CS 4768 Project Report

Submitted By: Osede Onodenalore, Samuel Ash, Jeff Conway

Commute Buddy

Instructor: Dr. Yuanzhu Chen

April 11, 2017

Contents

[Overview 3](#_Toc479708221)

[User Guide 3](#_Toc479708222)

[First Tab Description 4](#_Toc479708223)

[Second Tab Description 8](#_Toc479708224)

[Third Tab Description 10](#_Toc479708225)

[Interface 14](#_Toc479708226)

[First Tab 14](#_Toc479708227)

[Second Tab 15](#_Toc479708228)

[Third Tab 16](#_Toc479708229)

# Overview

For our project, we created a walking app called Commute Buddy. It is a three-tabbed application that is used for walking metrics, mapping your current location and displaying weather information. When launched, the app first asks you for permission to use your current location, and for permission to use core motion data.

The first tab displays all walking metrics such as session time, the number of steps taken, the distance traveled in meters, the average number of steps per second, your current, average and maximum speed in meters per second and kilometers per hour. There is a convert button that toggles between these two speed units. There is also a reset button to reset the current session and set all these values mentioned back to zero. It does this all by using the CoreMotion and CLLocation framework.

The second tab is a map that shows your current location using the geolocation feature from the CLLocation framework. The MapKit framework is used as well to display the map. The application will also draw a line corresponding to everywhere that the user has traveled while using the application. This was done using reference from the BreadCrumb library.

The third tab displays your current weather information. It does this by using again the CLLocation framework. In addition, we are also using a weather API called OpenWeatherMap (openweathermap.org). This tap displays information such as the current temperature, the daily high and low temperature, the humidity, the air pressure, the wind and direction, and snowfall and rainfall totals. We are accessing this information by using the coordinates for latitude and longitude from the CLLocation framework.

# User Guide

After launching the app, you will be brought to the first view tab. This tab will display all your movement information since launching the app. The values are displayed in meters per second format by default, but you can click the "km/h" button to convert it to kilometers per hour. Once you have clicked this button once, its label will change to "m/s" and can be clicked again to convert back to meters per second. If you wish to reset the values and start over, you can click the "reset" button. You can click the map icon at the bottom middle of the screen to navigate to the second tab, or the weather icon at the bottom right of the screen to navigate to the third tab.

The second tab will display your current location on a map as well as the path you have traveled. You can double tap the screen to zoom in slightly on a location. While having two fingers on the screen, bring your fingers closer together to zoom out, move them farther apart to zoom in, or move them up or down (together) to change the angle of the map. Towards the bottom of the screen is a selector (segmented control) that can be used to change how the map is displayed. The first, and default, option is a standard few that shows a basic map with labels for roads and some landmark locations. The second option shows a satellite view of the map. The third option is a hybrid view that shows a satellite view with labels for roads and some landmark locations. You can click the icon of a man waking at the bottom left of the screen to navigate to the first tab, or click the weather icon at the bottom right of the screen to navigate to the third tab.

The third tab will display weather information about your current location. You can click the icon of a man waking at the bottom left of the screen to navigate to the first tab, or click the map icon at the bottom middle of the screen to navigate to the second tab.

# First Tab Description

The first tab is used to display useful information about travel speed and information to the user. The view itself is a simple UIView and uses a basic UIViewController. Each shade of green in the view is displayed using a UILabel, with colors generated using an online color scheme generator (http://paletton.com/#uid=1350u0kllllaFw0g0qFqFg0w0aF). On top of each of these UILabels are other UILabels that are used to display static text and dynamically changing fields. Finally, at the very bottom are two UIButtons, one that is used to reset the values of all dynamic fields, and the other used to convert the speed (max/current/average) from meters per second to kilometers per hour.

The ViewController.h file for this view imports the CoreMotion and CoreLocation frameworks. It uses <CLLocationManagerDelegate>, which is required when using CLLocationManager. It also declares some instance variables:

NSNumber \*maxRecordedSpeed

int timeTick;

int minutes;

NSTimer \*timer;

These variables are used to keep track of the max speed the user has traveled since last reset or starting the app, as well as variables to keep track of how long the app has been gathering information. Finally, it creates the CMPedometer and a CLLocationManager that will be used in the view, and links all the storyboard elements.

