cores on our desktops because the hardware engineers keep slicing silicon so concurrency keeps getting more and more important. *NSOperationQueue* and *NSOperation* remove much of the pain of multi-threading. *NSOperationQueue* is a sort of thread pool that maintains an array of pending *NSOperation* objects that it schedules to run in a background thread based on a number of factors from system hardware to system state to the relative priority of a given *NSOperation*. You can even declare one *NSOperation* dependent on the completion of another. You normally subclasses *NSOperation* to override one method: `main`, which is where you put the work you want on a background thread. It's called when the operation is run. The only thing we as programmers have to be wary of in this situation are the usual data access caveats. Try not to mutate data at the same time you're reading it. <<<todo: work in a link to http://developer.apple.com/Cocoa/managingconcurrency.html>>> There are tools for this, too. We can use the various permutations of `performSelectorOnMainThread:`... and `@synchronized()` directives are useful, too.

There is a helpful tool in other languages for this kind of problem called blocks. Blocks are another name for closures, with which you may have familiarity from Ruby, LISP, Python, SmallTalk, and others. They're like function pointers that take a (usually const) snapshot of their local stack variables so you can run them later with the information you shove in them now. They're little portable units of computation that carry their state around that are extraordinarily useful with concurrent operations. Because they have a snapshot of their state, they're easier to deal with in a concurrent environment. Useful though that they would be, they don't officially exist yet. They're being added to Objective-C by the folks who are bringing us the open source Clang and LLVM projects <<<http://lists.cs.uiuc.edu/pipermail/cfe-dev/2008-August/002670.html http://

www.macresearch.org/cocoa-scientists-part-xxvii-getting-closure-objective-c >>> and there is no guarantee, though it seems likely, that Apple will bring them to the iPhone. If or when they do add them to the iPhone, switching from Plausible blocks will be simple. You'll revert to Apple's compiler and remove the Plausible blocks framework from your project.

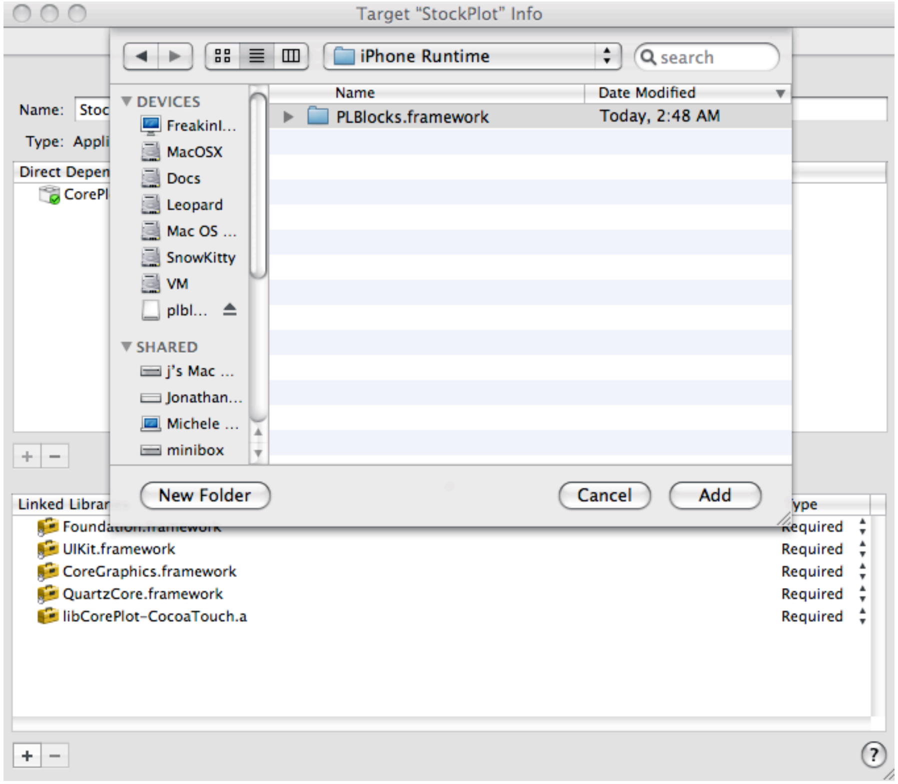
These additions to the objective-C language and runtime are open source and and they've been implemented in gcc 4.2, so it is actually quite possible to back port them to the iPhone, so of course they have been. Plausable Blocks from Plausible Labs is available at http://code.google.com/p/plblocks/ and is, as of this writing, shipping their second beta of a gingerly patched version of the standard, stable GCC 4.2 compiler that ships with the OS X Leopard (10.5) and the iPhone software development kits. I have found it to be very stable and It works with both iPhone OS 3.0 and 2.2.1 targets. There is some example code for their use on the primary author's github repository available at http://github.com/landonf/block_samples/tree/master. Next we'll install the Plausible Blocks compiler and add its static framework to our project so we can easily place our downloading, parsing, and saving code in a block to be executed by an *NSOperation* to be scheduled by an *NSOperationQueue* (in the house that Jack built). All of the things that make our application weak and strange are being gradually engineered away and we're even using future technology!

