Unique Issues and Considerations in App Design for Mobiles that are not Applicable When Designing for Laptops and Desktops

Anurag Kashyap, Sujay Mishra
MIT College of Engineering & Management,Hamirpur, H.P.
E-mail: *anurag_kash1000@rediffmail.com*

Abstract
Development of apps for mobile has many similarities with development of apps for other platforms. However, the differences are so unique that developers must draw their attention to them, as the traditional development platforms don't present them at all. For example, mobile or other handheld devices are not run on the same operating systems, the mobile screens have limited power supply, a smaller screen, and environmental sensors that are not seen in laptop and desktop computers.

In this paper, apart from the differences between laptops and mobile design, the differences between iOS and Android machines that have an impact on app design are also discussed.

Keywords: *App Design, Android, iOS*

INTRODUCTION

Designing for the specific device our app will run on is extremely important! Applications that work well on a traditional computer may be complete disasters if ported to a mobile platform without redesigning the logic to fit the device's capabilities. Additionally, the capabilities of the device enable us to design an application that can do different things than an application on a traditional computer. Apps are cheap and easy to obtain. If ours doesn't work well, there is likely to be an acceptable alternative. A well-designed app can be a delight to use. A poorly designed app will not be used for long, if at all. The operating system, device size, and mobility all impact design and must be accounted for.

OPERATING SYSTEM DESIGN ISSUES

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

The primary technical difference between mobile device operating systems and operating systems used on laptop and desktop computers is that the mobile operating system is not a true multitasking system. On mobile devices, only one app can be active at a time. When another app is started, or the app is interrupted by another app (for example, a phone call), the app that was running gets put in the background. It remains in the background until the user specifically accesses it again. If it remains in the background too long, or if available memory gets too low, the operating system may kill it. This back-and-forth between different states is called the app's life cycle. Both Android and iOS apps have a life cycle. The life cycle is based on the user's interaction with the app and the operating system's need for memory and processing resources. As users interact with the device, they may switch between apps or different views within a single app. When this happens the app goes through different states, requiring the developer to handle this switch so that users don't lose data or get unnecessarily interrupted in the task they were performing. This makes understanding, and designing for, the app life cycle extremely important to the successful app developer.

ANDROID LIFE CYCLE

To understand the Android life cycle it is useful to first understand the states that an Android app user experiences. When users touch an app's icon, the app is started and becomes visible to the users. While the app is visible, the users can interact with it. This is considered the Resumed or running state. As the users interact with the app, they may be interrupted with a pop-up window, or they may be distracted and not touch the screen for a period of time. If the users stop interacting for a period of time, the app will fade but still be partially visible. In either of these two cases the app enters the Paused state. If the users close the pop-up or touch the screen, the app becomes fully visible again, and the app again enters the Resumed state. If users don't touch the screen for a longer period of time and the screen goes black, or the user starts another app so that the original app is no longer visible, the app enters the Stopped state. If users turn on the screen or use the Back button to get back to the app, the app again enters the Resumed state. An app can remain in the Stopped state for quite some time. However, if the device is rebooted or a user runs a number of other apps before coming back to the original app, that app can be destroyed by the operating system to free up resources for other apps that the

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

user is actually interacting with. To design an app that functions well given this pattern of use, developers must understand what happens as the app enters and leaves these states, as well as what they should design the app to do in those instances. This requires understanding the Android life cycle.

The Android life cycle (see Figure 1) begins when a user touches an app's icon. This action causes the *onCreate* method in the app's initial activity to execute. This method includes code to load the screen (called a layout) associated with the initial activity to load. The developer needs to place code in this method that initializes variables and layout objects to the settings required for the user to begin interacting with the app. After the activity has been created, the *onStart* method is executed. This method does not have to be implemented but is useful if the app requires certain settings to be the same for every time the app starts, whether it is an initial start after the activity is created or restarted after the activity is brought back from a stopped (but not destroyed) state. After the activity has started, the *onResume* method is executed. This method also does not have to be implemented but is very useful to return the app to the running state that the app was in before it paused. This includes turning on system services used by the app (for example the GPS or the camera), restarting animations, and any other settings needed to allow users to pick up where they left off.

