Android PC - Splash Brothers
Design Specifications

Contributors:
Zach Bair
Taronish Daruwalla
Joshua Duong
Anthony Nguyen

1. Technology background

The Android-x86 project has been in existence since 2011. Since then there has been a port of android that is able to run on PCs. While the initial port took a considerable amount of effort, the current project consists of updating the system with each subsequent Android version release. At this moment, a few auxiliary members have been creating small changes such as external mouse support.

1.1 The Android Architecture

The Android OS framework is layered in blocks from bottom to top. The OS starts its foundation with a linux kernel that comes with the necessary drivers to handle input/output devices, power management, etc. On top of the linux kernel is the native libraries written in C/C++, the android runtime libraries and the dalvik virtual machine. The next layer is the application framework, and on top of that, the applications.

1.1.1 The Linux Kernel

The Android OS is built on top of a linux kernel, with all the necessary drivers for input/output controls, power management, etc. Even though the OS is built from a linux kernel, Android is not linux, in that it does not have a native windowing system, no GNU libc (glibc) support, and does not include the entire set of standard linux utilities. Also, the Android OS comes with other custom drivers specific to the Android platform.

The reason the developers chose to use the Linux kernel is because it has great process management, as well as a long-standing security model. Also, the linux kernel has good driver support and is open source.

1.1.2 The Native Libraries

On top of the linux kernel are the native libraries written in C and C++ such as function libraries, hardware abstraction libraries and the native servers. These libraries provide low-level

functionalities and computationally-intensive functionalities to the Android platform. The native library also includes bionic libc, which is essentially a custom glibc built to be efficient on embedded systems, and small enough to not affect RAM performance. The functional libraries do all of the intensive work of the android platform, and are abstracted in higher level APIs. The native servers handle all of the interactions with the input/output devices and provide them to the android platform. The hardware abstraction libraries provide an abstraction between the hardware and the higher level APIs for applications. Bionic libc is not based off of glibc, so bionic libc is not compatible with glibc and thus all native code must be compiled with bionic libc.

1.1.3 The Android Runtime

The android runtime provides two essential things: the core libraries and the dalvik virtual machine. All of the applications run in the dalvik virtual machine, and the virtual machine makes it so the application can run in any hardware system, provided it has the dalvik virtual machine. The core libraries provide the application developers with all of the familiar java APIs used for things like network access, file access, standard data structures, etc.

1.1.4 The Application Framework

Sitting on top of the native libraries and the android runtime is the application framework. The application framework is all written in the Java language, contains all of the classes and core system services used for application development. This is also where the system UI of the Android system resides.

1.1.5 The Applications

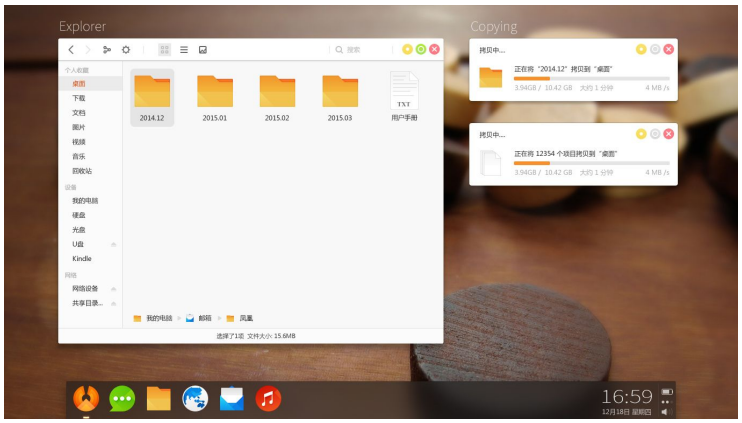
On top of the application framework lies all of the applications built by the developers for the android OS such as the browser, the file manager, the camera, etc. Applications exist on the android device as an android application package, or .apk file. These are built from their original source code and installed on the android device.