The ViewController.m file contains all the code related to the functionality of the first tab. The first function, as usual, is (void)viewDidLoad. Within this function the base operations of the view begin and variables are set up. First, the timer variables are set to zero:

timeTick = 0;

minutes = 0;

self.maxRecordedSpeed = 0;

kmh = false;

Next, the CMPedometer and CLLocation manager are allocated and initialize and told to begin tracking user movement:

self.pedometer = [[CMPedometer alloc] init];

[self.pedometer startPedometerUpdatesFromDate:[NSDate date] withHandler:^(CMPedometerData \*\_Nullable data, NSError \*err) {

[self updateData: data];

}];

locationManager =[[CLLocationManager alloc]init];

locationManager.delegate = self;

locationManager.desiredAccuracy = kCLLocationAccuracyBest;

locationManager.distanceFilter = kCLDistanceFilterNone;

[locationManager requestWhenInUseAuthorization];

[locationManager startMonitoringSignificantLocationChanges];

[locationManager startUpdatingLocation];

Finally, the timer is set to start recording:

[self startTimer];

The next function (after the required/default (void)didReceiveMemoryWarning function) is the function that is called when we started the CMPedometer:

(void)updateData:(CMPedometerData \*)data

Within this function, checks are made to determine if data user data is available. The data we chose to display from the CMPedometer generated data are steps taken, distance traveled, average speed, and steps per second (cadence). If these values are available they are displayed in their corresponding UILabels, otherwise, the UILabels are set to "N/A" (not available).

Next is the function that is used (and required) by CLLoationManager to track the user's location:

(void) locationManager:(CLLocationManager \*)manager didUpdateLocations:(NSArray \*)locations

The first thing this function does is get the most update location for the user's device from the locations array that is receives from CoreLocation. It then displays the speed the device is traveling and displays it in the corresponding UILabel. Finally, it checks if the current speed is greater than the max recorded speed. If it is greater. it sets the max speed, to the current speed. It then displays the max speed in the corresponding UILabel.

To properly implement CLLocationManager, the following method must be used to catch any errors:

(void)locationManager:(CLLocationManager \*)manager didFailWithError:(NSError \*)error

This function simply catches errors and, if an error is found, stops the CLLocationManager from updating

Now we implement the functions for the timer. First we define the function that was called at the end of viewDidLoad:

(void)startTimer

This function simply declares the timer and calls the handler function:

(void) myTicker

Within this function the timer is calculated and displayed. First, it increments the timeTick variable, which is used to keep track of seconds, since we declared out timer with an interval of 1.0. If this value is less than 10, it is simply formatted so it does not have a zero in front of it and displayed. Then there is a check to see if the seconds (timeTick) value has reached 60 (one minute). If it has, the seconds are reset to zero and the minute variable is incremented by one. The minute value is then displayed in the appropriate UILabel.

The final two functions are used to handle the buttons being pressed. First, we have the function for the reset button:

(IBAction)resetButton:(id)sender

This function simply resets the values being displayed to and redeclares the CMPedometer to start tracking again to clear the total steps taken.

The last function is used to handle the conversion button:

(IBAction)convertButton:(id)sender

This is used to convert the values from meters per second to kilometers per hour by multiplying the current values 3.6 and displaying the results. It also sets a Boolean value, kmh, to true or false. When true, the values to be displayed on update are in km/h, and when false, the updates will revert to m/s.

There were several challenges associated with the implementation of this tab. The first was a limitation of CMPedometer, in that it is not possible to get live (step by step) updates. We had to settle for updating several times a minute. Another challenge, which we failed to overcome in the time allowed, was to keep CMPedometer recording data while in the background. This was since up until very recently, it seems, CoreMotion was entirely unable to do so. After much research, we were unable to find a solution that could be implemented. While this feature is not included in this version, we are confident that given more time we could accomplish this goal.

# Second Tab Description

The second tab is used to display a map of the user's current location. The map offers three choices of style:

* Standard, which is the standard map view, with no satellite imagery,
* Satellite, which is a purely satellite view with no details about street names, or landmarks
* Hybrid, which is a satellite image with street names and landmarks included.

The user may also tap "my location" in the top right hand corner to zoom in (or out) on their current location. Besides being just a simple map, however, the map also generates a red path on top of the map overlay to show the user the path they have recently taken (since the app was opened).

The ViewController.h file for this view imports the MapKit and CoreLocation frameworks. It uses <MKMapViewDelegate, CLLocationManagerDelegate>, which are required when using MapKit and CLLocationManager. It declares references to the map view and the custom classes CrumbPath and CrumbPathRenderer, which are classes included in Apple's "BreadCrumb" demo application. These classes will be discussed in detail below.