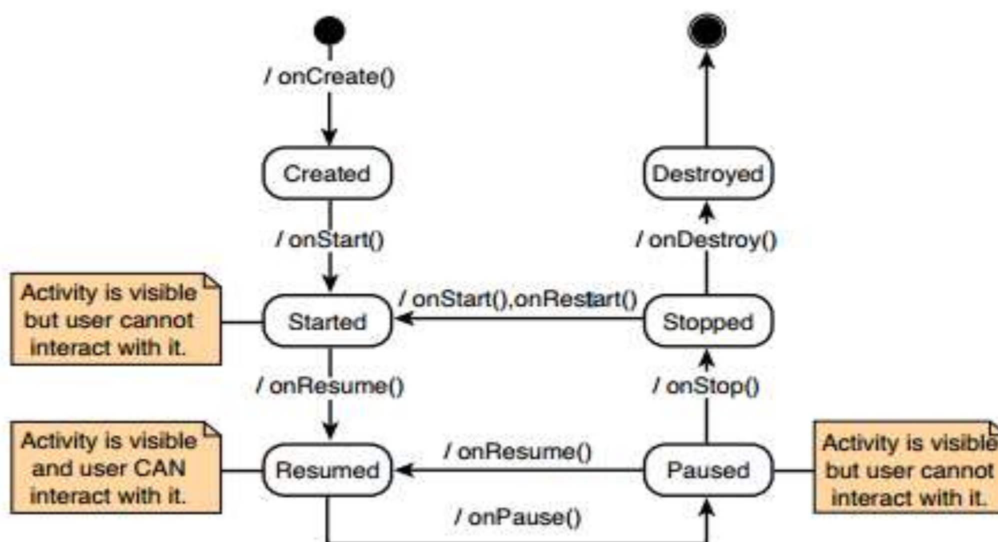

Figure 1 *Android life cycle*

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

When a user stops directly interacting with the app, the path to destruction begins. None of the methods executed on the path to destruction have to be implemented. However, they often serve a useful purpose and should be considered. The first method executed is onPause.

This method should be used to stop services that the app is using, to stop animations, or to store important state information so that users can start using the app exactly as they left it. If the app is about to become invisible, the onStop method will be executed.

This method should make sure important data is permanently stored so that as system resources are consumed by other apps, they are not lost. Finally, if not restarted, the onDestroy method will be executed just before the operating system takes away all the app's resources. This is our last chance to capture important data before all is lost.

IOS LIFE CYCLE

The life cycle for iOS is similar to Android's. However, iOS uses both an app life cycle and a screen (called view) life cycle to accomplish essentially the same things. As with Android, the life cycle (see Figure 2) begins when the user taps an app's icon. The *application: didFinishLaunchingWithOptions:* method is similar to an activity's onCreate method. However, in iOS this method is used to set up the operating environment for the complete app, not just a single activity.

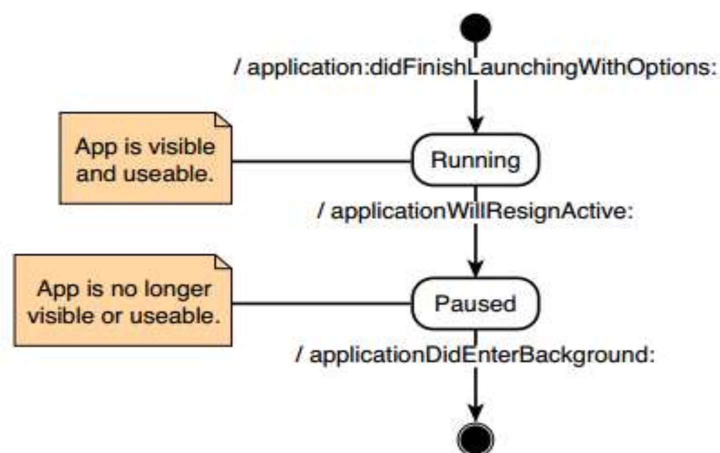

Figure 2 iOS App life cycle.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

The *applicationWillResignActive*: method is executed when the app is interrupted, similar to when the *onPause* method is executed in Android. Finally, when the app is no longer visible, the *applicationDidEnterBackground:* method is executed. As with Android, code in these methods should be used to turn off services and save important data for the user before it's potentially lost.

Unlike Android, iOS has a separate life cycle for displayed screens (called ViewControllers). The view life cycle (see Figure 3) begins after the application has finished loading or the user goes to a different page in the app. After the view is loaded into memory, the *viewDidLoad:* method executes. This method is executed only once if the view stays in memory. We should write code in this method to set the initial state of the view. After the view has loaded into memory, just before the view is visible to the user, the *viewWillAppear:* method is executed. Code in this method should be used to load any data into the views that will be visible to the user and turn on services that the user needs to interact with the app. This method executes every time the view reappears on the device.

