ACTIVE PEN INPUT AND THE ANDROID INPUT FRAMEWORK

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Andrew Hughes

June 2011

COMMITTEE MEMBERSHIP

TITLE:	Active Pen Input and the Android Input Framework
AUTHOR:	Andrew Hughes
DATE SUBMITTED:	June 2011
COMMITTEE CHAIR:	Chris Lupo, Ph.D.
COMMITTEE MEMBER:	Hugh Smith, Ph.D.
COMMITTEE MEMBER:	John Seng, Ph.D.

Abstract

Active Pen Input and the Android Input Framework

Andrew Hughes

User input has taken many forms since the conception of computers. In the past ten years, Tablet PCs have provided a natural writing experience for users with the advent of active pen input. Unfortunately, pen based input has yet to be adopted as an input method by any modern mobile operating system. This thesis investigates the addition of active pen based input to the Android mobile operating system.

The Android input framework was evaluated and modified to allow for active pen input events. Since active pens allow for their position to be detected without making contact with the screen, an on-screen pointer was implemented to provide a digital representation of the pen's position. Extensions to the Android Software Development Kit (SDK) were made to expose the additional functionality provided by active pen input to developers. Pen capable hardware was used to test the Android framework and SDK additions and show that active pen input is a viable input method for Android powered devices.

Android was chosen because it is open source and therefore available to modify and test on physical hardware. Gingerbread (Android 2.3) was used as the code base for this thesis. All modifications to the Android framework that are detailed in this thesis will be made available online. The goal of this thesis is to explore methods of integrating and exposing active pen input in Android and encourage the implementation and adoption of active pen input by Google as a standard input method in Android.

Acknowledgements

I would like to thank Jeff Brown at Google for his communication, insights, and discussion of the Android framework and active pen input. His thoughts have been invaluable in exploring and adding to the Android framework.

I would like to thank James Carrington at N-trig for his discussions of active pen input as well as his help in acquiring an N-trig/Nvidia prototype tablet.

I would like to thank my thesis advisor, Dr. Chris Lupo, for his feedback and support on this thesis.

I would like to thank my fiance, Alicen Ramin, for her emotional support of my research and execution of this thesis.

I would like to thank my roommate, Kyle Husmann, for his amazing friendship, encouraging and challenging me over the past five years.

I would like to thank my parents for their love and support. I could not have gotten this far without you.

Most importantly, I would like to thank God for giving me the gifts and passions He has built in me and for guiding me in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of user input has always been to make interacting with devices as natural as possible. Input methods like the mouse have provided great intuitive functionality to users by providing them with a digital representation of something they are moving in real life and allowing them to interact with objects on the screen. A challenge for the mobile industry has been to reproduce this intuitive functionality in a transparent and seamless way. To do this, mobile devices must have input methods that are built into the design of the device. They must also implore methods that users are used to in their everyday lives.

Touchscreens have played a major role in mobile devices by allowing users to directly interact with digital objects using their fingers. With the introduction of multi-touch touchscreens, users have been able to interact with content in more dimensions than ever before. These input methods, however, are still limited to the use of our fingers, or objects, such as passive styli, that mimic fingers. As humans, we have developed other analog tools to augment our everyday lives. Most notably, pen and paper have been invaluable to us as productivity tools. In an attempt to use these tools as an input method for computers, Tablet PCs

were created. Active pen input, the technology behind Tablet PCs, is an input method that can be integrated into devices and uses a pen-like device, or stylus, that the user uses to interact with objects on the screen. The main benefit of active pen input is the fact that writing is a well established human activity. Microsoft saw the potential in these devices and developed an operating system with active pen input in mind called Windows XP Tablet PC Edition. Active pen input has continued to be a part of the available input methods in Microsoft operating systems, as well as many Linux distributions.

Active pen input, however, has not yet been adopted by any modern mobile operating system as a viable input method. The difference between mobile and non-mobile operating systems, is that mobile operating systems were designed from the ground up to be navigated and used on the go. They do not expect a user to be sitting down at a table or desk using a mouse. They are generally focused on one full screen task at a time and provide a "touch friendly" interface to the user. Devices that run modern mobile operating systems are designed to be instant on, lightweight, and generally get all day battery life. Adding active pen input as an input method for these devices will open up an opportunity for enhanced productivity, capability, and applications to take advantage of the new functionality.

This paper presents the first open implementation of active pen input in any mobile operating system. The Android mobile operating system was chosen because it is open source and because there is hackable hardware readily available. This thesis investigates additions to the Android framework that take advantage of and expose the special functionality provided by active pen input. The goal of this thesis is to present an implementation of active pen input in Android and encourage Google to include active pen input as a standard input method in

future releases of Android. In doing so, this research hopes to further the mobile computing industry.

This paper is broken down into the following chapters: Chapter 2 discusses previous work in the field of active pen input. Chapter 3 provides background information on Android and active pen input. Chapter 4 details an initial evaluation of active pen input in Android. Chapter 5 looks at the current state of the Android input framework in Gingerbread (Android 2.3). Chapter 6 explores the changes and additions made to the Android input framework to allow for active pen input. Chapter 7 looks at the additions to the Android Software Development Kit (SDK) that give developers access to the additional functionality provided by active pen input. Chapter 8 describes how the framework and SDK additions were validated using physical test hardware. Chapter 9 outlines various limitations in the research implementation. Chapter 10 discusses recent updates to the official Android SDK by Google. Chapter 11 speculates on how this work can be beneficial to Google and the mobile industry and what more can be done. Chapter 12 summarizes the findings in this paper and provides final thoughts.

Chapter 2

Previous Work

Pen based computing has been attempted many times in the past, however each time it has generated more traction than the last. In November 2000, Microsoft began marketing a new type of device they called the Tablet PC [10]. This device included both active pen input as well as a special release of Windows XP called Windows XP Tablet PC Edition. This Windows version included special input methods designed to take advantage of the pen and make it more useful. Not only could the active pen interact with applications, but the operating system could perform handwriting recognition as a text input method. Since the introduction of Tablet PCs, many studies and papers have been published about the importance and potential of pen input in areas such as education and medical fields.

Education is an especially important sector for pen computing. It not only provides new productivity applications for students such as digital note-taking in class, but also for teachers to use pen input to annotate lectures. In a study performed at Seton Hall University in 2003, students were asked "Overall which of the following best describes your feeling regarding the use of tablet computing

by your professor in this course in terms of teaching and learning effectiveness?" Student responses ranged from 48% "very positive" and 33.7% "positive" to 3.1% "negative" and 5.1% "don't know/no opinion" [4].

Another study was performed in a software engineering course at the University of Alaska Anchorage in 2004 [11]. In this classroom, the instructor used a Tablet PC to write notes on PowerPoint lectures which contained code, UML diagrams, and standard bulleted slides. The instructor used the pen to highlight items on the slides, write notes, and coordinate exercises. In addition, the lectures were recorded (both audio from the classroom and video of the annotated slides) and made available to the students to watch again later. When presented with the statement "I prefer the computer "whiteboard" to a traditional blackboard (i.e. chalk) for use in the classroom", 47.6% of students responded "strongly agree", 19% responded "somewhat agree", 9.5% responded "neither agree nor disagree", and no student disagreed (23.8% did not respond).

Medical professionals have benefited from Tablet PCs as well. In *The Case for the Tablet PC in Health Care*, Hewlett-Packard states "being forced to work with paper can result in manual data entry, lost patient information, and an increase in medical errors" [5]. The white paper goes on to state, "The Tablet PC, with its versatile form that supports pen navigation, and the capability to write directly on the screen and convert to printed text, offers todays mobile health care workers the ability to revolutionize the ways they work by making it easy to capture, access, and use information wherever the job takes them." Specially designed Tablet PCs have been made for medical professionals, such as Motion Computing's Motion C5v Tablet PC which it calls "the industry's first Mobile Clinical Assistant (MCA)" [12].

Apart from the Microsoft Tablet PC platform, there has been only one other

recent device that takes advantage of active pen input technology. In May 2011, HTC released the Flyer tablet and Scribe pen [8] [6]. The HTC Flyer/Scribe combination uses N-trig's DuoSense digitizer technology [7], which combines both active pen input and capacitive multitouch into a single digitizer [14]. Together, the devices allow for active pen input in Android 2.3 using custom additions to the Android framework. These additions have not been made public and it is still unclear whether HTC intends to allow access to the active pen's functionality through a third party Android SDK add-on. The importance of developing a standard active pen input implementation in Android is to establish a common framework and feature set for applications to build off of. This will ensure maximum compatibility for applications across devices running Android and providing active pen input capability. In addition, device manufacturers will be more likely to create devices with active pen input if it is already officially supported in Android. Without a standard implementation made available by Google, active pen input will not fully penetrate the mobile computing industry.

