Middle-East Journal of Scientific Research 20 (11): 1644-1650, 2014
ISSN 1990-9233

DOI: 10.5829/idosi.mejsr.2014.20.11.1948

Android Devices Integrate with the User Interface

[1]K.P. Kaliyamurthie and [2]D. Parameswari

[1]Departmnet of CSE, Bharath University, Chennai-600 073, India
[2]Department of Computer Applications,
Jerusalem College of Engg., Chennai-600 100, India

Abstract: This paper proposes an approach to developing guidelines for creating a mobile phone user interface (UI), specific to Android devices. The approach is based upon four basic components of an Android UI, namely Icons, Widgets, Activities and Menu. This paper is targeted towards providing an enhanced experience for the users of Android devices, by developing style guidelines specific to Android devices. These guidelines help the designer to develop a UI which is easy to use as well as visually appealing to the user.

Key words: Component % Icon % Widget % Activity % Intent % Menu % Context % Task

INTRODUCTION

Android, being an open-source platform provides manufacturers, the ability to customize the user interface of their particular devices. Though this helps the manufacturers to stylize and customize their interfaces in a unique way, it creates lot of inconsistencies among the devices with the same version of the Android operating System. Due to this, the Look and Feel of each device is different and hence the users face difficulty due to the lack of consistency of interface styles [1].

It is generally a time consuming process for the users of a mobile phone to get used to the user interface and navigate through the device. With changing layouts and designs for devices with even the same version of operating system yet different manufacturers, it creates a necessity for users to learn the user interface of each and every device individually due to lack of a standard layout for developing the user interface [2].

Interface style guideline is a document that assists a designer by providing him with optimization techniques on how to provide the user with a rich layout, thereby giving the user an enhanced usage experience. Even though there are various sets of guidelines available for developing mobile phone UIs, with the newly emerging technologies there are a lot of new components and actions that the user is allowed to perform in an interface, such as touch gestures, etc. This creates an urge to improve the existing guidelines to suit the trends of the current day. Android devices, becoming a popular technology recently, requires a standard set of style guides for itself, to support the components and layouts that this platform provides to the designers [3].

UI Guidelines are not definite rules that should be followed, but are mere suggestions that help the designer to optimize the layout and provide a user friendly environment to the device. Since this paper emphasizes on Android devices, the style guidelines specified in this paper would be specific to the same. The user interface of Android devices contains four major components which are the target area for user interactivity. Guidelines for designing each component shall be discussed individually in detail [4].

Guidelines for Icon Design: An icon is a graphic symbol that represents either an action that the user can perform or a link to navigate to another part of the application [5]. There are various types of icons available in the Android platform, such as launchers, menu, actionbar, statusbar, tabs, dialogs and listview.

Android operating system is meant to be used on a variety of devices that offer a range of screen sizes and resolutions. While designing icons for an application, it's important to keep in mind that the application may be

Corresponding Author: K.P. Kaliyamurthie, Departmnet of CSE, Bharath University, Chennai-600 073, India.

installed on any of the devices. Android platform enables the designer to provide icons in such a way that they will be displayed properly on any device, regardless of the device's screen size or resolution. In general, the recommended approach is to create a separate set of icons for each generalized screen density. Then, store them in density-specific resource directories in the application. When the application runs, the Android platform checks the characteristics of the device screen and loads the icons from the appropriate density-specific resource folder. Apart from supporting various display sizes and resolutions, here are a set of guidelines that aid in designing an icon for an application [6].

Use Common Naming Conventions for Icon Assets: The naming of icon files should be done in such a way that related assets will group together inside the directory when they are sorted alphabetically. In particular, it is helpful to use a common prefix for each icon type.

Asset	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons	ic_menu	ic_menu_archive.png

Use Vector Shapes Where Possible: It is possible to create icons as a combination of vector shapes and raster layers and effects with the use of image designing programs. But it is recommended to use vector shapes wherever possible, so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

Design with Large Canvases: Since there is a need to create assets for different screen densities, it is best to start your icon designs on large canvases with dimensions that are multiples of the target icon sizes [6]. For example, launcher icons are 96, 72, 48, or 36 pixels wide, depending on screen density. If you initially draw launcher icons on an 864x864 canvas, it will be easier and better to tweak the icons when you scale the canvas down to the target sizes for final asset creation.