Installing the Plausable Blocks compiler and adding it to our project

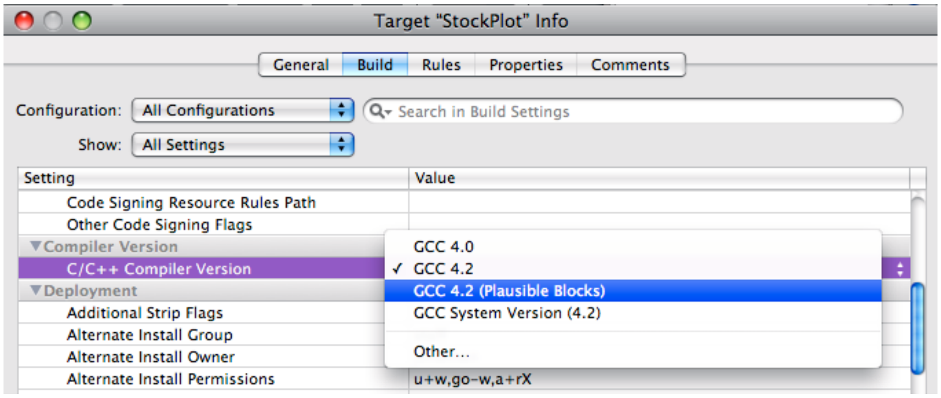


First, download the latest dmg of the Plausible Blocks compiler and frameworks from http://code.google.com/p/plblocks/downloads/list. Mount the dmg and run the included package. This installs the patched compiler as an Xcode plugin.

Now copy the iPhone Runtime folder, which includes the static framework we'll need to link against, into the StockPlot project. Double-click on the StockPlot target, select the General tab, and click the plus (+) button in the lower left corner of the window to add a new Linked Library. Click *Add Other* in the resulting sheet, navigate to and select the framework for addition.



Now we need to tell Xcode to use the special compiler. Double click the StockPlot target to bring up the build settings window. Select the Build tab. Select All Configurations from the upper-left dropdown. Now select the GCC 4.2 (Plausible Blocks) compiler. We now have blocks support.



Configuration: All Configurations
Show: All Settings
C/C++ Compiler Version: GCC 4.2 (Plausible Blocks)
Alternate Install Permissions: u+w,go-w,a+rX

We're going to use some convenience categories and objects from the Plausible Blocks sample code mentioned earlier. They're included with the sample code in <<Examples/05StockPlotParallelDownloads>> in files called NSThread+PLBlocks.h/m and NSOperationQueue+PLBlocks.h/m. Add them to the StockPlot project.

Using Blocks, NSOperation, and NSOperationQueue in StockPlot

To get asynchronous downloading, parsing, and saving the first thing we need to do is make something synchronous. Go figure. Our downloading code is using *NSURLConnection* to download the data asynchronously from yahoo. *NSURLConnection* doesn't like to be launched asynchronously from any thread other than the main thread because that would be silly. This isn't a big deal, because we're going to place all downloading, parsing, and saving in a background thread using the *NSOperation*/*NSOperationQueue* objects. This has the added benefit of making our downloading code simpler. Instead of asynchronously adding data to a *NSMutableData* object and defining a bunch of *NSURLConnectionDelegate* methods, we need only call the *NSURLConnection* `sendSynchronousRequest:returningResponse:error:` method. It blocks execution while downloading and can be run from a non-main thread, which is exactly what we want. Every time we call a delegate method from the background thread, we make sure that the delegate gets called on the main thread. Usually, we would use the `performSelectorOnMainThread:`... family of calls, but it's easier to wrap them in a block and have our new category on NSThread execute the block on the main thread. Here is the new `fetchIfNeeded` method.

APYahooDataPuller.m

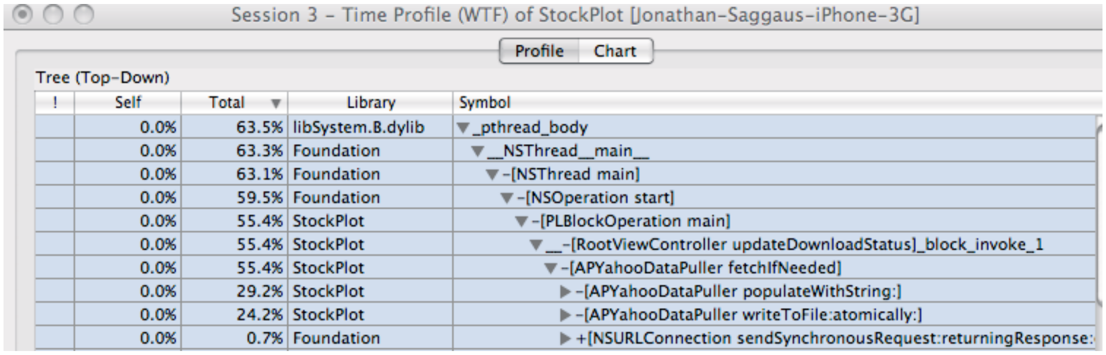
```
-(void)fetchIfNeeded
{
    if ( self.loadingData ) return;

    //Check to see if cached data is stale
    if ([self staleData])
    {
        self.loadingData = YES;
        NSString *urlString = [self URL];
        NSLog(@"Fetching URL %@", urlString);
        NSURL *url = [NSURL URLWithString:urlString];
        NSURLRequest *theRequest=[NSURLRequest requestWithURL:url
                                          cachePolicy:NSURLRequestUseProtocolCachePolicy
                                      timeoutInterval:60.0];
        // create the connection with the request
        // and start loading the data
        NSURLResponse *theResponse;
        NSError *theError;
        [self downloadWillStart];
        self.receivedData = [NSURLConnection sendSynchronousRequest:theRequest
                                                  returningResponse:&theResponse
                                                              error:&theError];
        if(theError)
        {
            self.loadingData = NO;
            self.receivedData = nil;
            NSLog(@"err = %@", [theError localizedDescription]);
            [[NSThread mainThread] pl_performBlock: ^{
            if(delegate && [delegate respondsToSelector:@selector(dataPuller:downloadDidFailWithError:)])
            {
                 [delegate performSelector:\
                       @selector(dataPuller:downloadDidFailWithError:)
                                withObject:self
                                withObject:theError];
            }
            }];
            [self connectionEnded];
        }
        else
        {
            self.loadingData = NO;
            NSString *csv = [[NSString alloc] initWithData:self.receivedData encoding:NSUTF8StringEncoding];
            [self populateWithString:csv];
            [csv release];
            self.receivedData = nil;
            [self writeToFile:[self pathForSymbol:self.symbol] atomically:NO];
            [self connectionEnded];
        }
    }
}
```