Chapter 3

Background

This chapter provides background information about Google's Android operating system and active pen input.

3.1 Android

Android is an open source Linux based operating system developed by Google for mobile devices. The Android platform is made up of five main components as seen in Figure 3.1: the Linux kernel, Android runtime, system libraries, application framework, and applications [18].

Android uses a Linux 2.6 based kernel and includes various additions to support features such as power management, inter-process communication, and framebuffers used by system libraries and the application framework. At this point in time, the Andriod kernel is a fork of the mainline Linux kernel due to disagreements between the Linux kernel maintainers and Google.

The Android runtime includes a Java virtual machine created by Google,

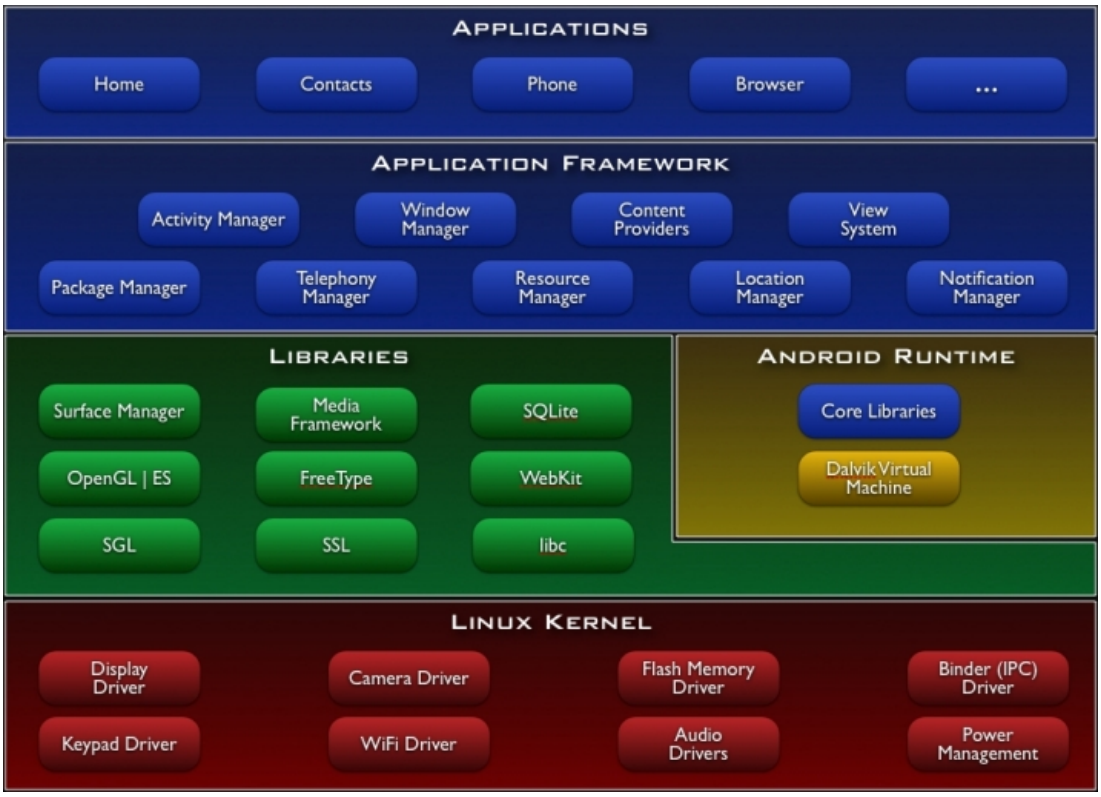


Figure 3.1: Android System Architecture.

called the Dalvik virtual machine, as well as a set of core libraries that provide most of the functionality available in the core libraries of the Java programming language. The Dalvik VM runs executables in the Dalvik Executable (.dex) format, which are classes compiled by a Java language compiler that have been transformed into the .dex format by the "dx" tool. The Dalvik VM is optimized for minimal memory footprint and good performance in resource constrained environments (such as mobile devices). The Dalvik VM is register-based and relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

The Android system libraries are a collection of C/C++ libraries used and exposed to developers by the Android application framework. This includes libraries such as the standard C system library, media libraries, SQLite database

library, WebKit engine, surface manager, SGL, and 3D libraries.

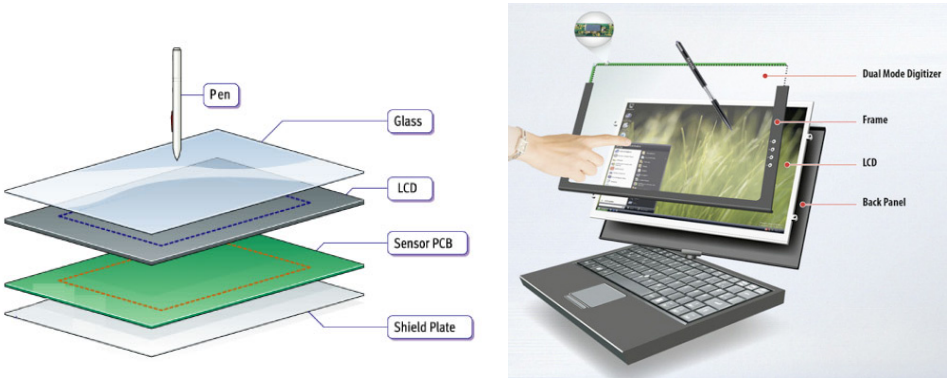
The Android application framework is a set of services and APIs that provide management of applications and components for building and running applications. The available developer APIs are exposed through the Android SDK [16].

Android applications are programs built by Google and third-party developers in Java (and C/C++ if using the Android Native Development Kit, or NDK [15]).

3.2 Active Pen Input

Active pen input provides many features not available with capacitive touchscreens. Some of these feature are dependent on the manufacturer of the active pen digitizer, but the basic functionality is: ability to detect the pen position (x,y) above the display, ability to detect position of contact with display (x,y), and report the pressure when in contact with the display. In both cases, the precision of the x,y coordinates are high, with sub-pixel accuracy. In addition, many pens contain side button(s) and/or a pressure sensitive "eraser" that can provide additional functionality to an application.

The primary manufacturers of active pen input technology are Wacom and N-trig. The former produces a digitizer panel that goes behind a display, while the latter makes a transparent digitizer that performs both pen and touch input and is positioned above a display. Wacom also has a touchscreen solution that goes above the display and integrates with a single pen and touch controller, but it is a physically separate component from their pen digitizer panel. A comparison of Wacom and N-trig's pen technologies can be seen in Figure 3.2 [17] [13].



(a) Wacom (digitizer below display) (b) N-trig (digitizer above display)

Figure 3.2: Wacom and N-trig Digitizer Technologies.

Chapter 4

Initial Evaluation

The use of active pen input in Android was initially evaluated using a Lenovo x61 Tablet PC and Android 1.6 (Donut). The x61 Tablet PC has a Wacom active pen digitizer integrated behind the display. The open source Android-x86 [1] project was used to get Android compiled and running for the x61 Tablet PC. In its early days, the Android-x86 project had limited support for mice with their own additions to the Android input framework. In function, an active pen is very similar to a mouse. They both support showing the position of a "pointer" on the screen and provide buttons (whether the tip or side of the pen) for interacting with objects on the screen. The main difference is that a mouse is a relative positioning device, while an active pen is an absolute positioning device.

The Android-x86 mouse patch [2] was used to determine the additions required to integrate an additional pointing device into Android. Using this as a guideline, a new "Stylus" input class was defined and assigned to the digitizer input device in order to track active pen events. When these events reached the WindowManager they were used to set the mouse cursor location.

Since the active pen digitizer in the x61 Tablet PC is a serial device, it will not communicate directly with the Android input system without the proper driver. Since the support for serial Wacom devices in Linux is limited to the X Window System (which Android does not use), an appropriate driver needed to be written. Since Android uses the input event system (input devices registered in /dev/input), a simple input event driver was written. This input event driver reads values from a sysfs node and creates input events appropriately. The utility "wacdump", included with the Linux Wacom Project source code [9] and originally designed to detect and report all active pen digitizer functions to an interactive terminal screen, was modified to write each action detected by the active pen digitizer to the sysfs node created by the input event driver. More details regarding the software stack used to test the hardware can be seen in Section 8.1.