The ViewController.m file contains all the code related to the functionality of the second tab. The first function, as usual, is (void)viewDidLoad. Within this function the base operations of the view begin and variables are set up. First, the map is set to show the user location:

self.mapView.showsUserLocation = YES;

Then, if the app is not authorized when in use to use location data already, the map view will request to do so:

if ([[[UIDevice currentDevice] systemVersion] floatValue] >= 8.0 && [CLLocationManager authorizationStatus] != kCLAuthorizationStatusAuthorizedWhenInUse)

{

[locationManager requestWhenInUseAuthorization]; }

After viewDidLoad is didUpdateLocations, which includes sending updates to the CrumbPath and CrumbPathRenderer. If this is the first time a location has been updated, initialize the CrumbPath:

\_crumbs = [[CrumbPath alloc] initWithCenterCoordinate:location.coordinate];

[self.mapView addOverlay:self.crumbs level:MKOverlayLevelAboveRoads];

As you can see, CrumbPath is an extension of MKOverlay. The CrumbPath is a collection of points to draw along a polyline on top of the map. This allow the user to see their recent path. Note that the drawing is only done if the user moves 10m or more from their previous location at update time.

If this is a subsequent location update, the point is added to the collection of points in CrumbPath, if it is 10m or more from the previous location:

[self.crumbPathRenderer setNeedsDisplayInMapRect:updateRect];

The next method of interest is a function called by the segmented button to choose the map type:

case 0:

self.mapView.mapType = MKMapTypeStandard;

break;

For each segment (0, 1, 2) the map type is changed depending on which segment is currently selected. The next method is zoomToCurrentLocation, which will zoom, in or out until the device's width display 725m across, on the user's current location. It is called when the button "My Location" is tapped:

float spanX = 0.00725;

float spanY = 0.00725;

MKCoordinateRegion region;

region.center.latitude = coordinate.latitude;

region.center.longitude = coordinate.longitude;

region.span.latitudeDelta = spanX;

region.span.longitudeDelta = spanY;

[self.mapView setRegion:region animated:YES];

The MKCoordinateRegion region is the 725mx725m zoom on the user's location.

This tab also presented our group with some challenges. While we attempted to draw the user's path through our own methods, it became clear that this was going to be a challenge. Initially, we attempted to use a Google Maps API, and a file known as<GoogleMaps/GoogleMaps.h>, which included some drawing methods to achieve this goal. However, this proved to be difficult and buggy, and we explored alternatives. Around this time, we discovered Breadcrumb, which is an Apple Developer app which may be downloaded and tested for free. We ended up borrowing some methods from Breadcrumb to achieve our drawing goal, as the time constraint for our project became an issue, and we did not have much choice.

# Third Tab Description

To keep get the weather information, we are getting the current latitude and longitude of the location from the instance of the CLLocationManger. We then use CLGeocoder to reverse-geocode and get the city name based on the coordinates and pass the coordinates as variables to a method to get the weather information.

The view itself is a simple UIView and uses a basic UIViewController. The background is set to a shade of blue like a clear sky. A UIImageView is used to display the current weather icon, and UILabels are used to display the weather information.

The ViewController.h file for this view imports <CLLocationManagerDelegate> which is required when using CLLocationManager. It then links all the storyboard elements

The ViewContoller.m file contains all the code related to the functionality of the third tab. First, it declares some instance variables that will be used in the view:

CLLocationManager \*locationManager;

CLLocation \*currentLocation;

NSString\* apiKey;

int currentDay;

LocationManger is a CLLocationManager that is used to generate the user's location. currentLocation is used to store the user's current location. the apiKey variable will store the API key from openweathermap.org that is required to get information from the site. The key is free upon registering an account on the site.