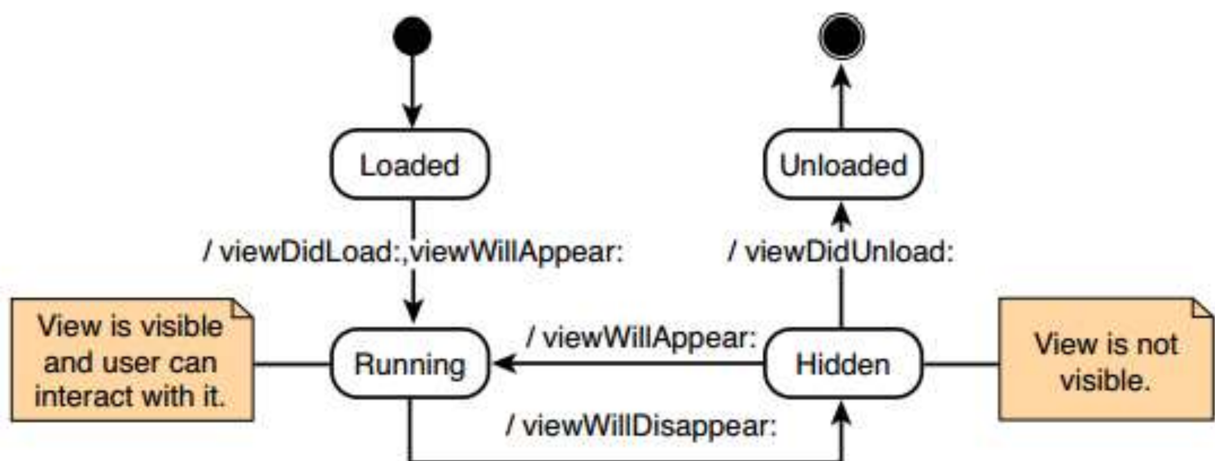

Figure 3 iOS View life cycle.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

Just like in Android, if an app is interrupted, or the user doesn't interact with the device for a period of time, or the user moves to another view in the app, the view is pushed into the background. Just before this happens, the *viewWillDisappear:* method is executed. Code in this method should turn off services and take steps to save the user's data. If the user doesn't interact with the view for a period of time while it is in the background, iOS may reclaim its resources. Just before the view is released from memory, the viewDidUnload: method is executed. This is the developer's last chance to preserve important data used in the view.

Understanding and properly coding the app to take advantage of the methods associated with the life cycle are important to ensuring a good user experience with our app. Take the time to understand these life cycles, and our app development experience will be significantly less frustrating!

SCREEN SIZE AND ORIENTATION ISSUES

The most obvious difference between mobile and traditional application design is the amount of real estate we have to work with. The mobile device has significantly less area to design the interaction that our users can experience with our app. Poor interface design is the easiest way to get bad reviews for our app. Mobile devices are also used in different situations than traditional computing devices are. App users are often multitasking (walking, talking with friends, and so on). The app design must allow users to switch to our app and do what they want to do right away, before they are distracted again. If users can't easily figure out how to use the app, no amount of help will satisfy them. This is no different from traditional development. However, the very limited screen real estate makes it a significant challenge. In addition, the focus among app developers has been on very good user interface design, so the competition is fierce for apps that work really well.

In response to the limited screen size, both iOS and Android have the capability to scroll to interface elements not on the screen. Scrolling can be both horizontal and vertical. However, both scrolling capabilities should be used judiciously, especially horizontal scrolling. Scrolling down a list has become a natural action on both traditional computers and mobile devices. However, horizontal scrolling has not. Horizontal scrolling should be

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

reserved for use for elements that start on the main screen and extend off the screen. Users won't naturally think to horizontally scroll to look for items they can't find on the main screen. Even vertical scrolling should be limited. Lists are obvious choices for vertical scrolling, but other types of interface elements should be limited. Additionally, when scrolling, we must also fix certain elements so that the user can perform needed operations without scrolling back through the entire contents of the screen.

The obvious answer to the limited screen size is to carefully plan the user's interaction with our app. Screens should focus on one, or a very limited and coherent, set of tasks that the user can or would want to do. Navigation should be planned and designed so that it is obvious to the user how to proceed to the next task. If a task requires multiple steps, those steps should be designed as distinct screens, and the user should be guided through the screens needed to complete the whole task.

Although screen size is a nontrivial design issue, the fact that by default, a screen's orientation can change as the user turns the device also presents design issues. When the user turns the device from vertical to horizontal orientation, the layout or view reorganizes to that orientation. This significantly changes the amount of vertical and horizontal real estate for our interface. Interface elements that were obvious to the user in the vertical orientation may become inaccessible in the horizontal orientation. Again, this can be a very frustrating experience for our user, unless we carefully plan for the layout in both orientations, and thoroughly test it as well. Scrolling can be implemented to alleviate some of the problems associated with orientation change. However, simply adding scrolling may not solve the user experience issues. If we cannot make it work in an alternative orientation, as a last resort we can code the app to work in only one orientation.