Chapter 5

Android Input Framework

The Android input framework is made up of a series of C++ and Java classes that detect, filter, categorize, and inject input events into the currently running Activity or system component (such as the notification bar). The input framework has changed significantly over the last few releases of Android. This section will discuss the state of the input framework as it is in Gingerbread (Android 2.3). The Android source code can be acquired from the Android Open Source Project (AOSP) git repository [3]. All references to code and comments throughout the rest of this paper can be found in the AOSP git repository.

5.1 Overview

Input events are detected and read in by the EventHub. The InputReader continually acquires new events from the EventHub and performs initial filtering and categorization on input events based on the device an event is from. This essentially turns "raw" input events into "cooked" events (from InputReader.h). These "cooked" events are then added to the InputDispatcher queue. The Input-

Dispatcher continually publishes queued events to all valid input targets. There can be multiple valid input targets listening for input events. These input targets can range from the currently focused application or system component to system services that are monitoring input events. Each valid input target at the time of event publishing is notified through an InputQueue that it has received an input event. The InputQueue then dispatches the input event to the InputHandler that has been registered with it. If the target is an application, the InputHandler then dispatches the input event to the corresponding View. Application View and Activity components are notified of input events via their event dispatchers and input event handlers.

5.2 Startup

When Android first starts, the SystemServer is created. The SystemServer is designed to launch all the major framework services. The SystemServer and all the framework components it creates are written in Java. Each component that must communicate with native C/C++ code either uses JNI (Java Native Interface) or Android's Binder IPC mechanism. To start the input framework, the SystemServer starts the WindowManagerService which in turn creates the InputManager. The InputManager uses JNI to create a native class called the NativeInputManager and provides callbacks to communicate with the Java InputManager. The NativeInputManager is designed to be the connection between the Java InputManager and the rest of the native input framework. The NativeInputManager also implements the InputReaderPolicy and InputDispatcherPolicy (see Section 5.3.4). When instantiated, the NativeInputManager creates the EventHub along with the aptly named InputManager. The (native) InputManager

creates two threads, one for the InputReader and one for the InputDispatcher. As briefly described in Section 5.1, the InputReader thread reads and preprocesses raw input events, applies policy, and posts messages to a queue managed by the InputDispatcher, while the InputDispatcher thread waits for new events on the queue and asynchronously dispatches them to applications.

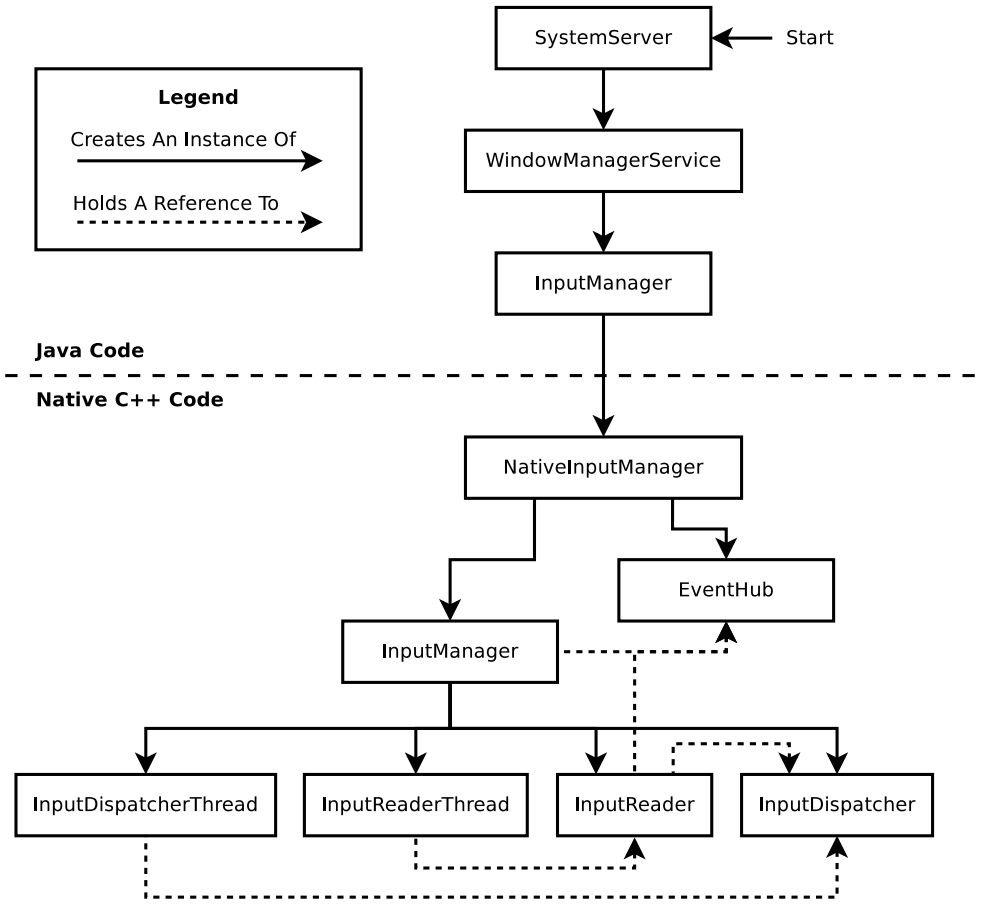


Figure 5.1: Input Framework Startup.

5.3 Event Processing Core

The native InputManager (and related classes) make up the core of the Android input framework. This section describes these components in more detail.

5.3.1 InputManager

The InputManager is a simple class that creates the InputReader, InputDispatcher, InputReaderThread, and InputDispatcherThread. The InputReaderThread simply calls the InputReader's loopOnce() method forever. Similarly, the InputDispatcherThread calls the InputDispatcher's dispatchOnce() method forever.

5.3.2 InputReader

The Android documentation describes the role of the InputReader as "processes raw input events and sends cooked event data to an input dispatcher" (from InputReader.h). The InputReader continuously retrieves a raw event from the EventHub (see Section 5.3.5) and processes it. It keeps track of a list of devices that are currently connected along with their capabilities. Each device (defined as an InputDevice) has an associated InputMapper that helps map each raw input event from an input device to a "cooked" event state that can be sent to the InputDispatcher (see Section 5.3.3).

When an InputDevice is created (after a new device is detected by the EventHub), it is assigned an InputMapper based on the input device class reported by the EventHub. When that input device produces input events, the events are given to the InputMapper for processing before they are dispatched. A list of

available InputMappers can be seen in Table 5.1. After processing the raw input event, each InputMapper notifies the InputDispatcher of its input event.

SwitchInputMapper	Maps switches such as the "lid switch"
KeyboardInputMapper	Maps physical keyboards and buttons to key events
TrackballInputMapper	Maps trackballs to motion events
SingleTouchInputMapper	Maps single pointer touchscreens to motion events (based on TouchInputMapper)
MultiTouchInputMapper	Maps multi pointer touchscreens to motion events (based on TouchInputMapper)

Table 5.1: Available InputMapper Classes.

In addition to processing input events, each InputMapper is responsible for reporting the source class and source of an input event. Each of these constants are defined in both the native code and in the Java code (in the InputDevice class). The Java source class and source constants can be seen in Tables 5.2 and 5.3. The only difference between the native and Java constants is the native constants are prefixed with "AINPUT_". The native constants are used by the InputReader and InputDispatcher.

SOURCE_CLASS_BUTTON	The input source has buttons or keys.
SOURCE_CLASS_POINTER	The input source is a pointing device associated with a display.
SOURCE_CLASS_POSITION	The input source is an absolute positioning device not associated with a display (unlike SOURCE_CLASS_POINTER).
SOURCE_CLASS_TRACKBALL	The input source is a trackball navigation device.

Table 5.2: Available InputDevice Source Classes.

5.3.3 InputDispatcher

The InputDispatcher can be broken down into two main parts: one part continually reads from the internal queue of "EventEntry" objects (run by the

SOURCE_UNKNOWN	The input source is unknown.
SOURCE_KEYBOARD	The input source is a keyboard.
SOURCE_DPAD	The input source is a DPad.
SOURCE_TOUCHSCREEN	The input source is a touch screen pointing device.
SOURCE_MOUSE	The input source is a mouse pointing device.
SOURCE_TRACKBALL	The input source is a trackball.
SOURCE_TOUCHPAD	The input source is a touch pad or digitizer tablet that is not associated with a display (unlike SOURCE_TOUCHSCREEN).
SOURCE_ANY	A special input source constant that is used when filtering input devices to match devices that provide any type of input source.

Table 5.3: Available InputDevice Sources.