The first function, as usual, in this view is (void)viewDidLoad. Within this function, we start by assigning the API key:

apiKey = @"9ee3e4133c207d8258520dbdff88ec66";

Then, we allocate and initialize the locationManager:

locationManager = [[CLLocationManager alloc] init];

locationManager.delegate = self;

[locationManager requestWhenInUseAuthorization];

locationManager.desiredAccuracy = kCLLocationAccuracyBest;

[locationManager startUpdatingLocation];

The next function is the handler function for the location manager:

(void)locationManager:(CLLocationManager \*)manager didUpdateLocations:(nonnull NSArray<CLLocation \*> \*)locations

Then, it gets the current location of the user, and checks if it is nil. If it is not nil, it gets the latitude and longitude of the user. Next, a CLGeocoder is allocated and initialized to get the user's current city. Finally, the function that will gather the weather information is called using the current latitude and longitude:

[self getWeatherAtLocation:lat Lon:lon];

Now comes the function is the required error catching function used by CLLocationManager:

(void)locationManager:(CLLocationManager \*)manager didFailWithError:(NSError \*)error

It simply checks for errors displays an alert if an error is found. This brings us to the function that is used to construct the URL that will be used to gather the weather information for the current latitude and longitude:

(void)getWeatherAtLocation:(NSString \*)lat Lon:(NSString \*)lon

This function constructs a URL using set strings and appends the latitude, longitude, and API key. It then calls the function that will get the data from the URL, which is also the next function in the code:

[self getDataFromURL:url];

(void)getDataFromURL:(NSURL \*)url

The first thing this function does is call the downloadData function within the AppDelegate.m file:

[AppDelegate downloadData:url withCompletionHandler:^(NSData \*data)

This will get all the relevant data from the URL and store as JSON data. Next, it converts the data to a dictionary so we can get values:

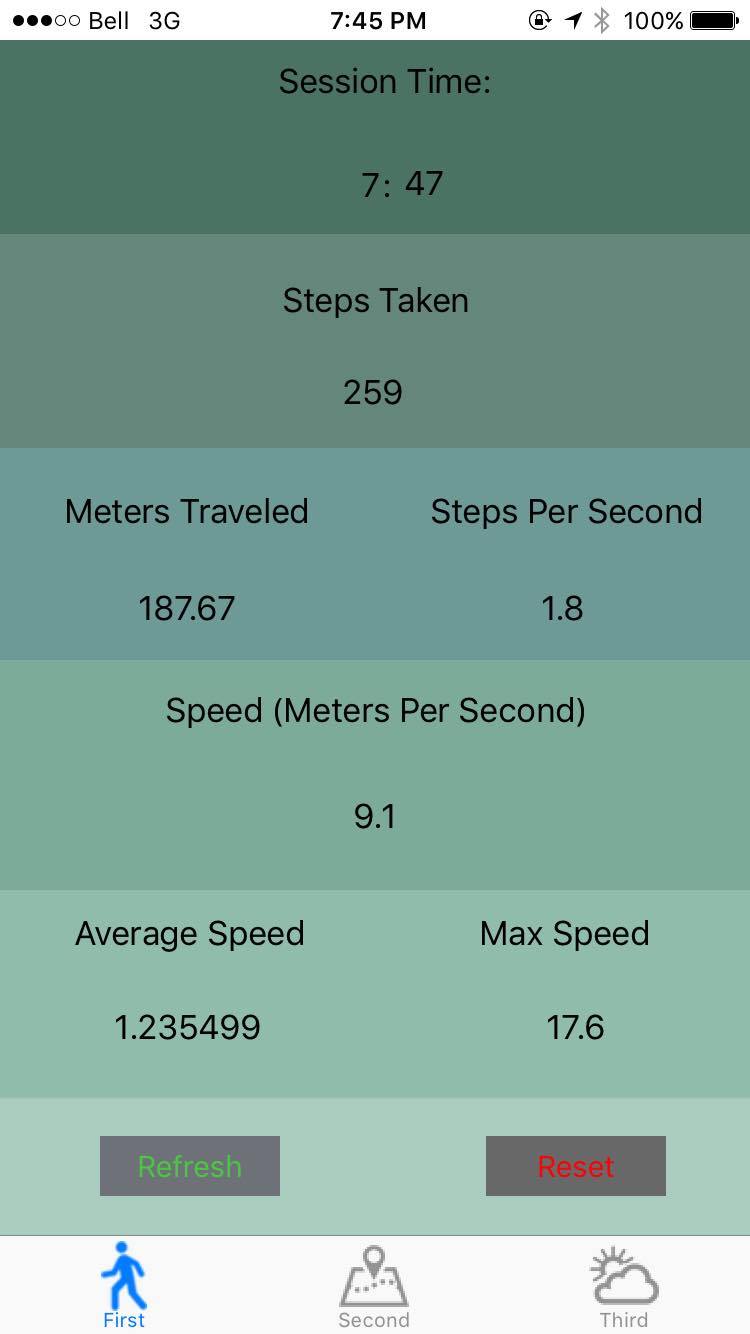
NSMutableDictionary \*returnedDict = [NSJSONSerialization JSONObjectWithData:data options:kNilOptions error:&err];

The rest of this function pulls values from the dictionary, formats them, and displays them.