The solutions to these screen size and orientation issues are planning and design. What does the user want to do with our app? What can our user do with our app? What are the logical steps needed to accomplish those tasks given the device limitations? These are the types of questions we must answer to design a successful app.

CONNECTIVITY ISSUES

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

One of the most important aspects of mobile devices is that they are able to communicate with other devices and the Internet. This enables the capability to create very powerful and useful apps. However, this also poses design problems. The device's capability to connect can be lost, or the connection speed may be very slow. Additionally, these problems can arise if the device moves even a few feet. Compounding the problem is that users may not recognize or even understand that there is a connectivity problem while they are using our app. Apple requires that all apps submitted to the app store include a user warning when the network connection is lost, but this doesn't address slow speeds, and is not required by Android at all.

Again, design and planning are our solutions.

The primary issue that the app developer has to be concerned with is blocking the user from working with our app. When the app gets or sends data, it can take a significant amount of time. Users are unlikely to be happy waiting for this action to complete before doing other tasks. This means we have to plan for uploading and downloading data asynchronously, which means we have to make it run outside the main thread of the app. It also means that the rest of the app should be designed to provide other things users can do unless the data is absolutely necessary for the task. If a user tries to do something that requires the data, provide a warning. The warning should provide enough information to help users decide what they should do next. If there is no connection or if there is a weak signal, tell them and give them options.

Uploading important data is also a concern. As with a download, uploads should be performed asynchronously. We need to check that the upload was completed fully so that if a connection is lost during the upload, the user's data is not corrupted. This means that the data should be cached locally until it is successfully uploaded. Finally, we may need to provide functionality to upload the data when a good connection becomes available.

Communication problems external to the app can impact our app's performance.wemust plan for this possibility to provide the best user experience possible.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

BATTERY ISSUES

Mobile devices are just that—mobile. This means that they are not always connected to a power source. They rely on batteries for their power, and batteries can be drained. Our job as a developer is to not drain those batteries unnecessarily. This is not just a courtesy issue. If every time our app is used the user's device quickly becomes a brick, it will be noticed. An app that quickly drains power will not get used, will get bad reviews, and eventually will not get downloaded at all.

The primary power draw for devices is the display. We cannot do much about that except to make sure that our code is efficient and doesn't take an unnecessary amount of time to complete the work that the user wants to do. Also, we should make sure that users can pick up where they left off if the app is interrupted so the screen doesn't need to be on so long.

After the screen, the primary power drains are the sensors. Global Positioning System (GPS), camera, communication, and other sensors are all big power draws. Fortunately, it is within our power to control these things. We control access to device hardware within our app and should turn on these capabilities only just before the user needs them. we should also turn them off as soon as the user completes the task that requires these items.

The app's life cycle plays an important role here. If the app is interrupted, all device access or use should be suspended immediately. When the app is about to become active, turn on as late as possible only those device capabilities needed. For example, in the previously mentioned app that uses weather data, the weather retrieval is started as one app activity becomes active. If the weather data is successfully retrieved, it is time stamped. The next time the activity becomes active, the weather data will be retrieved only if it is outdated, thus saving battery power.

Some battery issues are beyond our control. We cannot make an app that extends battery power. However, we can definitely make an app that significantly reduces battery life. Be sure to plan for battery use when designing our app.

HARDWARE ISSUES

A very cool aspect of mobile computing is the set of hardware components available on the device. Many devices have the capability to locate the device within a few meters using the GPS, have sensors that can capture device orientation, have lights that can be turned on and off, have

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

cameras, and have other hardware components that allow the device to interact with the environment. Access to these components can make fun and useful apps. However, employing them within our app is not without potential problems. The battery issue was discussed in the previous section, and this is always a concern when using hardware devices. However, each component has its own set of issues that, when used poorly, can make an app less desirable.

The first issue to be aware of is availability of the component. Different manufacturers make Android devices, and some include devices that others do not. iOS devices are generally more homogeneous, but differences still exist. Because of this, it is very important to consider how important the component is to the primary functionality of our app. If it is only tangential, we may want to consider not using it because using it will often prevent the app from be loaded onto the device. At best, the absence of the hardware component on the user's phone or tablet will cause frustration with our app. Another concern is situational availability.

For example, for a device to get a GPS signal, the device has to have the capability to get the satellite signal required for operation. If the user is indoors, the GPS may not work.