InputDispatcherThread described in Section 5.3.1), the other part adds entries to the internal queue from the notify calls made by the InputReader. Each time an event is removed from the internal queue, the current valid input targets are determined. This includes the currently focused application, along with other viewable windows or system components, and any system services monitoring input events. If the event is a motion event, it is determined whether or not the event is within the bounds of the focused window area, obscured area outside the focused application, or in the case of multiple pointers (or touches) whether or not the motion event can be "split" across multiple windows.

Once the InputDispatcher determines how to handle the event, it dispatches the event to each of the valid input targets. To do this, a "dispatch cycle" is prepared for each input target. This sets up a Connection object with the input target and enqueues the event on the connection's outbound queue. When the dispatch cycle is started, the event is published to the connection's InputPublisher object. After the event has been published, a "dispatch signal" is sent

using the InputPublisher. This notifies the InputConsumer on the other side of the connection that there is an event ready to be consumed. Once the event is consumed, the InputConsumer sends a "finished signal" which is received by the InputPublisher. This tells the InputDispatcher that this event was successfully consumed. The connection's outbound queue is checked for any additional events and the dispatch cycle is repeated. When there are no more events on the connection's outbound queue, the connection is closed. This allows for events to be continually added to a connection's outbound queue as long as it is still sending them without the Connection needing to be re-established (this is referred to as event streaming).

5.3.4 InputReaderPolicy and InputDispatcherPolicy

The InputReaderPolicy and InputDispatcherPolicy are used by the InputReader and InputDispatcher respectively to call into Java code via JNI and communicate with the WindowManagerService and other system components. Both the InputReaderPolicy and InputDispatcherPolicy are implemented by the NativeInputManager. These policy callbacks perform various tasks from checking input injection permission to notifying the WindowManagerService of an ANR (Application Not Responding) to determining filtering parameters. Some of these callbacks access system properties or build configurations that represent the input configuration of a specific Android device. This allows the behaviour of some elements of the input framework to be modified simply by changing these system properties and rebooting.

5.3.5 EventHub

Android detects input events from devices that have an input event driver which registers a device under /dev/input. When a new input event is created in /dev/input, the EventHub determines if it is a device that is supported by Android by checking its capabilities. If the device is supported, one or more input device classes are assigned to the input device. A list of available input device classes can be seen in Table 5.4 (the first cell in the table is a prefix for each of the input device classes). The EventHub generates synthetic add/remove events whenever a device has been connected or disconnected. A stream of events is detected and returned via the EventHub::getEvent() function. This is called by the InputReader (see Section 5.3.2) and is guaranteed to be called by a single caller (so there is no need for locking). All input events are added to an input buffer which is read by getEvent(). This allows for events to be retrieved from the devices that generated them in the order they occurred. When there are no more events in the input buffer, getEvent() calls poll on the input device file descriptors and waits for more input.

INPUT_DEVICE_CLASS_	
KEYBOARD	The input device is a keyboard.
ALPHAKEY	The input device is an alpha-numeric keyboard (not just a dial pad).
TOUCHSCREEN	The input device is a touchscreen (either single-touch or multi-touch).
TRACKBALL	The input device is a trackball.
TOUCHSCREEN_MT	The input device is a multi-touch touchscreen.
DPAD	The input device is a directional pad (implies keyboard, has DPAD keys).
GAMEPAD	The input device is a gamepad (implies keyboard, has BUTTON keys).
SWITCH	The input device has switches.

Table 5.4: Available Input Device Classes.

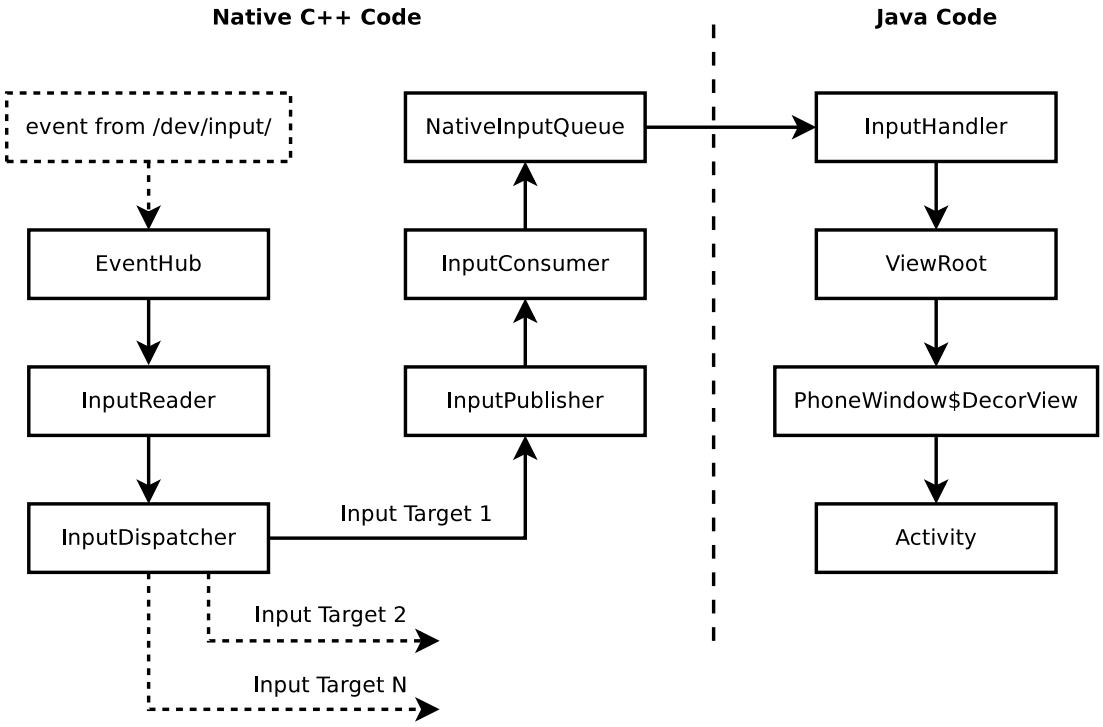


Figure 5.2: Event Propagation To an Activity.

Chapter 6

Input Framework Changes

In order for active pen input to be detected and propagate all the way to an application, various additions and changes were made to the Android input framework. This chapter describes those changes.

6.1 New Input Device Class

In order to detect a new type of device for interpretation, a new input device class had to be made. An addition was made to the existing input device classes shown in Table 5.4 called the INPUT_DEVICE_CLASS_ACTIVE_DIGITIZER. This new input device class defines the input device as an active pen digitizer. An input device is assigned this class in the EventHub if it is an absolute positioning device and it tests positive for BTN_TOOL_PEN. This shows that the input device has at least a pen tool type.

6.2 ActiveStylusInputMapper

A new input mapper was added to the InputReader for active pen input, called the ActiveStylusInputMapper. The TouchInputMapper was used as a basis for the ActiveStylusInputMapper because of the underlying similarities in the devices. Both device types support down, move, and up events, however an active pen supports additional actions such as "hover" and "proximity." A SingleActiveStylusInputMapper (extending ActiveStylusInputMapper) was created to further refine the ActiveStylusInputMapper for a digitizer that only supports a single active pen. Since most currently available active pen digitizers are single input (unless used by more than one person at a time, there's no need for more than one active pen), the focus is on the SingleActiveStylusInputMapper, however room is still left for a MultiActiveStylusInputMapper.

The SingleActiveStylusInputMapper is assigned to input devices with the new INPUT_DEVICE_CLASS_ACTIVE_DIGITIZER. When active pen input events are generated, the process() method gets called by the InputReaderThread. This method was modified to support the various input event values reported by the active pen input event driver. These values (defined in input.h) are listed in Table 6.1. Not all the values are valid for every active pen digitizer and additional values may be needed to completely represent all possible functions of an active pen digitizer. For example, not all active pens have a pen tool on one end (BTN_TOOL_PEN) and an eraser (or "rubber") tool on the other end (BTN_TOOL_RUBBER). Also, not all active pens have two (or any) side buttons (BTN_STYLUS and BTN_STYLUS2).

Once event values have "accumulated" and a SYN_REPORT is received, the sync() method is called and the event state is determined. Whether the pen is

BTN_TOUCH	The stylus is in the "touched" or "down" position.
BTN_TOOL_PEN	The pen tool is being used.
BTN_TOOL_RUBBER	The "rubber" or eraser tool is being used.
BTN_STYLUS	The primary stylus side button is pressed.
BTN_STYLUS2	The secondary stylus side button is pressed.
ABS_X	The absolute x position of the stylus.
ABS_Y	The absolute y position of the stylus.
ABS_PRESSURE	The absolute pressure of the stylus tip.
ABS_TOOL_WIDTH	The absolute tool width of the stylus.
SYN_REPORT	Synchronization command stating that all values have been reported for a single event.