Aside on block syntax goes here. What's with the funny carat symbol?

This method is called from the *RootViewController*'s `updateDownloadStatus` method from within a `pl_addOperationWithBlock` class method that has been added to *NSOperationQueue*. This class method adds a *PLBlockOperation* to the queue and schedules it for execution. Notice that blocks need to be copied rather than retained. The *NSOperation* object subclass *PLBlockOperation* that gets instantiated here copies the block we pass it into an ivar (blocks are also objective-C objects!) and simply executes it in its `main` method. Since all of the stack variables are copied into the block, we needn't worry if they change or go out of scope before the block is called.

```
-(void)updateDownloadStatus
{
    while ([stocksToDownload count])
    {
        APYahooDataPuller *dp = [stocksToDownload objectAtIndex:0];
        NSOperationQueue *q = [(StockPlotAppDelegate *) [[UIApplication sharedApplication] delegate] 
globalQ];
        [q pl_addOperationWithBlock: ^{
            [dp fetchIfNeeded];
        }];
        NSUInteger idx = [stocks indexOfObject:dp];
        NSUInteger section = 0;
        NSIndexPath *path = [NSIndexPath indexPathForRow:idx inSection:section];
        UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:path];
        if(nil != cell)
            [self setupCell:cell forStockAtIndex:idx];
        [stocksToDownload removeObject:dp];
    }
}
```

Uninstall, build and run on the device. NSOperationQueue tends to be conservative on the iPhone, so you'll probably see stock information downloaded one symbol at a time; the application will remain responsive throughout. Just for fun, let's uninstall it and run it through shark again. If you've deleted it, add in that temporary call to *sleep()* as well.


Session 3 – Time Profile (WTF) of StockPlot [Jonathan-Saggaus-iPhone-3G]
Profile
Chart

Tree (Top-Down)

!	Self	Total ▼	Library	Symbol
	0.0%	63.5%	libSystem.B.dylib	▼_pthread_body
	0.0%	63.3%	Foundation	▼__NSThread__main__
	0.0%	63.1%	Foundation	▼-[NSThread main]
	0.0%	59.5%	Foundation	▼-[NSOperation start]
	0.0%	55.4%	StockPlot	▼-[PLBlockOperation main]
	0.0%	55.4%	StockPlot	▼__-[RootViewController updateDownloadStatus]_block_invoke_1
	0.0%	55.4%	StockPlot	▼-[APYahooDataPuller fetchIfNeeded]
	0.0%	29.2%	StockPlot	▶-[APYahooDataPuller populateWithString:]
	0.0%	24.2%	StockPlot	▶-[APYahooDataPuller writeToFile:atomically:]
	0.0%	0.7%	Foundation	▶+[NSURLConnection sendSynchronousRequest:returningResponse:

Notice that the application didn't really run any faster, we've just parallelized it. Multi-threading isn't so painful after all. Welcome to the future.