A second issue to be aware of is time delays. To access a hardware component we must use the component's Application Program Interface (API). The component may take some time to turn on and respond with the information we need. If this delay is significant, it may impact the user experience in such a way that our app is viewed negatively. For example, the GPS system takes time to acquire enough satellite signals to accurately locate our device. This could take more than a minute. Stopping app function until this happens should be avoided if possible. If the user is left waiting for the device to respond, the screen may time out. This issue may be encountered even if we did everything properly and turned off the services when the app is about to be sent to the background, and then turned them on again when the user re-opens the app. However, if the activation of the device takes time, our app will end up hanging every time it returns from the background.

This vicious circle will not please the user. The proper solution to this particular problem is to use a separate execution thread to do the initialization, thus allowing the user to interact with other

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

parts of our app while the services are being activated.

A final important issue with the use of hardware devices is accuracy. There are several aspects of this issue. First, the accuracy of the component can differ among manufacturers. Consider what the minimal level of accuracy is needed for effective use of our app, and design for that.

Be sure to give the user options if the required level of accuracy is not available. Second, accuracy often takes time. For example, to find the location of the device within a few hundred meters is often very quick. However, accuracy of a few feet often takes much more time. What is the required level of accuracy for our app's functionality? What can the user do if the device cannot achieve this? How quick does the acquisition of location need to be? All are important considerations when we are designing the app. A very good design strategy when we need better accuracy is to keep the user informed of progress. The Google Maps app provides an example of this. When finding our location on the map, the app first shows a big blue circle that gets progressively smaller as the accuracy improves. Finally, how the device returns data to the app may impact the level of accuracy our app can access. Again, using GPS as an example, the number of digits reported for the latitude and longitude coordinates dictate the level of accuracy of those coordinates. In some cases, the number of digits reported can differ. This is primarily an issue for the Android platform because it can differ among versions of the Android OS.

DEVICE DIFFERENCES

Android devices (phones and tablets) and iOS phones and tablets each have a unique set of hardware and software capabilities that make the way the user interacts with the device different for each. Again, to fully capture the device's capabilities and not degrade the user experience, we must design for those unique characteristics. Remember, users can and will do things we are not expecting. Even if it makes no sense to you, they will do it! If the app crashes or loses important data because of something they did, it does not matter: IT IS our FAULT! Plan accordingly.

ANDROID

Android devices originally used four hardware buttons (see Figure 4) to support the user's use of the device. These buttons were the Home button, the Menu button, the Search button, and the Back button.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

The user could press any of these buttons at any time during use of our app, which would impact the functioning of our app. The Home and Back buttons worked independently of our code, whereas the Menu and Search buttons provided functionality only if our app was specifically coded to use these buttons.

Figure 4 Android hardware buttons

However, more recent Android devices (running Android 3.0, API 11 and greater) have replaced these buttons with virtual buttons at the bottom of the screen and an action bar at the top of the screen (Figures 5 and 6 , respectively). The Back and Home virtual buttons remain the same in both form and function. However, the Menu and Search buttons were eliminated, and a Recents virtual button was added. The Recents button shows the user's recently used apps. The action bar displays the app's icon and title and the menu. Menu items will be displayed with an icon (if defined). If there are too many menu items to be displayed with an icon, the extra menu items are accessible through the three vertical dots on the right side of the menu bar. If an app is targeting older versions of Android as well as newer, the action bar presents only the three dots at the far right. When pressed, these dots perform the same function as the Menu button.

Figure 5 Android virtual buttons.

Figure 6 Android action bar

The Home button immediately moves our app to the Stopped state. This causes the onPause and onStop methods to execute. It will not destroy the app unless it needs the system resources. This means we must pay attention to these events even though we may not be anticipating this behavior when our app is in use.

The Back button immediately goes back one action or activity. This can have several implications for our code. For example, if the user is looking at an activity and presses the Back button, the visible activity will be immediately moved to the Stopped state (causing the *onPause* and *onStop* methods to execute). It will

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

move the previous activity into the Running state. This will cause the onStart and *onResume* methods to execute for that activity. If our activity has displayed a pop-up, the activity is currently in the Paused state (because it is partially visible). Pressing the Back button will hide the pop-up and cause the *onResume* method to execute, and our activity will be placed in the running state. If the soft keyboard is displayed, our app is also in the Paused state. Pressing the Back button will hide the keyboard and again put our activity in the Running state.

If the user presses the Menu button, our app will not do anything unless we have specifically programmed it to have a menu. Menus can be useful ways to provide the user with access to functionality that is not used as the normal course of events in using our app; thus, we don't want to waste valuable screen real estate to provide access to that functionality.

Finally, the Search button also does nothing unless we code it. We can use this button to allow the user to search for information within our app.