Table 6.1: Active Stylus input.h Events.

in the proximity of the digitizer is determined by the value of BTN_TOOL_PEN and BTN_TOOL_RUBBER. If neither value is reported, then the pen has just left the proximity of the digitizer. If either value is reported, then the pen is in proximity. To determine if the pen has just entered proximity, the current and previous state of the stylus must be compared.

6.2.1 Actions

Once the current state of the stylus is saved, the state is analyzed and compared to the previous state to determine which type(s) of MotionEvents to send to the InputDispatcher. Actions that have been added to MotionEvent can be seen in Table 7.1 and are described in Section 7.1.1. If the pointer count has changed, then the active pen has either entered or exited the proximity of the digitizer. If the previous state had a pointer, then the active pen has exited proximity and an AMOTION_EVENT_ACTION_EXIT_PROXIMITY (the native version of MotionEvent.ACTION_EXIT_PROXIMITY) event is sent to the

InputDispatcher. If the pointer count has changed and the current state has a pointer, then an enter proximity event is sent to the InputDispatcher. The rest of the MotionEvent actions are dependent on if the active pen is in proximity of the digitizer (up, down, move, and hover events can only happen while the active pen is in range). These events are sent to the InputDispatcher in between proximity enter and exit events. An AMOTION_EVENT_ACTION_HOVER event is sent if the stylus is not currently "down", otherwise an AMOTION_EVENT_ACTION_MOVE event is sent. Finally, up and down events are sent based on the current and previous state of the pointer.

AMOTION_EVENT_HOVER	The input device is hovering above but not touching the surface.
AMOTION_EVENT_ENTER_PROXIMITY	The input device has entered detection range.
AMOTION_EVENT_EXIT_PROXIMITY	The input device has exited detection range.

Table 6.2: New Native Motion Event Actions.

6.2.2 Side Buttons

In all cases, the state of the active stylus side button(s) are saved in the meta state shared across all input devices. The meta state is used for keys that modify the behavior of something. In this case, the side buttons on an active pen generally modify the behavior of the pen in some way. The actual behavioral modification is up to the application using the pen. The fact the the button was pressed or depressed is not as critical as knowing if the button is *being* pressed when a MotionEvent occurs. In order to keep track of the side buttons, two new meta states and keycodes were added to KeyEvent: META_BTN_STYLUS_ON,

META_BTN_STYLUS2_ON, KEYCODE_BUTTON_STYLUS, and KEYCODE_BUTTON_STYLUS2. Each corresponding meta state and keycode represent the pressed state of the stylus side button. KEYCODE_BUTTON_STYLUS represents the primary side button while KEYCODE_BUTTON_STYLUS2 represents the secondary side button. The native code representations of these meta states and keycodes can be seen in Tables 6.3 and 6.4.

AMETA_BTN_STYLUS_ON	A meta state mask used to check if the primary stylus side button is being pressed.
AMETA_BTN_STYLUS2_ON	A meta state mask used to check if the secondary stylus side button is being pressed.

Table 6.3: New Native Meta States.

AKEYCODE_BUTTON_STYLUS	A keycode for the primary stylus side button.
AKEYCODE_BUTTON_STYLUS2	A keycode for the secondary stylus side button.

Table 6.4: New Native Keycodes.

6.2.3 Tool Type

The last piece of information an active pen digitizer might provide is a tool type. Some digitizers, such as those made by Wacom, provide both a pen tool and an eraser (or "rubber") tool on a single active stylus. New "TOOL_TYPE" constants were added to the MotionEvent class to provide this information (see Section 7.1.3). Since these tool type constants add an entirely new field to MotionEvent, a lot of internal communication mechanisms and objects needed to be modified to support tool types. An in-depth discussion of these changes can be seen in Section 6.3.

By default a TOOL_TYPE_PEN is assigned to MotionEvents generated by the ActiveStylusInputMapper. This is useful for active pens that do not report a tool type. If an active pen does report a tool type, the appropriate tool type is assigned to an event when reporting to the InputDispatcher. In the InputReader, these tool type constants are referred to as AMOTION_EVENT_TOOL_TYPE_PEN and AMOTION_EVENT_TOOL_TYPE_RUBBER as seen in Table 6.5. In the event that hardware becomes available that supports new tool types, support for these tool types could be easily added.

AMOTION_EVENT_TOOL_TYPE_NONE	The input device does not support tool types.
AMOTION_EVENT_TOOL_TYPE_PEN	The motion event is being performed by the pen tool.
AMOTION_EVENT_TOOL_TYPE_RUBBER	The motion event is being performed by the "rubber" or eraser tool.

Table 6.5: New Native Motion Event Tool Types.

6.3 Adding Tool Type Support

Unlike the new actions, meta states, and keycodes discussed in Section 6.2, which simply expand on previously defined fields and communication methods, "tool type" is a completely new MotionEvent field. Support for this new field, including storage, setters and getters, and constructor parameters needed to be added throughout the input framework. In native code, a tool type member was added to the MotionEntry object, which is used for motion event communication within the InputDispatcher. A tool type parameter was added to the obtainMotionEntry() method to support creating the new MotionEntry object.

The notifyMotion() method used by vaious InputMappers in the InputReader to inform the InputDispatcher of a new motion event was modified to support a tool type parameter. The InputPublisher and InputConsumer classes, internal to the InputTransport class, were modified to support tool type along with the InputMessage object used to pass messages between the publisher and consumer using shared memory. Tool type was added to both the native and Java versions of MotionEvent along with corresponding updates to the JNI "glue" in android_view_MotionEvent.cpp. Finally, all instances of the modified method calls or object constructor calls needed to be changed to support the new parameter lists, including all input framework test classes, along with the transfer of tool type data within appropriate functions. A list of classes whose fields, constructors, and accessors were modified can be seen in Table 6.6, while a list of methods whose parameters were modified can be seen in Table 6.7.

Class	File	Description
MotionEvent	MotionEvent.java	Java MotionEvent received by an application.
MotionEvent	Input.cpp/h	Natvie MotionEvent used throughout input framework.
InputPublisher	InputTransport.cpp/h	Publishes events to be consumed by an InputConsumer.
InputConsumer	InputTransport.cpp/h	Consumes events published by an InputPublisher.
InputMessage	InputTransport.cpp/h	Private intermediate representation of input events as messages written to an anonymous shared memory buffer.

Table 6.6: Classes Updated to Support Tool Type Field.

Method	File	Description
obtainMotionEntry	InputDispatcher.cpp/h	Allocates a new MotionEntery event for use within the InputDispatcher.
notifyMotion	InputDispatcher.cpp/h	Called by InputMappers in the InputReader to notify the InputDispatcher of new motion events.
initialize	Input.cpp/h	Native MotionEvent initializer method.
publishMotionEvent	InputTransport.cpp/h	Publishes a MotionEvent to the InputChannel for an InputConsumer to consume.
populateMotionEvent	InputTransport.cpp/h	Populates a MotionEvent object with values from the InputMessage shared memory object created by the InputPublisher.

Table 6.7: Methods Updated to Support Tool Type Parameter.

6.4 InputDispatcher Changes

A few changes were made to the InputDispatcher logic to allow the new MotionEvent actions to be dispatched appropriately. To dispatch a motion event, the input targets must first be determined. This includes either the currently focused or touched window, along with any monitoring targets (such as the PointerManagerService described in Section 6.6). Depending on certain event criteria, either findTouchedWindowTargetsLocked() or findFocusedWindowTargetsLocked() is called. Originally, if the motion event was a "pointer event" (has the source class AINPUT_SOURCE_CLASS_POINTER) the touched window targets were found, otherwise the focused window targets were found. This worked fine when the only two types of motion events were either touchscreen or trackball events. Touchscreen events would always determine touched window targets while trackball events would look for focused window targets.

When determining touched window targets, the InputDispatcher simply ig-

nores an event if it receives something other than a down event and the pointer is not already down. This prevents spurious move or up events from occuring when a pointer is not already in the down state, keeping an application from receiving any unexpected event sequences. However, this means the new hover and proximity events would be dropped since these events occur while a pointer is up. To solve this, the criteria was changed for which touched or focused windows targets were found. Instead of always finding touched window targets when a motion event is a "pointer event", it was determined whether a motion event is a "non-touch event". When a motion event is a non-touch event, focused window targets are found, otherwise touched window targets are found. A motion event is a non-touch event if either it is *not* a pointer event or it *is any* of the three new motion event actions (hover, enter proximity, or exit proximity). Since these three actions should never occur while a window target is being touched, this check can be done without also requiring the event source to be AINPUT_SOURCE_ACTIVE_STYLUS, allowing other new input sources to use these new actions as well (and have the events dispatched to focused window targets).