Redraw Bitmap Layers During Scaling: If an image is scaled up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

Remove Unnecessary Metadata While Saving Image Assets: Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from the assets. There are various tools such as OptiPNG that can ensure that this metadata is removed and that your image asset file sizes are optimized.*F. Make sure that corresponding assets for different densities use the same filenames*

Corresponding icon asset files for each density must use the same filename, but be stored in density-specific resource directories. This allows the system to look up and load the proper resource according to the screen characteristics of the device. For this reason, make sure that the set of assets in each directory is consistent and that the files do not use density-specific suffixes [7].

Guidelines for Widget Design: App widgets (sometimes just "widgets") are a feature introduced in Android 1.5 and vastly improved in Android 3.0 and 3.1. A widget can display an application's most timely or otherwise relevant information at a glance, on a user's Home screen. The standard Android system image includes several widgets, including a widget for the Analog Clock, Music and other applications[7]. This paper describes how to design a widget so that it fits graphically with other widgets and with the other elements of the Android Home screen such as launcher icons and shortcuts. It also describes some standards for widget artwork.

Standard Widget Anatomy: Typical Android app widgets have three main components: A bounding box, a frame and the widget's graphical controls and other elements. App widgets can contain a subset of the View widgets in Android; supported controls include text labels, buttons and images. For a full list of available Views, see the Creating the App Widget Layout section in the Developer's Guide. Well-designed widgets leave some margins between the edges of the bounding box and the frame and padding between the inner edges of the frame and the widget's controls.

The table below provides a rough estimate of your widget's minimum dimensions, given the desired number of occupied grid cells:

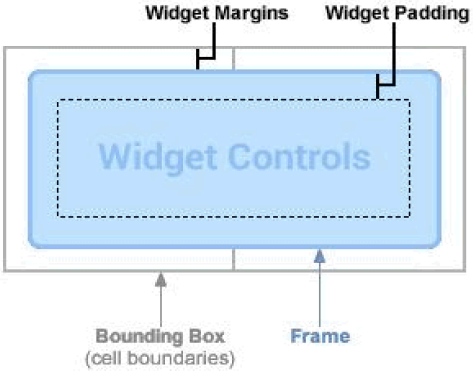

Fig. 2.1: Widgets generally have margins between the bounding box and frame and padding between the frame and *widget controls*.backgrounds and flexible layouts for app widgets will allow your widget to gracefully adapt to the device's Home screen grid and remain usable and aesthetically awesome.

No. of Cells (Columns or Rows)	Available Size (dp) (minWidth or minHeight)
1	40dp
2	110dp
…	…
n	70 × *n* – 30

Determining a Size for Your Widget: Each widget must define a minWidth and minHeight, indicating the minimum amount of space it should consume by default[8]. When users add a widget to their Home screen, it will generally occupy more than the minimum width and height you specify. Android Home screens offer users a grid of available spaces into which they can place widgets and icons. This grid can vary by device; for example, many handsets offer a 4x4 grid and tablets can offer a larger, 8x7 grid. When your widget is added, it will be stretched to occupy the minimum number of cells, horizontally and vertically, required to satisfy its minWidth and minHeight constraints. As we discuss in Designing Widget Layouts and Background Graphics below, using nine-patch

Resizable Widgets: Widgets can be resized horizontally and/or vertically as of Android 3.1, meaning that minWidth and minHeight effectively become the default size for the widget. You can specify the minimum widget size using minResizeWidth and minResizeHeight; these values should specify the size below which the widget would be illegible or otherwise unusable. This is generally a preferred feature for collection widgets such as those based on ListView or GridView.

Adding Margins to Your App Widget: Android 4.0 will automatically add small, standard margins to each edge of widgets on the Home screen, for applications that specify a target SDK Version of 14 or greater. This helps to visually balance the Home screen and thus we recommend that you do not add any extra margins outside of your app widget's background shape in Android 4.0.