2. Design goal

The end goal of the project is to make a series of functional and aesthetic changes to the Android-x86 system in order to bring it to the level of usability exhibited by modern personal computers. This means we will have to architect major changes to core applications inside the system, as well as perhaps manufacture some stand-alone applications from scratch. The current proposed changes are as follows:

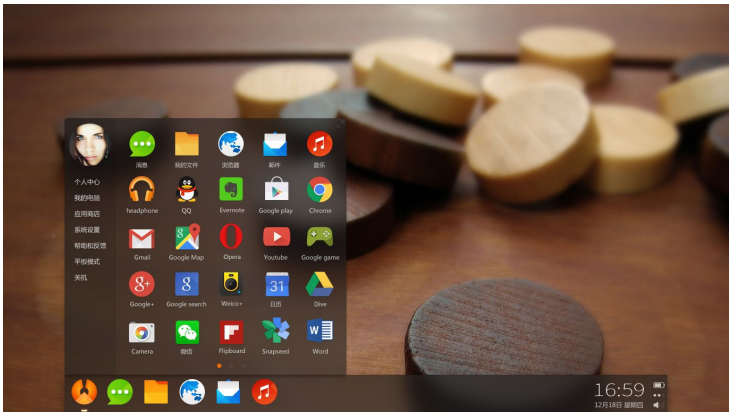
1. File Manager
2. Home Screen (Launcher)
3. Taskbar
4. Multi-Window Support

These applications have been determined as having a large impact on user experience, as well as lacking in its current form. Not only this, but these applications should be manageable in the scope of this project.



The file manager does not compare to that of other operating systems. The current version is simplistic but functional; it starts from a given directory and shows the contents in a list. In other operating systems the view defaults to clickable icons for each, and provides secondary navigation tools such as most visited places, back and forward, and displays the current path. There are also keyboard commands that make file managing more easy such as commands for copy and paste. Another ability that should be implemented is the dragging & dropping of files and directories.

The home screen needs significant changes in order to mimic that of normal personal computer operating systems. One of the biggest changes that can be seen in the two concept pictures above is the addition of a taskbar at the bottom. This is used to quickly change between applications that are currently running. This is featured on all common consumer OSs today, and should be added to Android-x86. Additionally a “Start” functionality that displays commonly used applications will be added.



Finally, we will implement functionality to support multiple independent windows. In the current android OS, whenever switching to a different app, the app will take up the entire screen space. We want to modify this behavior so that on application startup, the application will open into an independent window that can be moved and resized. Also, when you click on a window in the background, that application will be the application in focus.

3. Architectural choices and corresponding pros and cons

Of the many design architectures, there seems to be three architectures that many developers use to build android applications: the Model View Controller (MVC) model, the Model View Presenter (MVP) model, and the Model-View-View Model (MVVM) model.

In the MVC model, the user interacts with the controller and sees the view. Whenever the user inputs something to the controller, the controller updates the model, which holds some type of data, and then the model will update the view so the user can see the changes.

In the MVP model, the user interacts with the view. Whenever the user interacts with the view, events will be sent from the view to the presenter. From here, the presenter will update its model. When the model changes, the model will send a confirmation message to the presenter indicating that an update has been made. Once the presenter receives this notification, it will update the view to reflect changes.

In the MVVM model, the view is the user interface, the model is the data representation of the screen, and the view and model are directly bound together. This direct binding means that whenever the user changes information on the view, the view notifies the model to update its data. In this design pattern, the view model simply exposes the model to the view and its properties and commands.

For the most part, the MVP and MVVM models are the same. The only difference is that the MVVM model requires bindings whereas the MVP model does not. This difference corresponds to more code-behind in the presenter of the MVP model, and less code in the viewmodel.

All of the models described above have loose coupling between the view and the model, which makes unit testing easier. The only difference is that with the MVC model, where the controller unifies all of the different windows together, limits the areas unit testing can cover.

4. Selected architecture

Since android supports all of the above implementation schemes, we have decided to go with the MVVM model, simply because our team members have experience with Windows WPF MVVM model. If later, we find that the Android API available on the Android-x86 platform does not support bindings, we will revert to the MVP model.

5. Implementation notes

We will have four development environments for testing purposes: development, integration, acceptance and production, where production is the release pushed into the final version of the product. The development environment will be used for the developers to build and test the product. The integration environment will be used for the testers to run the product in a test environment. The acceptance environment will be to test use cases in the actual android environment.