6.5 GetMaxEventsPerSecond

An input framework property that was experimented with was getMaxEventsPerSecond(). This function checks for a value stored in the system property "windowsmgr.max_events_per_sec" and returns a value of 60 if no value was set. The sample rate of current active pen digitizers ranges from 100Hz to 200Hz, depending on the digitizer hardware. Setting this system property to 200 allows the input framework (specifically the InputDispatcher) to throttle input events more appropriately. Due to the fact that it is only a system property (which is read

directly from a system property text file), this value can be easily changed by device manufacturers depending on the hardware that is used in a specific device.

6.6 PointerManagerService

A new system service was added to manage on screen pointers. This service is started by the SystemServer and manages the position, appearance, and visibility of pointers in Android. At the moment this service only supports a single pointer, but it could be easily expanded to support multiple pointers if an appropriate use case is determined. The term "pointer" is used, rather than "cursor," to cover a wider range of devices that could be represented. A cursor has generally been associated with the use of a mouse, while a pointer could represent the on screen position of an active stylus, mouse, or other "pointing" device.

To inform the user which pointing device is being represented by a pointer, a different icon could be used. For example, traditionally an arrow is used to represent a mouse, while a circular dot is used to represent an active stylus. The service is designed to use an image from the Android framework to represent each type of pointing device supported by Android. A setting could even be added to let a user choose from a list of available icons which they would prefer as the default icon for each type of pointing device. Additionally, an application developer can temporarily change the look of the pointer through the SDK by providing an image resource through the PointerManager (see Section 7.2).

The PointerManagerService functions by registering with the WindowManager for input events. An InputHandler callback is provided in which MotionEvents are checked to see if the device that generated the events are of SOURCE_CLASS_POINTER. If they are, then they are dispatched for further parsing, oth-

erwise they are ignored. Although there is a SOURCE_MOUSE in Android 2.3, there is no way for this input source to be assigned to an input event. Any input device that is both a relative controller and has a mouse button is classified as a SOURCE_TRACKBALL. Since trackballs are not pointing devices that should be represented by an on screen pointer, the active stylus is the only pointing device that needs to be handled.

The visibility of a pointer is tied to ACTION_ENTER_PROXIMITY and ACTION_EXIT_PROXIMITY. A pointer is only visible as long as the active pen is in range (disappearing when the user pulls the pen away from the device). This could easily be adjusted to allow the pointer to fade out after a few seconds rather than disappear immediately. Additionally, the visibility of a pointer can be controlled by each application.

The PointerManagerService uses a Surface object to draw the pointer on the screen. Each time a valid MotionEvent is received and the pointer should be displayed, an openTransaction() is performed on the Surface and the position and layer are set. The position of the Surface is set according to the x,y position of the MotionEvent, while the layer is set to be one layer higher than the top animation layer on the screen (see Section 6.7). Afterward, Surface.closeTransation() is called to complete the changes to the Surface.

Whenever a pointer's visibility or look is changed by an application, the pointer Surface must be modified to reflect those changes. Changing the visibility simply sets the Surface's show() or hide() method appropriately. This must again be surrounded by a Surface open/closeTransaction() block. To change the image used by a pointer, a Drawable object is retrieved from the provided resource id associated with the application requesting the pointer change. The size of the Drawable is determined and the Surface size is set accordingly. The set-

Size() Surface call must also be surrounded by a Surface open/closeTransaction() block. Next, a Canvas object is locked and retrieved from the Surface, filled with transparency, and the Drawable is drawn on the Canvas. Finally the Canvas is unlocked and posted to the pointer Surface.

6.6.1 Pointer Modification Using the Side Button

To illustrate how the side button(s) could be used to modify the pointer, the pointer is changed to a different image while the side button is being pressed. This is only done while there is no application modifying the look or visibility of the pointer to keep from unintentionally changing the pointer when it should not be changed. If a change in the primary side button is detected, the pointer is set to the look defined for the new state. For example, a ring could appear around the pointer while the primary side button is being pressed. This would be useful in order to notify the user that they have modified the behavior of the stylus while holding down the side button. Applications could change the pointer image to something that represents the new pointer behavior, and change it back once the button has been released.

6.7 WindowManager Additions

An intra-framework call was added to the WindowManager: getTopAnim-Layer(). This call simply returns the integer value representing the top animation layer on the screen. This is used by the PointerManagerService to correctly draw the pointer directly above the current highest layer.

Android.mk
core/java/android/app/ContextImpl.java
core/java/android/content/Context.java
core/java/android/view/IWindowManager.aidl
core/java/android/view/InputDevice.java
core/java/android/view/KeyEvent.java
core/java/android/view/MotionEvent.java
core/jni/android_view_MotionEvent.cpp
core/res/res/values/attrs.xml
include/ui/EventHub.h
include/ui/Input.h
include/ui/InputDispatcher.h
include/ui/InputReader.h
include/ui/InputTransport.h
include/ui/KeycodeLabels.h
libs/ui/EventHub.cpp
libs/ui/Input.cpp
libs/ui/InputDispatcher.cpp
libs/ui/InputReader.cpp
libs/ui/InputTransport.cpp
libs/ui/tests/InputDispatcher_test.cpp
libs/ui/tests/InputPublisherAndConsumer_test.cpp
libs/ui/tests/InputReader_test.cpp
native/android/input.cpp
native/include/android/input.h
native/include/android/keycodes.h
services/java/com/android/server/InputManager.java
services/java/com/android/server/SystemServer.java
services/java/com/android/server/WindowManagerService.java

Table 6.8: List of Modified Files in frameworks/base/.

core/java/android/app/IPointerManager.aidl
core/java/android/app/Pointer.aidl
core/java/android/app/Pointer.java
core/java/android/app/PointerManager.java
core/res/res/drawable/pointer.png
core/res/res/drawable/pointer_side_button_pressed.png
services/java/com/android/server/PointerManagerService.java

Table 6.9: List of New Files in frameworks/base/.

Chapter 7

Developer API Additions

A few additions were made to the Android SDK that allow application developers to access the additional functionality provided by active pen digitizers.

7.1 MotionEvent

7.1.1 Actions

Three new actions were added to MotionEvent to support functionality provided by active pen input: ACTION_HOVER, ACTION_ENTER_PROXIMITY, and ACTION_EXIT_PROXIMITY (see Table 7.1). An ACTION_HOVER event is produced whenever an active pen is hovering above but not touching the surface of the display. It is similar to an ACTION_MOVE event in that it is a constant stream of pointer coordinates as the input device moves around. As soon as the active pen touches the display, the standard ACTION_DOWN and ACTION_MOVE events are produced until the pen leaves the surface, which creates an ACTION_UP event. Immediately following the ACTION_UP are more

ACTION_HOVER events until the pen leaves the detection range of the display or touches down again. When the pen leaves the detection range, an ACTION_EXIT_PROXIMITY event is produced to inform the application that the pen is no longer in range. Similarly, when the pen enters the detection range, an ACTION_ENTER_PROXIMITY event is immediately produced before any ACTION_HOVER or subsequent MotionEvents.

ACTION_HOVER	The input device is hovering above but not touching the surface.
ACTION_ENTER_PROXIMITY	The input device has entered detection range.
ACTION_EXIT_PROXIMITY	The input device has exited detection range.

Table 7.1: Additional MotionEvent Actions.

7.1.2 Meta State

Two new meta states were created: META_BTN_STYLUS_ON and META_BTN_STYLUS2_ON. These meta states represent the pressed state of the "stylus" (primary) and "stylus2" (secondary) buttons on the side of an active pen. Since meta states are shared across all input devices, MotionEvent.getMetaState() simply calls KeyEvent.getMetaState(). The new meta states, along with KEYCODE_BUTTON_STYLUS and KEYCODE_BUTTON_STYLUS2 (representing the two stylus buttons), were added to KeyEvent. On any MotionEvent, the getMetaState() method can be called and tested against the meta state masks in order to determine if either side button is being pressed.