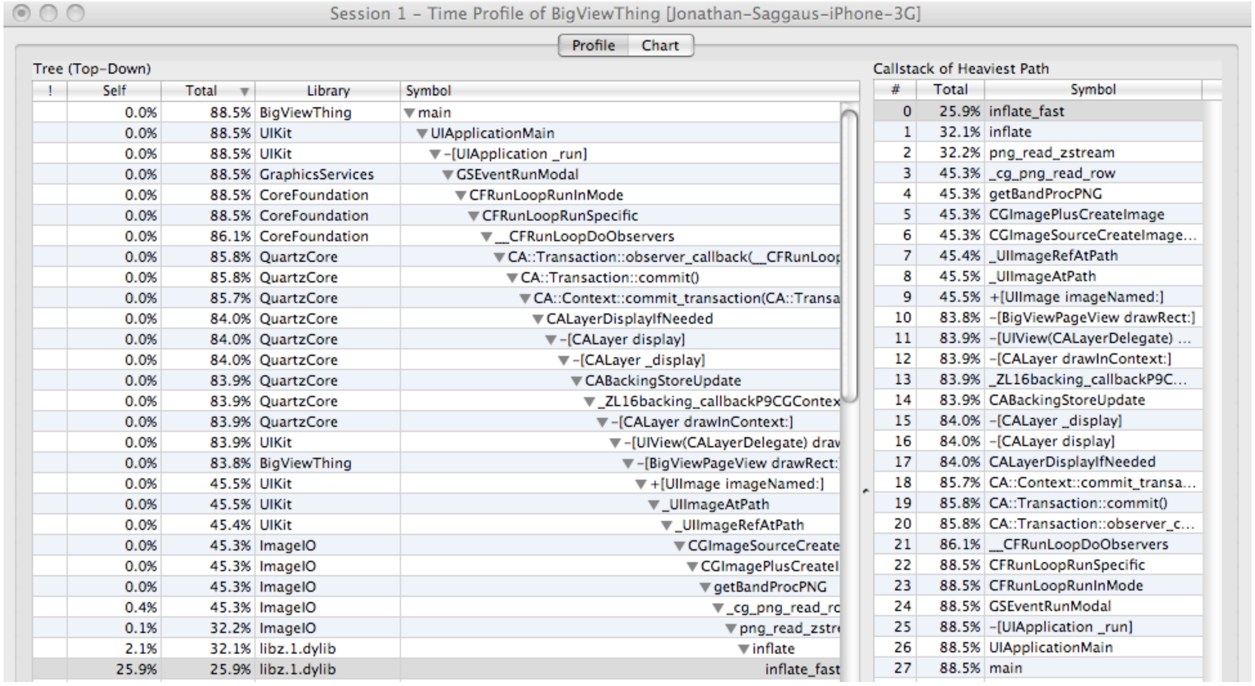
> Apple has not officially announced any intention to bring blocks to the iPhone, though it's a fair bet that they will do so once blocks are added to the desktop runtime and compiler collection. You should very thoroughly test any application using a non-standard compiler and be prepared for things to break in spectacular and unexpected ways. That said, Plausible Blocks appears well on its way to release-level stability.

Displaying large amounts of data

How easily the iPhone UI can be brought to its knees by performing something as seemingly simple as downloading, processing, and caching data to disk. Now we're going really make it hurt by throwing it an application that has to work very hard to draw anything at all. In this section, we'll examine a project for drawing a vertical succession of very large images a zoomable scroll view. So that we might encounter some of the difficulties inherent in dealing with large amounts of data, we are going to add an admittedly somewhat contrived requirement: the images cannot be pre-sliced, thumbnailed or otherwise massaged outside of the device. All drawing code must use the original large png images shipped with the application. If we can make this example perform reasonably well, we'll have a reusable framework for drawing any processor-intensive tiled scroll view. <<<IMAGES FROM NASA IMAGE OF THE DAY I THINK... NEED TO FIND COPYRIGHT INFO OR DIFFERENT IMAGES...>>>

We'll begin with a modified version of Apple's <<<TiledScrollerThingForgotItsName+ url>>> example available at <<< URL >>> called BigViewThing. The original Apple sample is designed to draw view tiles that are chunks of a larger image; it shows how to reuse view objects in two dimensions similar to the way the *UITableView* dequeues and enqueues rows in one dimension. In the case of Apple's sample, the image chunks are meant to ship with the application. BigViewThing is already partially implemented as a result of being derived from this sample code. It handles double-tap to zoom, suspends tile redraws when the user is interacting with the view and it draws only onscreen tiles. It's in the <<<Examples/06BigViewThingOriginal>> directory of the sample code. Build it and run. There are quite a number of large images in it, so it will take a while to copy over to the device.

Once you have it running, you'll notice a few issues. Whenever a new tile comes on screen, it takes a while to render. The image doesn't redraw at higher resolution when you zoom. It remains grainy. Let's profile it in Shark to see what is going on. There is no need to add a sleep() to this application as the performance problems appear throughout rather than just on startup. Start the application and attach Shark. Remember, the longer you sample, the longer you will wait for results so scroll around enough to get it to draw just a couple of images.


Session 1 – Time Profile of BigViewThing [Jonathan–Saggaus–iPhone–3G]
Profile
Chart

Tree (Top–Down)