The hardware/virtual buttons provided by Android devices either have an impact on our app or can be used to extend the functionality of our app. In either case it is important to plan for the impact of these buttons when designing our app.

iOS

The primary hardware button of concern on iOS devices is the Home button. This button immediately moves any app presently running to the background. The *viewWillDisappear:*, *applicationWillResignActive:*, and *applicationDidEnterBackground:* methods will all be called. Plan our app so that this action will not cause problems.

INTRODUCING OUR FIRST APP

To learn both Android and iOS design and development, we will build the same app on each platform. Building the same app on both platforms is useful for understanding differences and similarities between the platforms. The app we will build is called MyContactList. Building a contact list app is a good way to learn mobile development for two reasons. First, its purpose and function is generally understood, so a significant part of any application development effort (understanding the functional requirements) does not need to be explained. Second, a contact list app requires utilizing many basic and

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

advanced features of mobile development; therefore, it is very useful in providing a context for learning these concepts.

The MyContactList app consists of four different screens. Each screen is used to illustrate basic app development concepts we will use in almost any subsequent app we develop. Additionally, you'll learn how to navigate between screens in an app.

CONTACT SCREEN

The contact screen shown in Figure 7is used to enter, edit, and save information about people in our contact list. While developing this screen, we learn some of the fundamental concepts of mobile user interface design and data entry. Later on, we use this screen as a way to learn how to create and store data in a database on a mobile platform. Finally, the contact screen shows us how to integrate hardware capabilities into an application by using the device's camera to capture a contact's picture and to make a phone call by tapping the contact's phone number

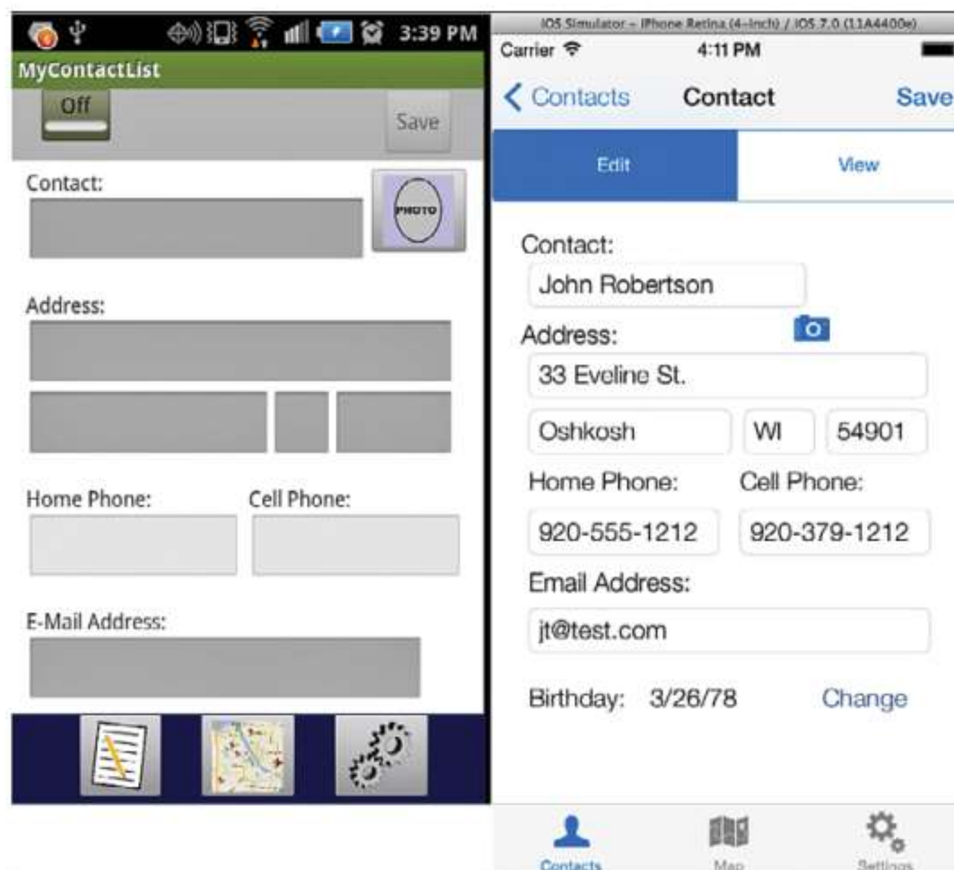

Figure 7 The Contact screen.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

CONTACT LIST SCREEN

The contact list (see Figure 8) is used to search for basic contact information and allow selection of a contact for further action (for example, editing and deleting). Lists are very important components of many apps on both Android and iOS. Developing this screen teaches us how to integrate them into any future app. This screen also demonstrates how to access information provided by hardware components of the device.