7.1.3 Tool Type

In order to represent the active pen tool currently being used, a new set of MotionEvent constants were created. This new category was called "TOOL_TYPE" and consists of TOOL_TYPE_NONE, TOOL_TYPE_PEN, and TOOL_TYPE_RUBBER as seen in Table 7.2. TOOL_TYPE_NONE is assigned by default to all MotionEvents that are not generated by an active pen. Among active pens, TOOL_TYPE_PEN is assigned by default in the case that the active pen does not support multiple tool types. In the event that the active stylus *does* support multiple tool types, the tool type of each MotionEvent can be determined through the getToolType() method. This method will return one of the TOOL_TYPE constants. The tool types available in MotionEvent could easily be expanded if hardware becomes available that supports additional tool types. The use of tool types could also be expanded to other input device types that use MotionEvents (such as touchscreens, trackballs, and other future input methods).

TOOL_TYPE_NONE	The input device does support tool types.
TOOL_TYPE_PEN	The MotionEvent is being performed by the pen tool.
TOOL_TYPE_RUBBER	The MotionEvent is being performed by the "rubber" or eraser tool.

Table 7.2: MotionEvent Tool Types.

getToolType()	Returns the tool being used for this event.
setToolType(int toolType)	Sets the event's tool type.

Table 7.3: New MotionEvent Methods.

7.2 PointerManager

The PointerManager is an object that allows an application to communicate with the PointerManagerService (see Section 6.6). The PointerManager allows for a pointer's image and visibility to be changed. A PointerManager object is obtained through the Context.getSystemService() method by passing in the POINTER_SERVICE string constant. The methods provided by the PointerManager are outlined in Table 7.4.

setPointer(Pointer pointer)	Change the current image used for the pointer.
resetPointer()	Reset the pointer to its default state.
setVisible(boolean visible)	Set whether the current pointer is visible or not.

Table 7.4: Pointer Manager Methods.

7.2.1 Pointer

A Pointer object was defined to represent a single pointer and to be an object of communication between an application (though the PointerManager) and the PointerManagerService. For this reason, a Pointer implements the Parcelable class, which is required for an object to be used by Android's Binder IPC. The constructor for a Pointer takes three parameters: an image resource id, a horizontal offset, and a vertical offset. The image resource id should point to an image in the applications resources that will be used for the pointer. The horizontal and vertical offset values are used to describe where in the image the "point" is. These offset values are used by the PointerManagerService to determine where to display the pointer image relative to the x,y coordinates reported by a MotionEvent.

Chapter 8

Validation

In order to determine whether the changes made to the Android input framework and SDK to support active pen input were valid, a build of Android was created with the changes and was run on test hardware.

8.1 Test Hardware

The input framework changes and SDK additions were validated using a Wacom active pen digitizer connected to a Viewsonic Gtablet running a customized build of Android. Just like the active pen digitizer in the x61 Tablet PC mentioned in Chapter 4, the Wacom digitizer used for testing was a serial device. The serial Wacom digitizer was connected to the Gtablet via an FT232RL serial to USB chip, which creates a serial node at /dev/ttyUSB0. The same modified "wacdump" utility (renamed "wacandroid") and input event driver were used as with the x61 Tablet PC. Wacandroid reads and parses data written to /dev/ttyUSB0 by the Wacom digitizer and writes values to a sysfs node at "/sys/devices/platform/wac_android/input". The "wac_android" input event

driver registers a new input device in /dev/input, reads values from the sysfs node, and creates corresponding input events for the Android input framework to receive. An outline of the test hardware and software stack can be seen in Figures 8.1 and 8.2.

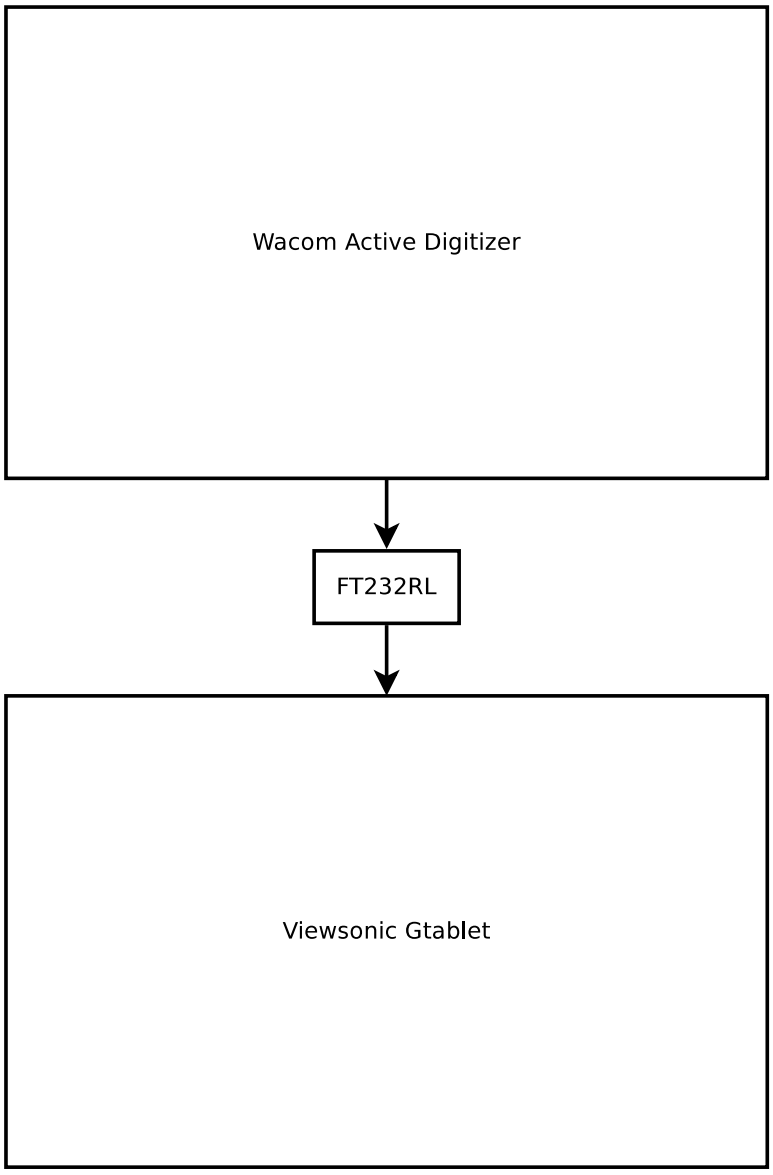


Figure 8.1: Test Hardware Setup.

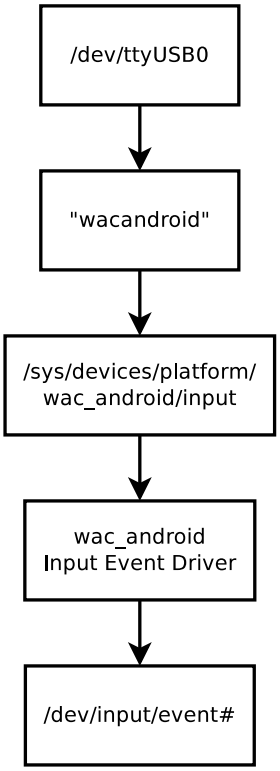


Figure 8.2: Test Software Setup.

8.1.1 Hardware Setup

The Wacom digitizer was not simply "plug and play." The pin-outs of the serial digitizer had to be determined. No official documentation was available outlining the connectivity of *any* Wacom digitizers, and requests for such information were denied. The pin-outs of the SU-025-C02 Wacom digitizer that were determined in testing can be seen in Table 8.1. Though the connector used by the Wacom digitizer contained fourteen pins, only seven of those pins were used by the serial communication (along with VCC), and the rest were not connected (represented by "NC" in Table 8.1). Once these pins were connected to the FT232RL serial to USB chip, a USB cable was connected to the Viewsonic Gtablet. This generated the /dev/ttyUSB0 serial device node and the Wacom digitizer could be communicated with.

Pin	Connection
1	GND
2-6	NC
7	DTR
8	RTS
9	RxD
10	TxD
11-12	NC
13	VCC 3.3V
14	GND

Table 8.1: Wacom SU-025-C02 Pin-outs.

Many hours of research were spent trying to determine the appropriate digitizer to use for testing, and many digitizers were acquired before determining the pin-outs for the digitizer used in testing. Attempts to acquire a development device with documentation were denied by Wacom, however N-trig and Nvidia provided a complete development tablet. Unfortunately, the framework changes were not able to be tested on the N-trig/Nvidia device because the necessary build files and proprietary binaries were not provided. In terms of communication and feedback, however, N-trig was willing to discuss ideas and was open to outside help and research in pushing for integration of active pen input in Android.