Concerning the third tab, we did endure a couple of challenges. Initially, what we found challenging was the conversion of JSON data from the NSMutuableDictionary to specific objective-c variables. Another challenge was understanding weather data from the API. For example, the weather API gives data points called “rain” and “snow”. Testing the application, we realized that these values were consistently zero. After doing some research, we found out that these are precipitation values for the previous three hours as opposed to current precipitation. Another challenge we faced was with another data point for wind direction. When getting the JSON data, the value given was in degrees. We had to convert this numeric value to a corresponding compass direction. For example, 0 degrees would be the compass direction North(N). To do this, we had to write an if statement checking the ranges for each compass direction.

# Interface

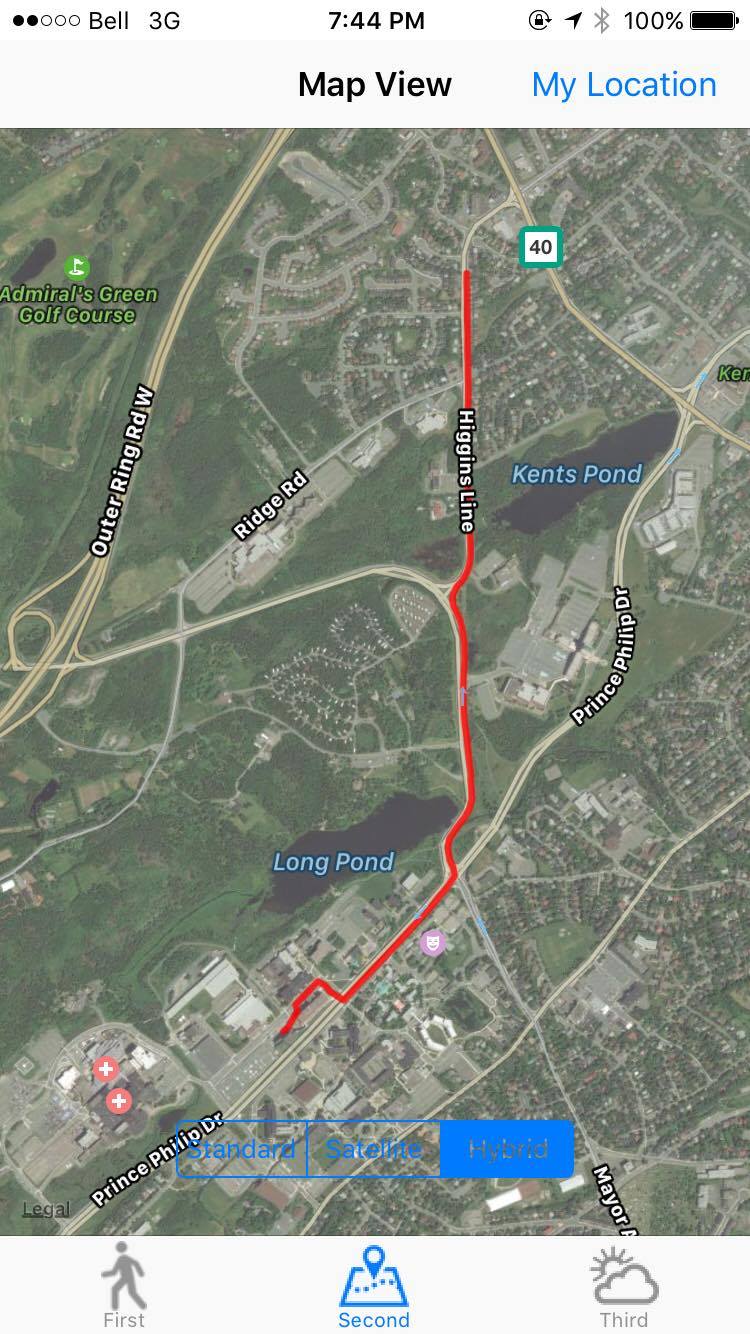
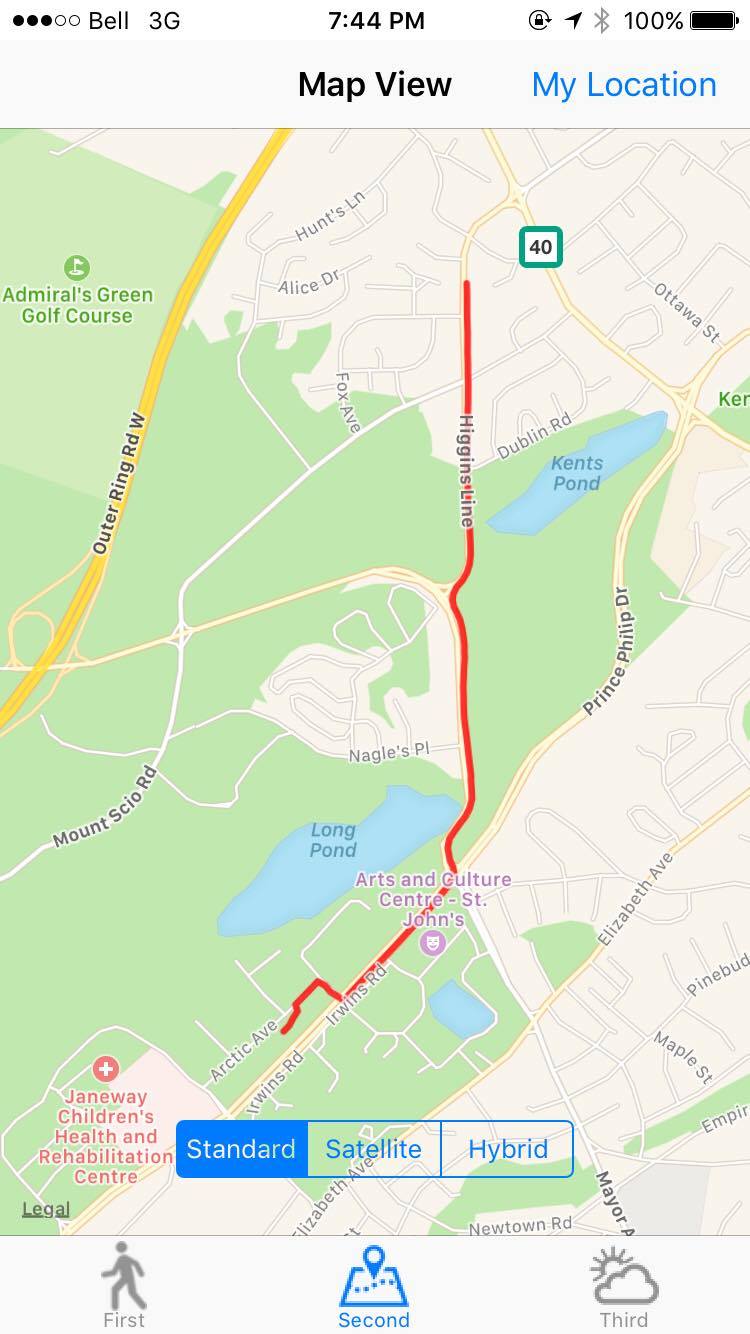
## First Tab