MAP SCREEN

The map screen (see Figure 9) is used to display the recorded location of a single contact or all our contacts on a map with a pin. The screen also demonstrates how to display the device's present location on the map and how to switch between different map views. The usefulness and importance of maps on mobile devices needs no further explanation. Through the development of this screen we learn how to integrate mapping into our apps. Additionally, the screen will be used to demonstrate another approach to accessing sensor information.

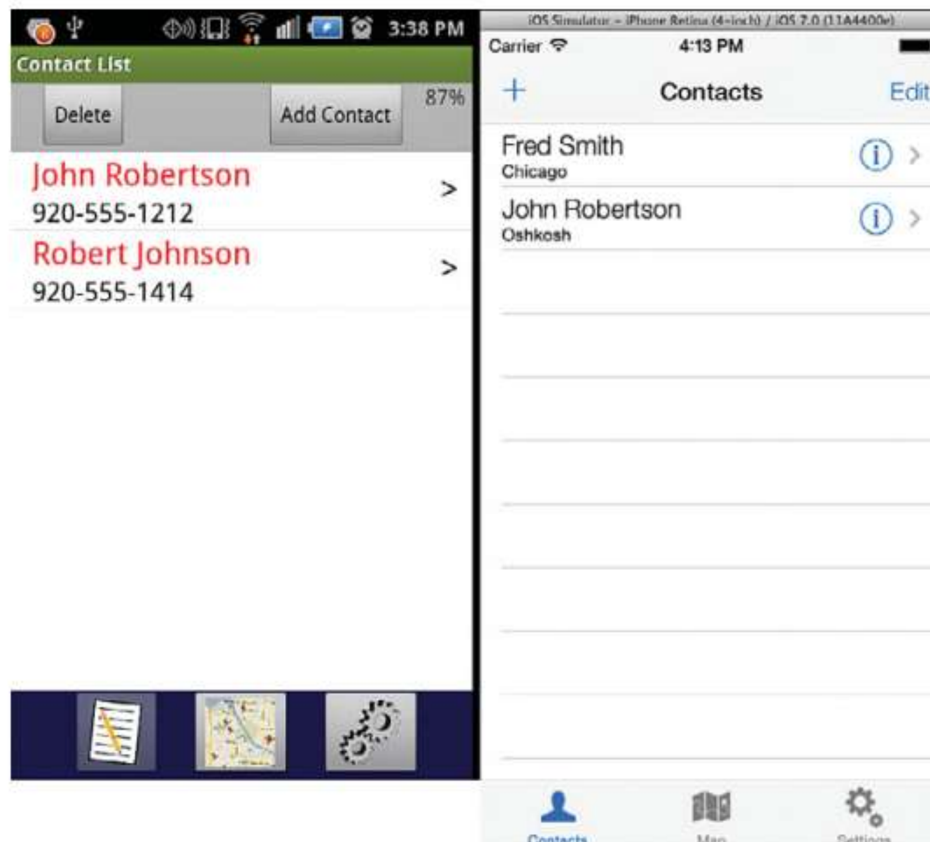

Figure 8 The Contact List screen.

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

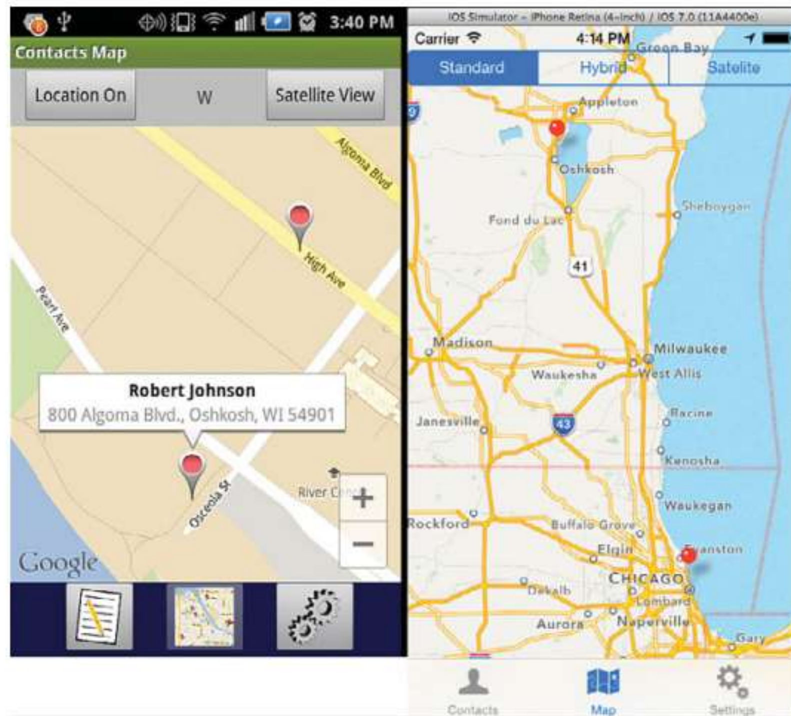

Figure 9 The Map screen.