Designing Widget Layouts and Background Graphics: Most widgets will have a solid background rectangle or rounded rectangle shape. It is a best practice to define this shape using nine patches; one for each screen density. Nine-patches can be created with a graphics editing program. This will allow the widget background shape to take up the entire available space [9]. The nine-patch should be edge-to-edge with no transparent pixels providing extra margins, saves for perhaps a few border pixels for subtle drop shadows or other subtle effects.

Guidelines for Designing Activities and Tasks: This section of the paper highlights the design decisions that are available to you while preparing activities for an application and the control they give you over the UI experience of your application.

Four fundamental concepts in the Android system that are to be understood to provide a good understanding of the flow of the application are Applications, Activities, Activity Stack and Intents.

The following are tips and guidelines for application designers and developers:*A. Use explicit intents when writing an activity that will not be reused and do not specify intent filters*.

If you're writing an activity that you don't want other activities to use, be sure not to add any intent filters to that activity. This applies to an activity that will be launched only from the application launcher or from other activities inside your application. Instead, just create intents specifying the explicit component to launch — that is, explicit intents. In this case, there's just no need for intent filters. Intent filters are published to all other applications, so if you make an intent filter, what you're doing is publishing access to your activity, which means you can cause unintentional security holes.

When Reusing an Activity, Handle the Case Where No Activity Matches: Applications can re-use activities made available from other applications. In doing so, you cannot presume your intent will always be resolved to a matching external activity — you must handle the case where no application installed on the device can handle the intent.

A test can be made to check if the activity matches the intent, which you can do before starting the activity, or catch an exception if starting the activity fails. To test whether an intent can be resolved, your code can query the package manager. The isIntentAvailable() helper method can be used to test when initializing the user interface. For instance, you could disable the user control that initiates the Intent object, or display a message to the user that lets them go to a location, such as Google Play, to download its application. In this way, your code can start the activity (using either startActivity() or startActivityForResult()) only if the intent has tested to resolve to an activity that is actually present

Consider How You Want Your Activities to Be Launched or Used by Other Applications: As a designer or developer, it's up to you to determine how users start your application and the activities in it. As an application is a set of activities, the user can start these activities from Home or from another application.

- Launch your main activity from an icon at Home
- Launch your activity from within another application
- Start an activity expecting a result
- Start an activity not expecting a result
- Launch your activity only from within another application
- Launch two or more main activities within a single application from separate icon at Home
- Making your application available as a widget

An application can also display a portion of itself as an app widget, embedded in Home or another application and receive periodic updates.

Allow Your Activities to Be Added to the Current Task: If your activities can be started from another application, allow them to be added to the current task (or an existing task it has an affinity with). Having activities added to a task enables the user to switch between a task that contains your activities and other tasks. Exceptions are your activities that have only one instance. For this behavior, your activity should have a launch mode of standard or singleTop rather than single Task or singleInstance. These modes also enable multiple instances of your activity to be run.

Notifications and App Widgets Should Provide Consistent Back Behavior: Notifications and app widgets are two common ways that a user can launch your app through something besides its main icon in Launcher. You must take care when implementing these so that the user has a consistent experience with the back button, not causing surprises in where they return to or the state the application ends up in.

The Handling Notifications section of the developer guide's Status Bar Notifications documentation provides an overview of how to write code to correctly handle notification. This applies equally to handling interactions with app widgets [4].

A notification always starts an activity as a new task (that is, it puts FLAG_ACTIVITY_NEW_TASK in the intent it passes to startActivity ()). This is done because interruptions to a task should not become part of that task.

Do Not Use Dialog Boxes in Place of Notifications: If your background service needs to notify a user, use the standard notification system. Do not use a dialog or toast to notify them. A dialog or toast would immediately take focus and interrupt the user, taking focus away from what they were doing: the user could be in the middle of typing text the moment the dialog appears and could accidentally act on the dialog. Users are used to dealing with notifications and can pull down the notification shade at their convenience to respond to your message.