!	Self	Total	Library	Symbol
	0.0%	88.5%	BigViewThing	main
	0.0%	88.5%	UIKit	UIApplicationMain
	0.0%	88.5%	UIKit	–[UIApplication _run]
	0.0%	88.5%	GraphicsServices	GSEventRunModal
	0.0%	88.5%	CoreFoundation	CFRunLoopRunInMode
	0.0%	88.5%	CoreFoundation	CFRunLoopRunSpecific
	0.0%	86.1%	CoreFoundation	__CFRunLoopDoObservers
	0.0%	85.8%	QuartzCore	CA::Transaction::observer_callback(__CFRunLoop
	0.0%	85.8%	QuartzCore	CA::Transaction::commit()
	0.0%	85.7%	QuartzCore	CA::Context::commit_transaction(CA::Transa
	0.0%	84.0%	QuartzCore	CALayerDisplayIfNeeded
	0.0%	84.0%	QuartzCore	–[CALayer display]
	0.0%	84.0%	QuartzCore	–[CALayer _display]
	0.0%	83.9%	QuartzCore	CABackingStoreUpdate
	0.0%	83.9%	QuartzCore	_ZL16backing_callbackP9CGContex
	0.0%	83.9%	QuartzCore	–[CALayer drawInContext:]
	0.0%	83.9%	UIKit	–[UIView(CALayerDelegate) drav
	0.0%	83.8%	BigViewThing	–[BigViewPageView drawRect:
	0.0%	45.5%	UIKit	+[UIImage imageNamed:]
	0.0%	45.5%	UIKit	_UIImageAtPath
	0.0%	45.4%	UIKit	_UIImageRefAtPath
	0.0%	45.3%	ImageIO	CGImageSourceCreate
	0.0%	45.3%	ImageIO	CGImagePlusCreateI
	0.0%	45.3%	ImageIO	getBandProcPNG
	0.4%	45.3%	ImageIO	_cg_png_read_rc
	0.1%	32.2%	ImageIO	png_read_zstre
	2.1%	32.1%	libz.1.dylib	inflate
	25.9%	25.9%	libz.1.dylib	inflate_fast

Callstack of Heaviest Path

#	Total	Symbol
0	25.9%	inflate_fast
1	32.1%	inflate
2	32.2%	png_read_zstream
3	45.3%	_cg_png_read_row
4	45.3%	getBandProcPNG
5	45.3%	CGImagePlusCreateImage
6	45.3%	CGImageSourceCreateImage...
7	45.4%	_UIImageRefAtPath
8	45.5%	_UIImageAtPath
9	45.5%	+[UIImage imageNamed:]
10	83.8%	–[BigViewPageView drawRect:]
11	83.9%	–[UIView(CALayerDelegate) ...
12	83.9%	–[CALayer drawInContext:]
13	83.9%	_ZL16backing_callbackP9C...
14	83.9%	CABackingStoreUpdate
15	84.0%	–[CALayer _display]
16	84.0%	–[CALayer display]
17	84.0%	CALayerDisplayIfNeeded
18	85.7%	CA::Context::commit_transa...
19	85.8%	CA::Transaction::commit()
20	85.8%	CA::Transaction::observer_c...
21	86.1%	__CFRunLoopDoObservers
22	88.5%	CFRunLoopRunSpecific
23	88.5%	CFRunLoopRunInMode
24	88.5%	GSEventRunModal
25	88.5%	–[UIApplication _run]
26	88.5%	UIApplicationMain
27	88.5%	main

Almost all of the application's execution time is being used in decompressing and drawing the png images. Our goal with this demonstration application is to simulate what happens when drawing very heavy, data intensive views. You never know when a user is going to try to load a giant document or image into your application. Some developers have run into this problem with Apple's *UIWebView*. It was designed to render small email attachments in various formats in the Mail application and to render web content. Several document reader applications fail when the user tries to load a large document because they are trying to leverage *UIWebView* to draw heavy content. It clearly isn't designed for such content.

Zooming a UIScrollView

One thing that *UIWebView* performs very well is the redrawing of zoomed content. In BigViewThing, we're currently allowing the scroll view to zoom for us and leaving the content alone when zooming is finished. This results in an unpleasant grainy appearance because the *UIScrollView* that we use to host our content simply applies a scaling affine transform to our content view. It's also expanding or contracting its own content size relative to the new drawn size of the overall view. *UIScrollView* does this for performance reasons. If it takes three seconds (and it does take that long right now in our application) to draw a view into a given square of pixels, imagine what it might look like to animate resizing by redrawing. 1/3 of a frame per second is subpar to say the least.

Search the internet for "UIScrollView zooming reset resolution" and you'll find a lot of developers pulling their hair out trying to get this to look right. A little caveman *NSLog* experimentation to figure out what the *UIScrollView* is really doing can reveal what's happening under the covers when you (say) pinch to zoom or directly set the zoomScale property of a *UIScrollView*.

UIScrollView internal zooming algorithm

1. UIScrollView checks to see if minimumZoomScale and maximumZoomScale are not equal to one another. It also checks the current zoomScale to see if it can zoom.
2. If so, it asks the *UIScrollViewDelegate* for a view to scale during the zoom with the `viewForZoomingInScrollView:` method call. We return our content view in the BigViewThing project.
3. As the zoom scale changes, the *UIScrollView* does two things:
    a. It sets an affine transform on the view it is zooming to scale it up or down without redrawing. It's a "square" transform that maintains aspect ratio, so there is no distortion.
    b. It resets its own contentSize, contentOffset, and zoomScale so as to to hold the content in place relative to the point about which it is zooming (in the case of pinching, that point was halfway between your fingers when you put them down).
4. If the zoom was performed with a pinch gesture or through the `setZoomScale:animated:` methods, it calls `scrollViewDidEndZooming:withView:atScale:` on its delegate when the zooming ends. However, it does not call this delegate method if the `animated:` argument was NO because the zoom is set instantly when you call the method. The *UIScrollView* assumes that you know that it finishes zooming right away in that case.
5. After zooming, the *UIScrollView* leaves the affine transform on the view, and it leaves the stretched contentSize, contentOffset, and zoomScale in place, which is why our view seems grainy. It's still being stretched when we zoom.

