Weather Application Development for the iOS Platform

William Lamond

***Abstract*—With such a large percent of the population bearing smartphones, application developers who are serious about deploying their applications to as many people as possible should earnestly consider mobile application development. As one of the most popular smartphone platforms in the world, iOS an excellent choice to reach a large audience.**

***Index Terms*—iOS, Weather, Apple Developer Connection, TouchXML, Jailbreaking**

I. Introduction

DURING the spring semester of 2011 at the University of Maine, I conducted an independent study regarding mobile application development for the iOS platform. A large portion of the consumer computing market already involves smartphone technology, and the size of this group is growing quickly. This trend highlights the importance of mobile application development for software development companies and educational organizations.

In order to meet consumer needs, application developers need to seriously consider mobile application development when creating applications for every-day use. To stay ahead of the competition, developers need intimate knowledge regarding their platform of choice in order to make functional, beautiful applications that are easy to use and meet consumers' needs.

My work involved developing a weather application, called *TouchWeather*, in order to learn about mobile application development and to develop the intimate knowledge needed to create such applications on the iOS platform. In order to complete my goals, I had to learn about the iOS platform, Objective-C, XML formatted data provided by the National Oceanic and Atmospheric Administration (NOAA), Xcode and related development tools, and a number of nuances and issues that need to be addressed when developing applications for iOS devices.

II. iOS

iOS is the operating system running on the iPhone, iPad and iPod Touch devices. iOS is based on similar technologies found in Mac OS X. iOS is divided into layers, with each layer providing a set of interfaces to application developers and/or higher layers. Apple recommends that application developers investigate technologies provided by higher layers before resorting to lower layers, since the abstractions provided by the higher layers results in simpler code with better readability. The higher levels provide the same services the lower levels provide, but with simplified, object-oriented interfaces. The lower levels are still provided for developers that wish to use features not directly accessible by the higher layers of iOS.

William Lamond is an undergraduate student in the Department of Computer Science, University of Maine, Orono, ME 04473. Phone: +1–207–974–9246, e-mail: lamond.will@gmail.com

A. Cocoa Touch and UIKit

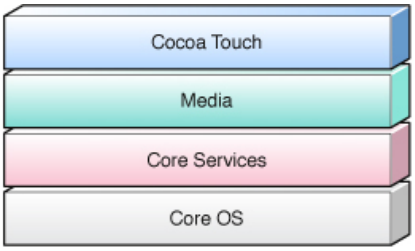

Fig. 1
iOS layers.

Cocoa Touch [1] provides high-level API's for behaviors and features users have come to expect from the iPhone. The name of the game when it comes to iPhone applications is usability, look, feel, and an intuitive sense. Cocoa Touch's UIKit framework provides the key infrastructure needed for creating event-driven, touch-based applications that bring to the table what iPhone users want to see in their applications.[2] Many applications use this layer for the majority of the implementation, since the high-level APIs provide the needed functionality with simpler interfaces (in comparison to the lower-level layers that do the same thing, in a more technical way).

B. Lower Layers

The lower layers of iOS provide access to lower-level details and features, such as sockets and file system access. These features are also included in the Cocoa Touch layer, and all applications rely on these layers to implement features found in higher layers. Lower levels do no impose the same structures and patterns on developers, increasing flexibility at the cost of increased complexity.

III. Getting Started

We need to do some essential things to start developing for iOS. To illustrate the basic development cycle for an iOS application, we will refer to *TouchWeather* throughout this report as a working example.

A. Join the Apple Developer Connection

Apple has an organization called the Apple Developer Connection (ADC), which is central to all Apple developers. The ADC provides developer tools, the iOS SDK,

documentation, sample code, developer forums, and more. There are two membership options in the ADC:

- Basic Apple Developer Program membership ($0): Provides documentation, tools, and the other features listed above. This membership does NOT provide a valid Apple developer key. The key is used to digitally sign your applications to verify your membership in the Apple developer program. You can still create the application, but you can only test it on the iPhone simulator (provided with the developer tools).
- Full Apple Developer Program membership ($99/year): Provides everything the free membership provides, and a valid key. The key allows developers to test their applications on actual iOS devices instead of just on the simulator, and to release their applications to the App Store once they have passed a thorough inspection by Apple.

B. Xcode and the iOS SDK

Apple provides developer tools for Intel-based Macs running Mac OS X 10.6. Xcode is Apple's collection of tools that provide support for code editing, interface design, debugging, performance tuning, project management, executable building, and more. Installation is very straightforward: all you need to do is download and mount the disk image, run the executable inside, and follow the prompts. Once Xcode is installed, you can launch it and begin development

C. Start a New Project

In Xcode, go to File⟹New Project and the New Project window will appear. Select a View-based Application to start making a simple application. Initially, the view-based application template only provides one view for displaying content, so to make more functional applications one needs to modify the template.