Do Not Take over the Back Button Unless You Absolutely Need to: As a user navigates from one activity to the next, the system adds them to the activity stack. This forms a navigation history that is accessible with the Back button. Most activities are relatively limited in scope, with just one set of data, such as viewing a list of contacts, composing an email, or taking a photo. For example, Maps uses layers to present different information on a map to the user: displaying the location of a search result, displaying locations of friends and displaying a line for a street path providing direction between points. Maps store these layers in its own history so the Back button can return to a previous layer [2].

Guidelines for Designing Menu: A menu holds a set of commands (user actions) that are normally hidden and are accessible by a button, key, or gesture. Menu commands provide a means for performing operations and for navigating to other parts of your application or other applications. Menus are useful for freeing screen space, as an alternative to placing functionality and navigation, in buttons or other user controls in the content area of your application.

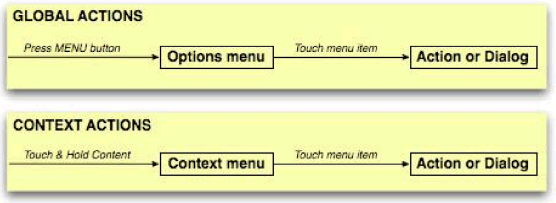

Fig. 5.1: Global actions Vs. Context actions

The Android system provides two types of menus you can use to provide functionality or navigation. Between them, you should be able to organize the functionality and navigation for your application. Briefly:

- The Options menu contains primary functionality that applies globally to the current activity or starts a related activity. It is typically invoked by a user pressing a hard button, often labeled Menu.
- The Context menu contains secondary functionality for the currently selected item. It is typically invoked by a user's touch & hold on an item. Like on the Options menu, the operation can run either in the current or another activity.

All but the simplest applications have menus. The system automatically lays the menus out and provides standard ways for users to access them. In this sense, they are familiar and dependable ways for users to access functionality across all applications. All menus are panels that "float" on top of the activity screen and are smaller than full screen, so that the application is still visible around its edges. This is a visual reminder that a menu is an intermediary operation that disappears once it's used.

Selecting the right kind of menu to present and using menus consistently, are critical factors in good application design. The following guidelines should assist user experience designers and application developers toward this end.

Separate Selection-specific Commands from Global Commands: Put any commands that are global to the current activity in the Options menu or place them fixed in an activity screen; put commands that apply to the current selection in the Context menu. (In any case, the command could either run as part of this activity or start another activity.) You can determine in which menu to place a command by what it operates on: If the command acts on selected content (or a particular location) on the screen, put the command in the Context menu for that content. If the command acts on no specific content or location, put it in the Options menu. This separation of commands is enforced by the system in the following way. When you press the Menu button to display the Options menu, the selected content becomes unselected and so cannot be operated on. For an explanation of why the content becomes unselected, see the article on Touch mode.

An example of a selection-specific Context menu is when a user performs a touch & hold on a person's name in a list view of a contacts application. The Context menu would typically contain commands "View contact", "Call contact" and "Edit contact".

Place the Most Frequently Used Operations First: Because of limited screen height, some menus may be scrollable, so it's important to place the most important commands so they can be viewed without scrolling. In the case of the Options menu, place the most frequently used operation on its icon menu; the user will have to select "More" to see the rest. It's also useful to place similar commands in the same location — for example; the Search icon might always be the first icon in the Options menu across several activities that offer search.

In a Context menu, the most intuitive command should be first, followed by commands in order of decreasing use, with the least used command at the bottom.

Don't Put Commands Only in a Context Menu: If a user can fully access your application without using Context menus, then it's designed properly! In general, if part of your application is inaccessible without using Context menus, then you need to duplicate those commands elsewhere.

Before opening a Context menu, it has no visual representation that identifies its presence (whereas the Options menu has the Menu button) and so is not particularly discoverable. Therefore, in general, a Context menu should duplicate commands found in the corresponding activity screen. For example, while it's useful to let the user call a phone number from a Context menu invoked by touch & hold on a name in a list of contacts, that operation should also be available by the user touching the phone number itself when viewing contact details.

The First Command in a Context Menu Should Be the Selection's Most Intuitive Command: Touching on an item in the content should activate the same command as touching the first item in the Context menu. Both cases should be the most intuitive operation for that item.