Armed with knowledge of some of the internal workings of *UIScrollView*, we can now reset drawing after a zoom by implementing and calling an `updateResolution` method when zoom finishes. My preferred method of updating resolution for zooming is as follows.

UIScrollView resolution update algorithm

1. Take a snapshot of the current (scaled) contentSize and contentOffset.
2. Take a snapshot of the current (unscaled) content view's frame size; it's being scaled by an affine transform, so its actual frame size is the same as it was before zooming.

3. Take a snapshot of the current minimum and maximum zoom scales.

4. If your scrollview is its own delegate as it is in BigViewThing, call `super` to set the minimum and maximum zoom scale both to 1.0 because setting zoom on self will eventually call `updateResolution` again; infinite recursion is so last year.

5. Set the current zoom scale to 1.0, this will rescale the content size internally back to the size of the content view and reset the affine transform on the content view.

6. Calculate new content offset by scaling the stretched/zoomed offset we took a snapshot of in step 1. We want the new content to appear in the same place in our scroll view:

    a. newContentOffset.x *= (oldContentSize.width / contentViewSize.width);

    b. newContentOffset.y *= (oldContentSize.height / contentViewSize.height);

7. Divide the old minimum and maximum zoomScale by the new zoomscale. This scales the minimum and maximum zoom relative to our new content size. If minimum zoom were 1.0 and maximum zoom were 2.0, when the user zooms to 2.0 and I reset, my new minimum zoom will be .5 and my new maximum zoom will be 1.0.

8. Set the content view's frame.size to the contentSize we took a snapshot of in step 1.

9. Set the scroll view's contentSize to the scaled contentSize we took a snapshot of in step 1. This stretches the overall size of the view to match our new zoom level (but without any affine transform applied).

10. Call the `setNeedsLayout` method on the scroll view. This will cause `layoutSubviews` to be called where we can reset the contentview's internal subview geometry.

Here is an implementation of the above that we'll add to our BigViewScrollView. We'll call it whenever zooming finishes.

```
- (void)updateResolution {
    //LogMethod();
    isdblTapZooming = NO;
    float zoomScale = [self zoomScale];

    CGSize oldContentViewSize = [contentView frame].size;
    //zooming properly resets contentsize as it happens.
    CGSize newContentSize = [self contentSize];

    CGPoint newContentOffset = [self contentOffset];
    float xMult = newContentSize.width / oldContentViewSize.width;
    float yMult = newContentSize.height / oldContentViewSize.height;

    newContentOffset.x *= xMult;
    newContentOffset.y *= yMult;

    float currentMinZoom = [self minimumZoomScale];
    float currentMaxZoom = [self maximumZoomScale];
    float newMinZoom = currentMinZoom / zoomScale;
    float newMaxZoom = currentMaxZoom / zoomScale;

    //don't call our own set..zoomScale, cause they eventually call this method.
    //Infinite recursion is uncool.
    [super setMinimumZoomScale:1.0];
    [super setMaximumZoomScale:1.0];
    [super setZoomScale:1.0 animated:NO];

    [contentView setFrame:CGRectMake(0, 0, newContentSize.width, newContentSize.height)];
    [self setContentSize:newContentSize];
    [self setContentOffset:newContentOffset animated:NO];

    [super setMinimumZoomScale:newMinZoom];
    [super setMaximumZoomScale:newMaxZoom];

    // throw out all tiles so they'll reload at the new resolution
    [self reloadData];       //calls setNeedsLayout, among other things for housekeeping
}
```

Build and run <<<Examples/07BigViewThingZoomAddition>>> in the simulator. The images should clear - up after a zoom. Speaking of the simulator, this demo application takes a very long time to install on the device because it's copying all of the images over USB each time. Since we are about to spend some time focusing on a single performance bottleneck in our code, image drawing, we can simulate this slowness in the simulator with a call to *sleep().* Avoiding the copy of those png files will make debugging go a little faster while simulating our problem reasonably well. Also, I tend to forget to remove these *sleep()* calls when compiling for the iPhone and wonder why everything slows down when I move back to the device, so let's #define this one to only compile into the simulator target. Add the following to drawRect: in *BigPageView.m*

```
    if(!drawingSuspended)
    {
        CGContextSetFillColorWithColor(context, [[UIColor whiteColor] colorWithAlphaComponent:0.5].CGColor);
        CGImageRef tempImage = [UIImage imageNamed:self.imageName].CGImage;
#if TARGET_Iphone_SIMULATOR
        sleep(2.5);
#endif
        CGContextDrawImage(context, tempbounds, tempImage);
        drawnPageOnce = YES;
    }
```

Build and run in the simulator. You should see similar sluggishness compared to running on the phone. Let's tackle that problem now.

Drawing into an offscreen context

Given our self-imposed limitations, we can't make the drawing much faster without digging into openGL. Even then, we'll have to decode the images and throw them up into texture memory no matter what we do, so the drawing itself would be fast, but we know from our Shark profile that the decoding is what takes a long time. It's time to take our *NSOperationQueue* and blocks magic to the next level and parallelize the drawing.