IV. Creating an Application

Now that the developer tools are installed and we have a project to work with, the first thing to do is start creating content. Before we start, we should consider the organization of our content. Since an application with one view can accomplish very little, it is advantageous to structure your views in a hierarchy. An easy way to do this is to use a `UINavigationController` [3] [4] to allow the user to traverse the hierarchy in an intuitive way. First, you must specify the content of your view to be pushed onto the navigation controller stack. The easiest way to create your content is to use the Interface Builder (IB) tool.

A. Create a New View Using Interface Builder

IB allows developers to build graphical user interfaces (GUIs) by dragging and dropping components onto a view. Create a new view-controller by going to File⟹New File and selecting `UIViewController` [5] subclass, which we will call `RootViewController`. By default, the checkbox labeled "With XIB for user interface" is checked, and in order to use IB we want to leave this as it is. This creates an implementation file (RootViewController.m), an interface file (RootViewController.h) and an XIB file [6] (RootViewController.xib). The XIB file is an XML file that describes the layout of a view for the view-controller to generate. This is an alternative to programmatically laying out the view by specifying widget frame size and position on the view in the implementation file. Files with the .xib extention are called nib files for historical reasons, so we will refer to them as nib files.

V. Adding Your Content

To continue our working example, we will add a text input field to the view we created earlier. To do this, simply open RootViewController.xib, which was created when we made the `UIViewController` subclass `RootViewController`. Go to Tools⟹Library and a window will open with a set of objects you can add to the view. Find the `UITextField` [7] in the library, and drag one onto the view. You can resize and move it on the view by moving the handles and dragging it, or change the values in the inspector. To access the inspector, go to Tools⟹Inspector and a window will open up.

Once the text field is in place, we have to add it in the code. Open the interface file, RootViewController.h, and add the following code.

```
@interface RootViewController : UIViewController
{
        //add this line:
        UITextField *input;
}

//and these lines:
@property (nonatomic, retain)
IBOutlet UITextField *input;
```

The interface declares the instance variable `input` as a pointer to a `UITextField` object and makes a property declaration.

A. Property Declaration and Accessor Methods

Properties are declared with the `@property` declaration. [8] Properties specify accessor methods for instance variables with given attributes, instead of of making accessors by hand to get and set the value of an instance variable. In the above example, `nonatomic` and `retain` are the access attributes. Atomic access is described in subsection N, and `retain` messages are described in the retain section of section M. `IBOutlet` states that `input` is an Interface Builder outlet, and is described in section Q. Combine with an `@synthesize` directive in the implementation file, properties automatically generate accessors to the instance variable with described implementation attributes. The basic structure of an implementation file (such as RootViewController.m) is:

```
@implementation RootViewController

@synthesize input;

//implement methods here...

@end
```

B. Atomic Access

The `nonatomic` [8] property states that access is not done atomically, or in one non-interruptible function. This means that access between threads is not safe, since one thread can modify the contents of the instance variable while another thread is accessing the contents. By default, properties specify atomic access. In an application without multiple threads accessing specific data however, nonatomic access is faster since these data do not need to be checked for consistency during calls to accessor methods.

VI. Memory Management

iOS applications manage memory using a "reference-count" system of object ownership. [9] Objects have a `retain` count, which is used to count the number of owners, or referrers, to the object. Object ownership follows these fundamental policies:

- You own an object allocated using `alloc`, `new`, `copy`, or `mutableCopy`.
- You own an object when you send it a `retain` message.
- You must relinquish ownership of an object when you are done with it by sending it a `release` message. Alternative, you can add the object to an autorelease pool to have it released at some point in the future.
- You must not release an object you do not own.

A. Retain

The `retain` property states that the setter method sends the object a `retain` message. A retain message is used designate object ownership by incrementing the object's `retain` count. When an object sends another object a retain message it is claiming (at least partial) ownership of the object. There are a few ways to gain ownership of an object, and multiple objects can all have partial ownership of one object. Any time you allocate a new object using methods such as `alloc`, `new`, `copy` or `mutableCopy`, you claim ownership through an implicit `retain` call. You can also explicitly `retain` an object if you are claiming ownership to something that already exists so that it is not deallocated before you are finished with it.

B. Release

When an instance variable is set using an accessor synthesized from a property, it sends a `release` message to the old value. The `release` message decrements the object's `retain` count. Once you are done with an object, you must send it a `release` message. When the `retain` count reaches 0, the object is removed from memory, or deallocated, and its `dealloc` method is called. Inside of the `dealloc` method, an object should release whatever instance variables it is keeping. You must send `release` messages to objects you own when you are finished with them. Failure to do so causes a memory leak. Releasing an object you do not own can cause it be deallocated before its owners are finished, causing the application to crash. The rule of thumb is to release an objects you allocate or ones you assign to an object's instance variable using an accessor with the `retain` property.