8.1.2 Features

The active pen stylus used with the Wacom digitizer provided the following features: pressure sensitive pen tip (pen tool), pressure sensitive eraser nub (eraser or "rubber" tool), and a single side button. The 12" digitizer provided a resolution of 24,780 by 18,630 (while the 10" screen resolution of the Gtablet was only 1024 by 600). This effectively allows the Wacom digitizer to provide approximately 587 samples per screen pixel.

8.2 Testing

To verify the Android framework changes, a build of Android was compiled and flashed to the test hardware. The use of active pen input was observed and behavior was tested throughout the system. An SDK build was created to test the new features provided in the SDK. A simple application was built to test all the additions to the SDK and verify their functionality.

Chapter 9

Limitations

9.1 Changing the Pointer

A current limitation of the PointerManagerService is that once an application changes the pointer through the PointerManager, the pointer will not change again until another application (or the same application) changes or resets the pointer. This could be easily remedied by having the PointerManagerService check which application is currently on the top of the Activity stack and compare it to the package that set the current pointer look or visibility. If the application package is not the same, then the pointer should be reset to the system default. This, unfortunately, is not an acceptable solution because this would require the application stack to be checked every time a new MotionEvent is received. Since an active pen digitizer generally produces input events at over 100Hz, a lot of unnecessary calls would be made and would generate unacceptable overhead. Instead, the PointerManagerService should be able to register with the Android framework (most likely the ActivityManagerService or WindowManagerService) to simply be notified every time a new application package is in focus. A study

of the ActivityManagerService and WindowManagerService could be performed to determine if this callback functionality is, or could be made, possible.

9.2 MotionEvent Actions

There is an inherent limitation in how the new MotionEvent hover, enter proximity, and exit proximity actions are dispatched to applications. Currently, dispatchTouchEvent() is used for all touchscreen and active stylus events, despite the fact that not all active stylus events "touch" the display. This can create issues in applications that rely on receiving an ACTION_DOWN event after an ACTION_UP event without any other events in between. A solution to this is now in place in the latest update to the official Android SDK where all non-touch events are dispatched to applications using the new dispatchGenericMotionEvent() method (see Chapter 10).

9.3 Mulitple Pointer Devices

An issue with having multiple pointer devices (such as a touchscreen and an active pen digitizer) is that the first "touch" on each device produces a MotionEvent with the same pointer index value (0). This can create problems for applications that do not make sure all events received in onTouchEvent() or dispatchTouchEvent() are from the same input source. In fact, even the Android framework does not usually perform this check. In the event that an application does not separate MotionEvents by source, and a finger and stylus are both put down and moved, the application will perceive that a single pointer is jumping back and forth between the finger and the stylus. An easy solution, though not

very robust, would simply be for applications to check the source of MotionEvents before determining what to do with them. Another solution could be to disable all touchscreen events while the active stylus is in range. This would have the added benefit of ignoring a hand resting on the display while the active pen is being used, however this would also prevent the touchscreen from working until the stylus is pulled away from the digitizer. Finally, though still not ideal, a common pointer index could be used between the various pointer class devices. This would keep the pointer indexes of different devices from ever being the same and causing unexpected behavior in applications.

Chapter 10

Update to Official SDK

On May 10, 2011 Google released an update to the official Android SDK for Android 3.1 (API level 12), however Google has yet to release the source code for anything later than Android 2.3. The latest version of the SDK adds many new interesting features, such as an ACTION_HOVER_MOVE action to MotionEvent, dispatchGenericMotionEvent() dipatcher, and an onGenericMotionEvent() event handler. Since a "hover" event is *not* a "touch" event, it is not dispatched using dipatchTouchEvent(), but rather dispatchGenericMotionEvent() instead. This provides a way for developers to handle hover events such as mouse or stylus movements without breaking older applications or applications that rely on always receiving an ACTION_DOWN after an ACTION_UP with nothing in between. Although the new hover action was designed for use by the mouse, which is now natively supported in Android, it could easily be used to support the hover feature of an active pen as well.

Since the Android SDK does not yet support active pen input, many features are still missing that are needed to support stylus functionality. This includes pen proximity detection, tool type, and side button state.

Though not visible to the developer, Android 3.1 also adds an on-screen mouse pointer. This was added directly to the InputDispatcher rather than as a separate input monitoring service as discussed in Section 6.6. Android 3.1 does not add anything to the official SDK to allow applications to interact in any way with the pointer, such as changing its look or visibility.

Chapter 11

Future Work

There are certainly more areas to be explored in the field of active pen input and mobile devices. Chiefly, how will active pen input be *used* in such environments? Understanding this can go a long way towards improving how active pen input is implemented and how features are made available to developers. There is a host of applications that would benefit from active pen input, including but certainly not limited to note taking, annotating and document review, textbook markup, drawing, photo editing, graphic design, CAD, and handwriting and drawing recognition.

An important step toward the adoption of active pen input is for a common framework to be laid for application developers. For Android, this would include providing support for active pen input in the Android framework and exposing functionality through the Android SDK. Although Android is open source, a review process is in place in which a patch submission must be approved by the Android team. Additionally, since the source has yet to be released for the latest version of Android (version 3.1), which is designed for tablets, the necessary changes to add active pen input support can only be made by the Android team.

The overall goal of this thesis is to help encourage Google to implement a standard for active pen input in Android. Collaboration with the Android team will continue while the necessary framework additions are developed.

Chapter 12

Conclusion

As the mobile computing industry moves forward, it is important for input methods to evolve as well. Active pen input allows for new yet intuitive and familiar ways to interact with devices. This thesis provided a playing ground for research and implementation of methods to handle active pen input in Android. The Android input framework was analyzed and event communication methods were expanded upon to provide support for active pen input. Additions were made to the Android SDK to allow developers access to the new functionality provided by active pen input. A system wide pointer was implemented to provide a visual representation of the location of an active stylus while hovering above or touching a display. Additionally, the pointer service was designed to be easily expanded to support the use of other input devices that make use of an on-screen pointer, such as a mouse.

It was found that most of the additional functionality could be made with few changes to the Android framework by utilizing and expanding upon pre-existing constructs in event communication. Completely new data that did not fit into existing constructs, such as tool type, could be added throughout the framework

with little difficulty. The Android framework and SDK additions were validated using physical hardware to ensure the viability of active pen input in Android. For the advancement of the mobile computing industry, consumer electronics, and consumer applications, it is essential that active pen input be integrated into the core of mobile operating systems such as Android.

References

[1] Android-x86 - Porting Android to x86. http://www.android-x86.org.

[2] Android-x86 Mouse Patch. http://code.google.com/p/patch-hosting-for-android-x86-support/downloads/detail?name=0001-fixed-different-build-breaks-added-mouse-cursor-sup.patch.

[3] Android Open Source Project Git Repository. http://android.git.kernel.org.

[4] R. Cicchino and D. Mirliss. Tablet pcs: A powerful teaching tool. In *World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2004*, pages 543–548. AACE: Chesapeake, VA, 2004.

[5] Hewlett-Packard. The Case for the Tablet PC in Health Care. http://www.hp.com/sbso/solutions/healthcare/hp_tablet_whitepaper.

[6] HTC Flyer Product Overview. http://www.htc.com/www/product/flyer/overviewa.html.

[7] Video: HTC Flyers Deep Pen Integration Makes This Android Slate Stand Out. http://blog.laptopmag.com/htc-flyer-hands-on.

[8] HTC Unveils HTC Flyer, the First Tablet with HTC Sense. http://www.htc.com/us/press/htc-unveils-htc-flyertrade-the-first-tablet-with-htc-sensetrade/31.

[9] Linux Wacom Project. http://linuxwacom.sourceforge.net.

[10] Microsoft Demonstrates Tablet PC Technology For Enterprise Computing Applications. http://www.microsoft.com/presspass/press/2000/Nov00/TabletPCPR.mspx.

[11] K. Mock. Teaching with Tablet PC's. *J. Comput. Small Coll.*, 20:17–27, December 2004.

[12] Motion Computing's Motion C5v. http://www.motioncomputing.com/products/tablet_pc_c5.asp.

[13] N-trig Elevates the Slate, Netbook, and Tablet Experience through its DuoSense Digital Pencil Further Enhancing Creativity and Productivity. http://www.cellulartec.com/?p=2068.

[14] DuoSense Overview. http://www.ntrig.com/Content.aspx?Page=DualModeTechnology.

[15] Android Native Development Kit. http://developer.android.com/sdk/ndk/index.html.

[16] Android Software Development Kit. http://developer.android.com/sdk/index.html.

[17] Wacom Components: EMR (Electro-Magnetic Resonance) Technology. http://www.wacom-components.com/english/technology/emr.html.

[18] What is Android? http://developer.android.com/guide/basics/what-is-android.html.