Danger Will Robinson! UIKit is not threadsafe. Try to draw to screen from another thread and bad things might happen, ugly things are almost guaranteed to happen. We can, however, draw our images into offscreen buffers (actually, cgContexts) then grab the pixels that we need to throw on the screen once the buffer is filled with data. There is nothing stopping us from filling that data asynchronously and reading it from the main thread.

Algorithm for drawing into an offscreen context

1. The first time one of our BigViewPageView objects is asked to draw, it will create a cgContext type instance variable into which it will quickly draw the half opaque white background that we are currently drawing as a placeholder when the BigViewPageView is inactive like so:

```
-(void)initOffscreenContext // do this on the MAIN thread
{
    CGSize layerSize = [self bounds].size;
    layerSize.height = floorf(layerSize.height);
    layerSize.width = floorf(layerSize.width);

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGContextRef ctx = (CGContextRef) [(id) CGBitmapContextCreate(NULL, layerSize.width, layerSize.height, \
                         8, layerSize.width*4, colorSpace, kCGImageAlphaPremultipliedLast) autorelease];
    CGColorSpaceRelease(colorSpace);
    CGContextTranslateCTM(ctx, 0, layerSize.height);
    CGContextScaleCTM(ctx, 1.0, -1.0);

    CGFloat tx = layerSize.width * (1.0 - scale) * 0.5;
    CGFloat ty = layerSize.height * (1.0 - scale) * 0.5;
    CGRect tempbounds = CGRectZero;
    tempbounds.size = layerSize;
    tempbounds = CGRectIntegral(CGRectInset(tempbounds, tx, ty));
    CGContextSetShadow(ctx, CGSizeMake(5,5), 5);
    CGContextSetFillColorWithColor(ctx, [[UIColor whiteColor] colorWithAlphaComponent:0.5].CGColor);
    CGContextFillRect(ctx, tempbounds);
    self.offscreenContext = (id) ctx;
}
```

2. It will draw whatever is in the offscreen context to screen in `drawRect:`

```
-(void)drawRect:(CGRect)rect
{
    //NSLog(@"drawRect");
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextRef osc = (CGContextRef) self.offscreenContext;
    UIGraphicsPushContext(osc);
    CGImageRef tempImage = CGBitmapContextCreateImage (osc);
    UIGraphicsPopContext();
    if(tempImage)
    {
        CGContextDrawImage(context, self.bounds, tempImage);
        CGImageRelease(tempImage);
        drawnPageOnce = YES;
    }
}
```

3. It will generate an *NSOperation* (that calls a block, of course) that will fill a new cgContext with the image data we will need:

```
-(void)createOffscreenCtx
{
    NSOperationQueue *q = [(BigViewThingAppDelegate *) [[UIApplication sharedApplication] delegate] 
globalQ];
    PLBlockOperation *op = [PLBlockOperation blockOperationWithBlock:^{
        //imgRef = [[UIImage imageNamed:imageName] CGImage];
        NSString* bundlePath = [[NSBundle mainBundle] bundlePath];
        UIImage *img = [UIImage imageWithContentsOfFile:[NSString stringWithFormat:@"%@/%@", bundlePath, 
                                                                                                 imageName]];
        CGImageRef imgRef = [img CGImage];

        CGSize layerSize = [self bounds].size;
        layerSize.height = floorf(layerSize.height);
        layerSize.width = floorf(layerSize.width);
        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGContextRef ctx = (CGContextRef) [(id) CGBitmapContextCreate(NULL, layerSize.width, 
                                          layerSize.height, 8, 
                                          layerSize.width*4, colorSpace, 
                                          kCGImageAlphaPremultipliedLast) autorelease];
        CGColorSpaceRelease(colorSpace);
        CGContextTranslateCTM(ctx, 0, layerSize.height);
        CGContextScaleCTM(ctx, 1.0, -1.0);

        CGFloat tx = layerSize.width * (1.0 - scale) * 0.5;
        CGFloat ty = layerSize.height * (1.0 - scale) * 0.5;
        CGRect tempbounds = CGRectZero;
        tempbounds.size = layerSize;
        tempbounds = CGRectIntegral(CGRectInset(tempbounds, tx, ty));
        CGContextSetShadow(ctx, CGSizeMake(5,5), 5);
#if TARGET_Iphone_SIMULATOR
        sleep(2.5); //fake slow drawing on the simulator
#endif
        CGContextDrawImage(ctx, tempbounds, imgRef);
        self.offscreenContext = [[(id) ctx retain] autorelease];
        NSLog(@"Image loaded for %d", pageToDraw);
        //when we're done filling, we need to redisplay content
        [self performSelectorOnMainThread:@selector(setNeedsDisplay) withObject:nil waitUntilDone:NO];
    }];
    [q addOperation:op];
}
```

4. When the *NSOperation* finishes, it will `setNeedsDisplay` on the view in the main thread so the view knows to draw the image data to screen. We can do this in real time. Drawing from a buffer is fast.

5. Any time the BigViewPageView is asked to `drawRect`, it pulls the image data from the current cgContext for drawing; it's also filling new cgContexts in the background if we change the expected drawing size of the image through some bizarre action like zooming. Before the new buffer is ready, our image will stretch to fill, possibly pixelated, while the *NSOperation* is preparing new data.