C. Autorelease Pools

Autorelease pools [10] are used to hold objects that are to be sent `release` messages when the pool is "drained." To use an Autorelease pool, allocate an `NSAutoreleasePool` object and initialize it. To put an object in the pool, send it an `autorelease` message. Objects that receive the `autorelease` message are sent a retain message when the autorelease pool receives a `drain` message.

Autorelease pools are most useful when you need to release an object but you are also returning it from a method. If you simply release the object, it will be deallocated and thus invalid when you try to return it. If you do not release it, it will result in a memory leak. To avoid this, the object can be added to a pool in the calling code, returned, and the pool can be drained later when the object has been retained by the caller. This allows you to conform to memory management policies while allocating objects in methods that are going to be returned eventually.

VII. Linking IBOutlets to Objects Laid Out With IB

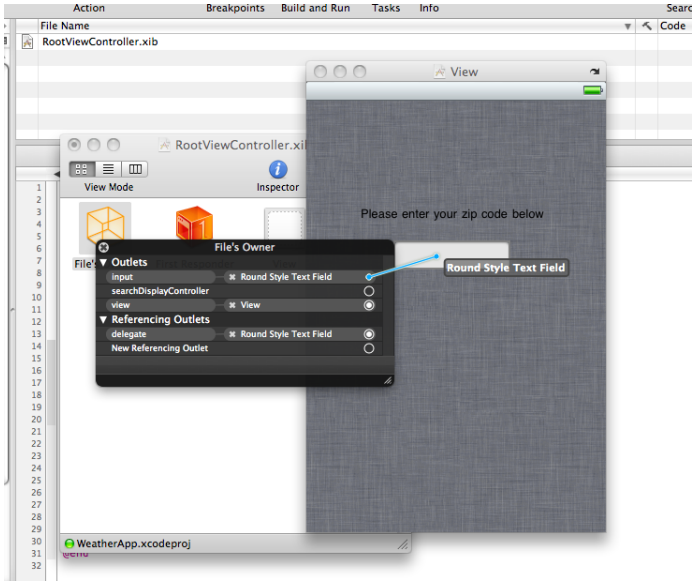

Fig. 2
Linking objects in the nib to objects in code.

The next step is to connect the text field in the interface to the text field in the code. This is done using IB to specify object owners for interaction protocols. In IB, right click on the `RootViewController`'s Files Owner, which will open an outlet inspector. In the outlet inspector, you will see the `UITextField input`'s outlet, as specified by the property declaration, that we created before. Click and drag from the circle on the right side of the name to the text field on the view itself, as shown in Figure 2.

This links the outlet in the interface file (RootViewController.h) to the `UITextField` in the nib file. Any changes made to one will affect the other. Now when text is entered into the field it can be accessed through `input` in the `RootViewController`.

VIII. Application Structure

The application starts like any C-like application: in the `main` method. This method creates an autorelease pool and calls the implicitly defined `UIApplicationMain` function.

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool
        alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil,
        nil);
    [pool release];
    return retVal;
}
```

All the action is happening in `UIApplicationMain`. This method instantiates and sets the delegate (`TouchWeatherAppDelegate`), as well as starts the main application loop. The `TouchWeatherAppDelegate` object is the top-level delegate object. It receives notifications from the main application loop (in `UIApplicationMain`), such as when the application finishes launching or a user touches the screen, so that custom behavior can be implemented in the methods called by the loop. One method called is *didFinishLaunchingWithOptions*. This method is used for top-level behavior specification, such as view organization and adding views to the window. In our case, we are interested in using a `UINavigationController` with our custom view controller, `RootViewController`. To achieve our hierarchy, we allocate the two controllers and push the `RootViewController` onto the `UINavigationController`'s view stack.

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary
            *)launchOptions
{

         //make the root view
        topView = [[RootViewController alloc] init];

         //push onto nav controller
        nav = [[UINavigationController alloc]
            initWithRootViewController:topView];

        //add the nav controller as a subview
        [window addSubview: [nav view]];

        [window makeKeyAndVisible];

                return YES;
}
```

In the `init` method of the `RootViewController`, you can specify the text at the top of the navigation controller. I made mine say, TouchWeather by adding the following line to `init`

```
self.title = @"TouchWeather";
```

Every view pushed onto the navigation stack has access to the navigation controller. The navigation controller makes its title whatever the top view controller sets its title to. The view controller at the top of the stack can also push another view onto the stack, which we will see shortly.

A. Model-View-Controller Paradigm

By creating a new `UIViewController` subclass and associated nib file, we have made our first application component that follows the model-view-controller (MVC) paradigm. [3] All iOS applications should follow the MVC paradigm, which results directly from using the high-level APIs of the Cocoa Touch layer and templates provided by Xcode. The MVC paradigm forces the separation of data (the model) and the view (UI), with the controller acting as a delegate between the two. This separation of concerns is very natural for anyone with object-oriented design experience. Xcode provides application developers with templates that provide method prototypes to implement the protocols needed to manage the view.