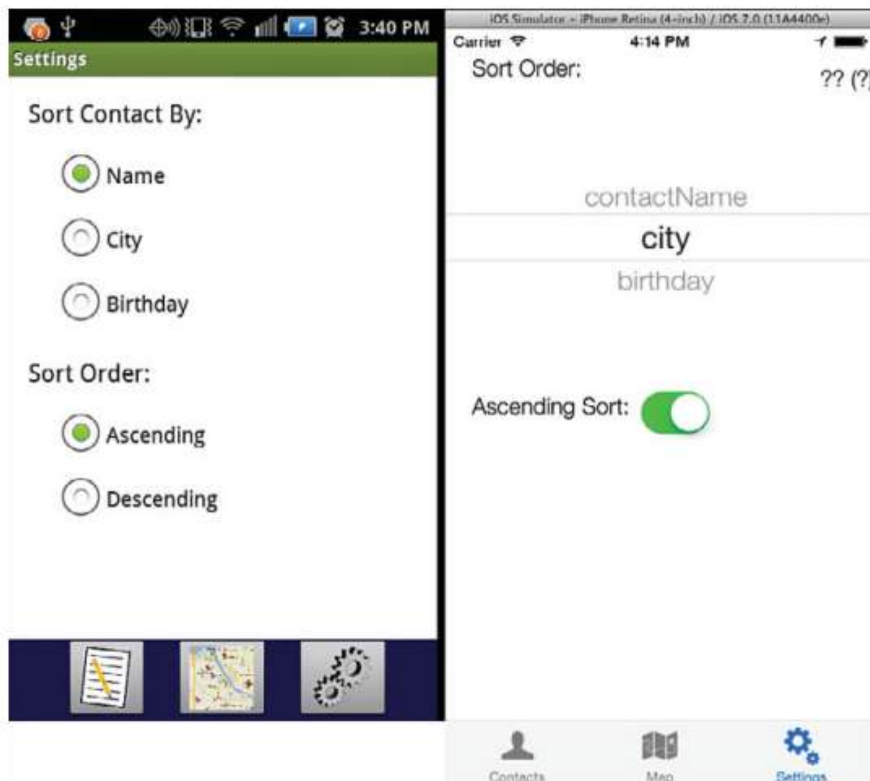

Figure 10 The Settings screen.

SETTINGS SCREEN

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

The Settings screen (see Figure 10) is used to set the sort order for the contacts in the Contact List. In developing this screen, we learn to use a method of data persistence designed for capturing and storing individual pieces of data. This type of data persistence is often used to capture user preferences for an app. we also learn to use a different type of display widget (view).

CONCLUSION

App development is different from traditional software development. we must design to take advantage of, and be aware of, the impact of the mobile operating system and the hardware that the app is running on. If we do not design our app to account for these differences in the device, we will ensure that our app does not get much use.

Android and iOS devices have many similarities and differences that require planning when we are developing an app that will run on both device families. To learn both platforms and learn the differences between them, we will develop the same app for both platforms in the next two sections of the book.

REFERENCES

1. Learning Mobile App Development, A Hands-on Guide to Building Apps with iOS and Android, pages 13 to 26, by Jakob Iversen and Michael Eierman

2. GAME AND GRAPHICS PROGRAMMING FOR IOS AND ANDROID® WITH OPENGL® ES 2.0, pages 2 to 8, by Romain Marucchi-Foino

3. Android™ Open Accessory Programming with Arduino™, pages 4 to 10, by Andreas Göransson and David Cuartielles Ruiz

4. Flash Mobile Developing Android & iOS Applications (2011), by Matthew David

5. Android Application Testing Guide, by Diego Torres Milano

6. Android vs. iOS: Comparing the Development Process of the GQueues Mobile Apps, http://blog.gqueues.com/2013/

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved

07/android-vs-ios-comparing-development.html

7. Android vs. iOS: Comparing the Development Process of the GQueues Mobile Apps, 1st ed. 2011 Edition, by Jeff Friesen and Dave Smith

8. Hello, Android: Introducing Google's Mobile Development Platform (Pragmatic Programmers), By: Ed Burnette

9. Beginning Android Games, By: Mario Zechner

10. Android Programming: The Big Nerd Ranch Guide (Big Nerd Ranch Guides), By: Bill Philips & Brian Hardy

Page 32-49 © MANTECH PUBLIATIONS 2016. All Rights Reserved