The sample code in <<<Examples/08BigViewThingOperationQueueRegular>>> has all of the additional code. It also prints the contents of the *NSOperationQueue* on a timer to show us what is in there. Build and run in the simulator. The application should remain responsive.

Or is it? Every time I zoom in or zoom out on an image, the view pushes another *NSOperation* onto the queue. If you watch the log messages printing the contents of the *NSOperationQueue*, you will see that there are an ever-growing number of operations for each view getting pushed when there is a lot of zooming going on. This makes the app seem like it's updating less and less often. The queue eventually clears, but not after drawing a given image several times, usually at zoom levels at which it is not currently needed to be rendered.

Wouldn't it be nice to be able to cancel only certain pending operations on the *NSOperationQueue*? You can. You just call the cancel method on your *NSOperation* object and the queue will eventually (but not immediately) remove it, but it will never actually run it. We can add a weak reference to our *NSOperation* subclass to point back to the BigViewPageView object that placed it on the queue and then ask each *NSOperation* that belongs to us to cancel before we add another operation to the queue. This way, we can be sure that there is little wasted CPU time[*].

Once we have that weak reference, it's easy to create a category on *NSOperationQueue* to cancel all pending *NSOperations* in the queue filtered by a *NSPredicate*.

```
- (void)cancelOperationsFilteredByPredicate:(NSPredicate *)predicate;
{
    NSArray *ops = [[self operations] filteredArrayUsingPredicate:predicate];
    for (NSOperation *op in ops)
    {
        if(![op isExecuting] && ![op isFinished] && ![op isCancelled])
        {
            [op cancel];
        }
    }
}
```

If you notice that the *NSOperation* objects stay in the queue for a while, that is okay. When *NSOperationQueue* decides that it is time to run a given operation, it will call start on the *NSOperation* and wait for that operation to finish executing. If `isCancelled` returns YES, the *NSOperation* will tell the *NSOperationQueue* that it is finished right away without ever calling the main method. Add the cancellation code into our BigViewPageView.

```
-(void)createOffscreenCtx
{
    NSOperationQueue *q = [(BigViewThingAppDelegate *) [[UIApplication sharedApplication] delegate]
                                                                                  globalQ];
    NSPredicate *filter = [NSPredicate predicateWithFormat:@"SELF.interestedObject == %@", self];
    [q cancelOperationsFilteredByPredicate:filter];
    PLBlockOperation *op = [PLBlockOperation blockOperationWithBlock:^{
//BUNCH of drawing code here
}];
```

---

[*] * In our implementation, an operation in progress cannot be cancelled, so it's still possible that the queue will have to run two operations for a given view in fairly rapid succession.

```
    [op setInterestedObject:self];
    [q addOperation:op];
}
```

Other suggestions

BigViewThing is not finished yet. We've just implemented something similar in behavior to *CATiledLayer*, perhaps we would find that even more performant than our *NSOperationQueue* code. *NSOperation* can have an attached priority. Perhaps we could place a series of low-priority operations on the drawing queue to fill the cgContext buffers with a low resolution version of each image so that the user's offscreen tiles will get drawn in the background using idle CPU cycles, thus removing the grey placeholders. When we zoom back and forth between different levels, we might not really need to re-render each time. Perhaps the transform from big zoomed-in image to small zoomed-out image looks okay to us without a redraw. Buffer size issues aside, perhaps we could allow a delay in redrawing the tiles at a smaller size when the user zooms out by lowering the priority of that operation, that way operations that dramatically change the user experience will run first*.

> Little summary of NSPredicate here and a link to the docs. ...NSPredicate was added in iPhone OS 3.0. It is an important part of CoreData. It has been available to Cocoa programmers on the desktop for a long time. It's super useful. I missed it.

Observations, tips, tricks

iPhone programming is embedded systems programming. While you can expect iPhone devices to become faster and faster over time, programming for iPhone is closer to that of a Nintendo DS or a LART box than a desktop computer. Our examples will seem slow before we optimize on the new, faster, iPhone 3GS, just less so than on the original device. It's always helpful to learn some embedded system programmers' tricks by programming for even more limited devices like LARTs or SBCs. You can often sort of "fake it 'till you make it" when it comes to code that requires a lot of system resources. UI Response variability is particularly annoying; users don't know why your app is slow on Edge network. "Sometimes it's slow; sometimes it's not; I dunno why." is a phrase to which we are becoming perhaps too accustomed. Clever caching of data while remaining responsive to the user's input can make an application shine, even when it isn't really doing much more than what it did before.

iPhone devices are severely memory constrained, disk read/write speed constrained, and bandwidth constrained when compared to their bigger iron cousins. Remember that UI and data share RAM, so you might get memory warnings at seemingly strange times . You'll notice some CPU and memory monitoring code in some of the example code, you can use it in your application to anticipate memory resource shortages and modify your application's behavior. Once you do get a memory warning, you receive a short warning and then the system kills your app without prejudice or allowing you to save precious user data, so be prepared to strip down your views and your data at a moment's notice. There is rudimentary handling of that in the BigViewThing example. Look at the method called `memoryWentBoom`.

Summary

…

OUTLINE::::