The model is responsible for handling the data, including the logic for accessing or changing the data. The views responsibility is to display the data to the screen, by using labels, images, scrolling text or web views, etc. The view can be designed in IB (as described before), or programmatically. In between these two components is the controller, which is responsible for working with model data, transitioning between portions of the logic, modifying the view, and handling events such as touches and gestures.

It is useful to think of this paradigm as an artist with a palette and canvas. The artist takes material from the palette and puts it on the canvas in the desired way. This metaphor is identical to the MVC paradigm: the data is the palette, the controller is the artist, and the view is the canvas.

When a view controller is in control it acts at the top-level delegate object, or it can nominate another object to take its place. This lets the controller receive signals from the main application loop so it can detect events such as device rotation, scrolling or pinching gestures, while making necessary changes to the view or data model.

Two important aspects of view control regarding memory management, loading and unloading from a nib, are discussed in later sections.

IX. Displaying The Data

In *TouchWeather*, XML data is downloaded from the NOAA and parsed into a `WeatherData` object (the model part of the MVC paradigm). A `UITableViewController` [11] is created to display each time periods information. The `TableViewController` is then pushed onto the `UINavigationController`'s view stack in the same way the `RootViewController` was. `RootViewController` has a method `textFieldShouldReturn` that specifies the behavior of the view after the *return* button is touched in the text field. In our case, we want to initialize the `WeatherData` object using a zip code given by the user, create and initialize `TableViewController` with the data, and push the new controller onto the `UINavigationController`'s view stack:

```
- (BOOL)textFieldShouldReturn:(UITextField *)
    theTextField
{
        if (theTextField == input) {

                [input resignFirstResponder];

                WeatherTableViewController *wtvc =
                [self updateZipCode];

                [self.navigationController
                pushViewController:wtvc
                animated:YES];

                [wtvc release];
        }

        return YES;
}
```

This code causes the `input`'s keyboard to leave the screen with the *resignFirstResponder* message, then initializes a `WeatherTableViewController` with the *updateZipCode* method. This method implementation looks about like the following code listing.

```
- (WeatherTableViewController *) updateZipCode
{
        //error code checking omitted for brevity...

        WeatherData *weather = [[WeatherData alloc]
        initWithZipCode:string];

        WeatherTableViewController *wtvc =
        [[WeatherTableViewController alloc]
        initWithWeather:weather];

        [weather release];
        return wtvc;
}
```

The method allocates and initializes a `WeatherData` object, `weather`, using the zip code string instance variable. It then allocates and initializes the `WeatherTableViewController` using `weather`, sends a release message to `weather`, and returns the controller.

With the controller allocated and initialized, all that is left is to push it onto the `UINavigationController`'s stack and send it a release message. At this point, the table view controller slides into view and populates itself. It makes more sense to explain the `UITableViewController` subclass, how it populates itself, and how it expects to access data before describing the model generated from the XML data.

A. *WeatherTableViewController*

TouchWeather uses a custom `UITableViewController` subclass called `WeatherTableViewController`. The interface file WeatherTableViewController.h looks like:

```
@interface WeatherTableViewController :
    UITableViewController {

        WeatherData *data; //data model
}


- (id)initWithWeather:(WeatherData *)incomingData;
- (void)dealloc;

@property (nonatomic, retain) WeatherData *data;

@end
```

To make the `UITableViewController` subclass, go to File⟹New File and select a `UIViewController` subclass. Make sure the `UITableViewController` subclass check box is selected, and follow the prompts to finish making the view controller.

Xcode provides a template, as usual. The template has method prototypes to manage table view cells (a small view designed to be one entry in a table), and implement the `UITableViewDataSource` protocol. [12] The controller needs to provide three methods to implement the `UITableViewDataSource` protocol: `numberOfSectionsInTableView`, `numberOfRowsInSection` and `cellForRowAtIndexPath`. The `numberOfSectionsInTableView` method returns the number of cell sections in the table, the `numberOfRowsInSection` method returns the number of rows in a given section of the table, and `cellForRowAtIndexPath` returns a table cell for the table at a given index, which corresponds to the row of the cell.

The only method worth discussing in detail is `cellForRowAtIndexPath`. The controller has two instance variables that come into play here: `tableView` and `indexPath`. [13] The `tableView` variable is like the canvas: it provides slots for the cells to reside in. These slots are filled by the controller's implicit calls to `cellForRowAtIndexPath`, which include `tableView` and `indexPath` as method parameters:

```
- (UITableViewCell *)
  tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{

    NSString *CellID = [[NSString alloc]
    initWithFormat:@"WeatherCell %d",
    [indexPath indexAtPosition:1]];

    //try to find existing cell
    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellID];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleSubtitle
        reuseIdentifier:CellID] autorelease];
    }

    //set content here...

    return cell;
}
```

The method uses the `indexPath` argument to determine the row by sending the *indexAtPosition* message with the argument of 1. With the row number found, the method can make a unique cell identification, called the CellID. The controller sends the `tableView` a *dequeueReusableCellWithIdentifier* message with the CellID as the argument. This message potentially returns a cell that has been marked for reuse, thus reducing cell allocation overhead. If the cell returned is null, the controller allocates and initializes the cell with a style and a reuse identifier, and marks the cell to be autoreleased. At this point, the controller, which also holds the data model in *TouchWeather*, can use the `indexPath` argument to set the content of the cell. By using each row's unique row identifier, each call that returns a cell can return a cell populated with specific data.