**Figure 1**

For the first tab, shown in **Figure 1**, the main storyboard has multiple UILabels for the corresponding metrics. This screenshot was an earlier version of our application. We know have removed the refresh button as the data constantly updates by itself. This screenshot was also taken with before we implemented proper constraints. We have also added a new conversion button which now converts speed values from meters per second to kilometers per hour. These three tab titles called “First”, “Second” and “Third” are now called “Movement”, “Map” and “Weather”. The reset button resets all numeric UILabels to 0.

## Second Tab



**Figure 2**

From the second tab, shown in **Figure 2**, this tab is an MKMapView which shows your current location. On top of that, the application is drawing a line from where you started using the app to your current location. As you can see, there are three display types that are usable. Standard View, Satellite View and Hybrid view you have the option to select which view you want to see at any time while using the application. Shown in **Figure 2** are Standard and Hybrid view. Satellite view is Hybrid view without displaying streets or other landmarks. At the top, there is a button called “My Location” which zooms in such that you’re the width of your phone represents 725 meters.

## Third Tab



**Figure 3**

From the third tab, shown in **Figure 3**, like the first tab, there are multiple UILabels for the corresponding weather data that was retrieved. We are also using a UIImageView to display an icon based on the current weather. These images are stored in the xcassets file. We obtained these images from the OpenWeatherApp website.