Selecting an Item in the Content Should Perform the Most Intuitive Operation: In your application, when the user touches any actionable text (such as a link or list item) or image (such as a photo icon), execute the operation most likely to be desired by the user. Some examples of primary operations:

- Selecting an image executes "View image"
- Selecting a media icon or filename executes "Play"

A Context Menu Should Identify the Selected Item: When a user does touch & hold on an item, the Context menu should contain the name of the selected item. Therefore, when creating a Context menu, be sure to include a title and the name of the selected item so that it's clear to the user what the context is. For example, if a user selects a contact "Joan of Arc", put that name in the title of the Context menu (using set Header Title). Likewise, a command to edit the contact should be called "Edit contact", not just "Edit".

Put Only the Most Important Commands Fixed on the Screen: By putting commands in menus, you free up the screen to hold more content. On the other hand, fixing commands in the content area of an activity makes them more prominent and easy to use.

Here are a number of important reasons to place commands fixed on the activity screen:

- To give a command the highest prominence, ensuring the command is obvious and won't be overlooked. For example, A "Buy" button in a store application.
- When quick access to the command is important and going to the menu would be tedious or slow. For example, Next/Previous buttons or Zoom In/Out buttons in an image viewing application.
- When in the middle of an operation that needs to be completed [8].

Use Short Names in the Options Icon Menu: If a text label in the Options icon menu is too long, the system truncates it in the middle. Thus, "Create Notification" is truncated to something like

"Create…ication". You have no control over this truncation, so the best bet is to keep the text short. In some versions of Android when the icon is highlighted by a navigation key such as a trackball, the entire descriptive text may be shown as a marquee where the words are readable as they scroll by [9].

A Dialog Should Not Have an Options Menu: When a dialog is displayed, pressing the Menu button should do nothing. This also holds true for activities that look like dialogs. A dialog box is recognizable by being smaller than full-screen, having zero to three buttons, is non-scrollable and possibly a list of selectable items that can include checkboxes or radio buttons. The rationale behind not having a menu is that when a dialog is displayed, the user is in the middle of a procedure and should not be allowed to start a new global task (which is what the Option menu provides).

If an Activity Has No Options Menu, Do Not Display a Message: When the user presses the Menu button, if there is no Options menu, the system currently does nothing. We recommend you do not perform any action (such as displaying a message). It's a better user experience for this behavior to be consistent across applications.

CONCLUSION

In this paper, we suggested various design tips for developing UI for applications of an Android device. Factors critical to UI designs were identified as general usability principles and UI components. These factors were combined to develop specific Guidelines for each UI component. The style guide considers various topics, including menus, UI components that are specific to Android platform and thus, this paper serves as a Standard for developing UIs for Android applications.

REFERENCES

1. Goyal, D., P.H.J. Chong, P. Shum, Y.C. Tong, X.Y. Wang, Y.X. Zuo and H.W. Kuek, XXXX. Design & Implementation Of User Interface For Mobile Devices.
2. Design Patterns For User Interface For Mobile Applications Erik G. Nilsson.
3. A Factor Combination Approach To Developing Style Guides For Mobile Phone User Interface Wonkyu Park, Sung H. Han, Sungjin Kang, Yong S. Park, Jaemin Chun
4. Http://Www.Mobile.Tutsplus.Com/
5. Kaliyamurthie, K.P., 2013. An Application Of Non-Uniform Cellular Automata For Efficient Cryptography, Indian Journal Of Science And Technology, 6(5): 4648-4652.

6. Kaliyamurthie, K.P., 2013. K-Anonymity Based Privacy Preserving For Data Collection In Wireless Sensor Networks, Indian Journal Of Science And Technology, 6(5): 4604-4614.
7. Kaliyamurthie, K.P., 2013. Highly Secured Online Voting System Over Network, Indian Journal Of Science And Technology, 6(6): 4831-4836.
8. Kumaravel, A., 2013. Vehnode: Wireless Sensor Network Platform For Automobile Pollution Control" Ieee Explore, pp: 963-966.
9. Kumaravel, A., 2013. Multi- Classification Approach For Detecting Network" Ieee Explore, pp: 1114-1117.