After the `WeatherTableViewController` is initialized and each cell is configured, the view is complete and can be displayed to the user.

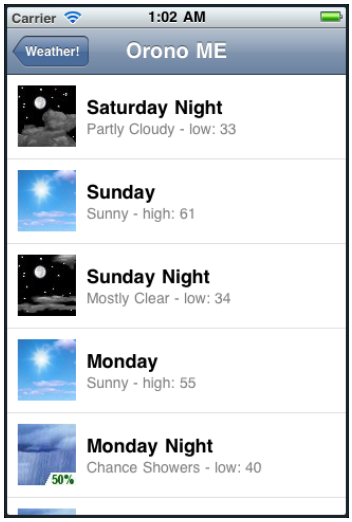

Fig. 3
TABLE WITH UINAVIGATIONCONTROLLER

At the top of the window is the navigation bar that is provided by the navigation controller. This has the label, which I have used for the city name and state in this view of my application, and a button that takes us back to the first view. The practical navigation provided by the few lines it takes to add the navigation controller was one of the simplest and most useful things I found while making this application.

X. NOAA's XML Data

TouchWeather gathers weather data from the National Oceanic and Atmospheric Association's (NOAA) XML feed. To build my data model, I used a library called TouchXML [14] to download XML data, which was parsed and organized in an array full of `NSMutableDictionary` objects. [15]

A. TouchXML

TouchXML is a libxml2 wrapper written in Objective-C. It is an easy-to-use DOM parser with XPath support that has good parsing speed, but with a rather bloated memory footprint. [16] It does not come with the SDK by default so in order to use it we have to get it. Make a directory called *ExtraLibraries* in the Developer directory. Download the TouchXML code at `https://github.com/TouchCode/TouchXML/tree/master/Source` and save it in the *ExtraLibraries* directory.

Once the source code is downloaded, we need to configure the project by telling Xcode where to find libxml2, and adding the TouchXLM classes to the project. With the project open in Xcode:

1. Go to Project⟹Edit Project Settings
2. Click the build tab.
3. Search for "Header Search Paths," and add /usr/include/libxml2
4. Search for "Other Linker Flags" and add -lxml2
5. Add the classes to the project by right clicking the Classes folder in the project window and go to Add⟹Existing Files... Navigate to the directory and highlight all of the source files, and click Add.
6. Add the line:

```
#import ''TouchXML.h''
```

to the header of the class the XML is being parsed in.

B. WeatherData

The data had to be organized in such a way to facilitate the `WeatherTableViewController`'s data access pattern, which as described before is based on the cell's row identifier. This naturally lends itself to an array, especially considering the rather small amount of data being downloaded. I chose to use an `NSMutableArray` [17] to store anywhere from 10 to 16 time periods of 12 hours each. I created an `NSMutableDictionary` [15] to store the data at each index of the array for each time period, and used unique string identifiers to access each of the data in the dictionary.

This array of dictionaries worked well. Each dictionary could be retrieved for the respective cell with ease by using the `NSIndextPath` object in the cell creation function of the `WeatherTableViewController`. The dictionary could also be passed to the next stage, the `TwoScrollViewController` and the `DetailViewController`.

In order to traverse to the next level of the view hierarchy, we need to decide what time period is selected by the user. Luckily, the `UITableViewController` subclass template provides us with a method to do such a thing.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
        //init the controller using the nib
        TwoViewController *twoViews =
        [[TwoViewController alloc]
        initWithNibName:@"TwoViewController" bundle:
             nil];

        //give the controller the correct dictionary
        [twoViews setWeatherData:[self.data
        getDictionary:[indexPath
        indexAtPosition:1]]];

        // push the new controller onto the stack
        [ self.navigationController
        pushViewController:twoViews animated:YES ];


        [twoViews release];
}
```

This method takes the same `NSIndexPath` argument, and thus can be used to access the appropriate data based on table row. Once the new controller is pushed onto the navigation controller's stack, the new controller's `viewDidLoad` method is called. This method has not been needed thus far. The `RootViewController`'s view was rather simple, and did not have any dynamic loading to be done and the `WeatherTableViewController`'s view was

created programmatically row by row. The `viewDidLoad` method is called when a view loads for the first time, and is used for dynamic customization. In our case, we want to use it to add content to each of the two views in the `TwoViewController`.

XI. Scroll Views

Because of the nature of our content, we will need to find a way to display data to the user that does not fit on the screen. A great way to do this is with a `UIScrollView`. [18] `UIScrollView`s are a useful way to display data to the user that does not fit on the screen and does not lend itself well to a `UITableViewController`, such as a large image or block of text.

TouchWeather uses a custom view controller subclass that contains two `UIScrollView`s called `TwoScrollViewController`. This view controller is loaded from a nib, and contains two scroll view laid out in Interface Builder, and as such, must be "wired" properly as described before with the `RootViewController` example.

With the two `UIScrollView`s laid out on the `TwoScrollViewController`'s view in IB and the code in the interface file, we can configure the content. We want the `TwoViewController` to initialize a `DetailViewController` to add as a subview to the top `UIScrollView`, and to tile three images downloaded from NOAA to add as a subview to the bottom `UIScrollView`.

The `DetailViewController` displays data in the dictionary using outlets configured in code that are linked to objects added to the view's nib file using Interface Builder. Simply create the nib file with the `UIViewController` subclass, lay out the objects needed to display the content, and link them to the `IBOutlet`s declared in the interface file. After allocating the `DetailViewController` and initializing it with the data to be displayed, you need to tell the scroll view how large the content to be displayed is:

```
sv1.contentSize = details.view.frame.size;
```

After that, add the `DetailViewController` to the `UIScrollView` as a subview and release the `DetailViewController` you allocated.

```
[sv1 addSubview:details];
[details release];
```

A. Tiling Content For Scroll Views

The bottom scroll view is more interesting. We want to tile three images downloaded from the NOAA that display hourly weather data for the given time period. To do this, the `TwoViewController` needs to specify the size of the tiled data in the `UIScrollView` by making a call to `CGRectMake`. [3] This function returns a `CGRect` object which can be used to specify a frame's size and position in a `UIScrollView` or any other view. In our case, we want to create a frame that encapsulates the three images tiled together. Before we can do that, we need to create the a frame for each of the images based on each other so they show up in the correct places once added to the `UIScrollView`.

```
//allocate the images
uiv1 = [[UIImageView alloc] initWithImage:
[data valueForKey:@"hourlytemp"]];
uiv2 = [[UIImageView alloc] initWithImage:
[data valueForKey:@"hourlywind"]];
uiv3 = [[UIImageView alloc] initWithImage:
[data valueForKey:@"hourlyprecip"]];

//set the image frames
uiv1.frame = CGRectMake(0.0,
0.0,
uiv1.frame.size.width,
uiv1.frame.size.height);

uiv2.frame = CGRectMake(0.0,
uiv1.frame.size.height,
uiv2.frame.size.width,
uiv2.frame.size.height);

uiv3.frame = CGRectMake(0.0,
uiv1.frame.size.height + uiv2.frame.size.height,
uiv3.frame.size.width,
uiv3.frame.size.height);
```

The parameters to `CGRectMake` in each call are *topLeftXCoord, topLeftYCoord, bottomRightXCoord, and bottomLeftYCoord,* respectively. The result is the first `UIImageView` starting at the very top left of the `UIScrollView`'s content view, the second `UIImageView` starting directly below the first, and the third directly below the second. After adding the line

```
scrollView2.pagingEnabled = YES;
```

`scrollView2` will snap to fit the content of each image as long as the size of the view in the controller's nib is the correct size. This will give the hourly weather graphs a clean feel by making it easy to view one graph at a time. The last couple things that need to be done are setting the overall content size of the bottom `UIScrollView`, and adding the `UIImageView`s as subviews.

```
//make this the size of all image frames
CGRect allHourlys = CGRectMake(0.0,
0.0,
uiv1.frame.size.width,
uiv3.frame.origin.y + uiv3.frame.size.height);

scrollView2.contentSize = allHourlys.size;

[scrollView2 addSubview:uiv1];
[scrollView2 addSubview:uiv2];
[scrollView2 addSubview:uiv3];
```

Once the `UIScrollView`'s content sizes are specified, the content views' frames are constructed, and the content is added to the `UIScrollView`s as subviews, the desired view is complete.

B. Memory Management: viewDidUnload

When a view's content is set with the `viewDidLoad` method, the best way to release any instance variables that were allocated is to implement the `viewDidUnload` method. The method prototype is already included in the template for a `UIViewController` subclass, so simply uncomment it and fill it in with release messages to the object's instance variables. When the view is unloaded from the screen (such as when the back button is hit on the `UINavigationController`, this method is called and the

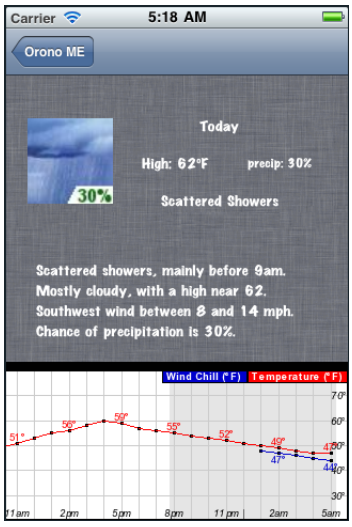

Fig. 4
UIScrollViews with content.

necessary objects are released as per the object ownership policy.

XII. Building the Application

There are a few options when it comes time to build and test your application. The easiest way is to select the desired build SDK (device or simulator), and *Build* button in Xcode. Building from the command line is also easy: simply change to the directory containing the project's .xcodeproj file and type:

```
$ xcodebuild -target Project_Name
```

XIII. Installing and Running on the iPhone Simulator

To simulate your application, set the Base SDK value to iOS Simulator 4.1 (or the most current version if that is what you are testing against). Once this is set, click the Build and Run button at the top of the Xcode project window. This will build and install the application on the simulator. The iPhone Simulator gives you a very good idea of what your application will look and feel like. The problem with relying on the simulator is that the iPhone has more limited hardware than a Mac. To really get the feel of your application in a real user setting, you must install the application on an iOS device to test it.

XIV. Installing on an iOS Device

Once the application is built, there are a couple of ways it can be installed on a device. If you possess an Apple developer key, the easiest way is to use Xcode. Go to Project⟹Project Settings. Change the base SDK to iOS Device 4.1 (or whatever iOS version you are developing for), and set the **Code Signing Identity** to your developer key. With the device connected to the computer, click Build and Go at the top of the Xcode project window. The application can also be installed from the command line by changing to the directory containing the project's .xcodeproj file and typing:

```
$ xcodebuild install -target Project_Name
```

to install the application on the connected device. Both of these methods of installing on the device require a valid key to sign the application. If you are short on cash, you can still install your application by jailbreaking your device.

XV. Installing an Application on a Jailbroken iOS Device

In order to install an application on a device without having a valid Apple developer key, you must jailbreak your device. Once the device is jailbroken, utility applications can be installed to transfer your application to the device. The first step is, of course, to jailbreak the device.

A. Jailbreaking Your Device

Jailbreaking is the process of gaining access to the root account on the iOS device. An easy and free way to do this is to use a program called *Greenpois0n*. By following the instructions in the README file provided with Greenpois0n, you can have the open application store *Cydia* installed on your iOS device and begin the rooting process. Once Cydia is installed, it will ask you how you will be using it (you should say developer), and it will prompt you to update (you should do a full update). Once the updates are complete, we need to add *ijailbreak.com's* source repository to get an application called *MobileTerminal* installed. This application is also available in the Featured section of Cydia, but as of the time this paper was written MobileTerminal from the "stock" Cydia repositories does not work on iOS 4.2.1.

1. Launch Cydia.
2. Touch Manage at the bottom.
3. Touch Sources in the middle.
4. Touch "Edit" at the top right on the navigation bar.
5. Touch "Add" at the top left on the navigation bar.
6. Type: "http://www.ijailbreak.com/repository" and touch "Add."
7. Once the repository is added, you can touch the *iJailBreak* entry in the Manage tab and install MobileTerminal from there.

Once MobileTerminal is installed, we want to change the root password. Open MobileTerminal and switch to the root user:

```
$ su root
```

The default root password is *alpine*. Enter that, and then change the root password:

```
$ passwd
```

MobileTerminal will prompt you for a new password twice. Type "exit" once you are finished to leave root.

B. Installing an Application on a Rooted Device

Once the root password is changed, we can safely install OpenSSH. Launch Cydia, and go to the Featured section. Scroll down to the section labeled "Console Utilities

& Daemons" and install OpenSSH from there by following the instructions presented.

With OpenSSH installed, you can now use SSH with your device, which we can used to transfer the application to the device by using SCP (or SFTP if you prefer).

Even though the device is rooted, we need to solve the problem of signing the application. Sign the application with a certificate created by KeychainAccess. [19] To create a certificate:

1. Launch Keychain Access in Applications/Utilities
2. In Keychain Access, go to Keychain Access⟹Certificate Assistant⟹Create a Certificate
3. For the name, enter Xcode Signature, and select Code Signing as the certificate type. Make sure the "Let me override defaults" box is checked, and click continue.
4. Enter a serial number. The number can be anything, as long as each certificate has a unique name and serial number. The default validity period is a year, but you can make it longer or shorter. Once the certificate expires you can no longer sign applications with it, but the applications built and installed using the expired certificate will still work on the device.
5. Enter some personal information.

The rest of the options work well by default, so click continue until the certificate is finished.

Next, we need to edit the info.plist in /Developer/Platforms/iPhoneOS.platform. Make a backup copy, and open info.plist in TextEdit, or an editor of your choice. Replace all instances of **XCiPhoneOSCodeSignContext** with **XCCodeSignContext** using the find-and-replace tool (there are many instances, so find-and-replace saves a lot of time). Save your changes.

With the info.plist file edited and the signature made, the next step is to configure and build in Xcode. With the project open, go to Project⟹Edit Project Settings and set the **Code Signing Identity** to **Xcode Signature**. One the project is configured, go to Build⟹Build. Accept the certificate, and ignore the warning about an invalid code signature since jailbreaking the device allows applications to be signed by something other than a valid key from the ADC.

To transfer the application to the device, open a terminal and change to the directory containing the project's .xcodeproj file where a **build** directory will appear once the project is built. Go to **build⟹Release-iphoneos**. Inside will be a directory called **Project_Name.app**. Type into the terminal:

```
$ scp -r Project_Name.app root@192.168.0.101:/
    Applications/
```

with your device's IP address in place of the one given. Type your password, and once the transfer is complete restart the device. The application will appear on the Springboard (the iPhone's desktop).

XVI. Final Thoughts

I enjoyed the challenges presented to me while developing *TouchWeather*. There was a bit of a learning curve in regards to Objective-C, but once that was overcome I could focus on the application's features and learning more about iOS development in general. I look forward to furthering my experience with mobile application development, not only with iOS but other platforms as well. By following these mobile computing trends, developers are not only targeting one of the largest consumer computing markets in the world, but are also targeting a big part of computing's future overall. If one trend has definitely been followed since the invention of the integrated circuit, it is that computers have gotten small, faster, and more user-friendly. This is continuing with the introduction of smart phones with multi-core processors, dedicated graphics chips, and constant connection to the internet through mobile network technologies. As these technologies continue to develop and become accepted on an even wider scale, designing and implementing applications that users want to use every day is going to become and even greater, more exciting challenge.

XVII. Resources

1. National Oceanic and Atmospheric Administration: `http://www.weather.gov/`
2. Greenpois0n: `http://greenpois0n.com/`
3. iJailBreak: `http://ijailbreak.com`

References

[1] Apple Inc., "Cocoa Touch Layer," November 2010, `https://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1`.

[2] Apple Inc., "UIKit Framework Reference," November 2010, `https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIKit_Framework/_index.html#//apple_ref/doc/uid/TP40006955`.

[3] Jonathan Zdziarski, *The iPhone SDK*, O'Reilly, 2009.

[4] Apple Inc., "UINavigationController Class Reference," May 2009, `http://developer.apple.com/library/ios/#documentation/uikit/reference/UINavigationController_Class/Reference/Reference.html`.

[5] Apple Inc., "UIViewController Class Reference," January 2011, `http://developer.apple.com/library/ios/#documentation/uikit/reference/UIViewController_Class/Reference/Reference.html`.

[6] Fraser Speirs, "What are XIB Files?," December 2007, `http://speirs.org/blog/2007/12/5/what-are-xib-files.html`.

[7] Apple Inc., "UITextField Class Reference," April 2010, `http://developer.apple.com/library/ios/#documentation/uikit/reference/UITextField_Class/Reference/UITextField.html`.

[8] Apple Inc., "Declared Properties," December 2010, `http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocProperties.html#//apple_ref/doc/uid/TP30001163-CH17`.

[9] Apple Inc., "Practical Memory Management," December 2010, `http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW1`.

[10] Apple Inc., "Autorelease Pools," December 2010, `http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmAutoreleasePools.html#//apple_ref/doc/uid/20000047-CJBFBEDI`.

[11] Apple Inc., "UITableViewController Class Reference," February 2010, `http://developer.apple.com/library/ios/`

#documentation/uikit/reference/UITableViewController_Class/Reference/Reference.html.
[12] Apple Inc., "UITableViewDataSource Protocol Reference," May 2010, http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDataSource_Protocol/Reference/Reference.html.
[13] Apple Inc., "NSIndexPath Class Reference," March 2011, http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSIndexPath_Class/Reference/Reference.html.
[14] Matt Tuzzolo, "Adding Local Weather Conditions to Your App," Sept. 2010, http://tinyurl.com/6ho62t6.
[15] Apple Inc., "NSMutableDictionary Class Reference," August 2010, http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableDictionary_Class/Reference/Reference.html.
[16] Ray Wenderlich, "How To Choose The Best XML Parser For Your iPhone Project," March 2010, http://tinyurl.com/y98kz9s.
[17] Apple Inc., "NSMutableArray Class Reference," August 2010, http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html.
[18] Apple Inc., "UIScrollView Class Reference," November 2010, http://developer.apple.com/library/ios/#documentation/uikit/reference/UIScrollView_Class/Reference/UIScrollView.html.
[19] kaamaru, "How to Fake Code Sign Applications in Xcode 3.2.3," August 2010, http://ihackmyi.com/forum/index.php?topic=24020.0.