2016

Android Thumb Click

BY:

MARIA KOZAKOV MICHAEL VALERSHTEIN

SUPERVISOR- NOAM SHALEV

TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING

Contents

1. Introduction

Our project goal is to integrate a new kind of touch into the android system, called FatTouch.

FatTouch performed by touching the screen with full thumb for short period of time and will enable adding new functionalities to user interactions with touch devices.

2. Motivation

Touch devices, such as smartphones and tablets are integral part of everyday life. As if today, touch is the most common way to interact with devices. There is limited number of touch gestures are at user’s disposal. As complexity of touch devices and applications grow, introduction of new interaction ways is inevitable.

Adding new touch based interaction methods, have proven to open new possibilities to application developers. For example, adding multi-touch made it possible to use pinch to zoom, develop games which require using use of multiple and simultaneous touch.

3. Background

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:

- Linux kernel
- Android runtime
- System libraries
- Application framework
- Applications

The components structure is described in Figure1:

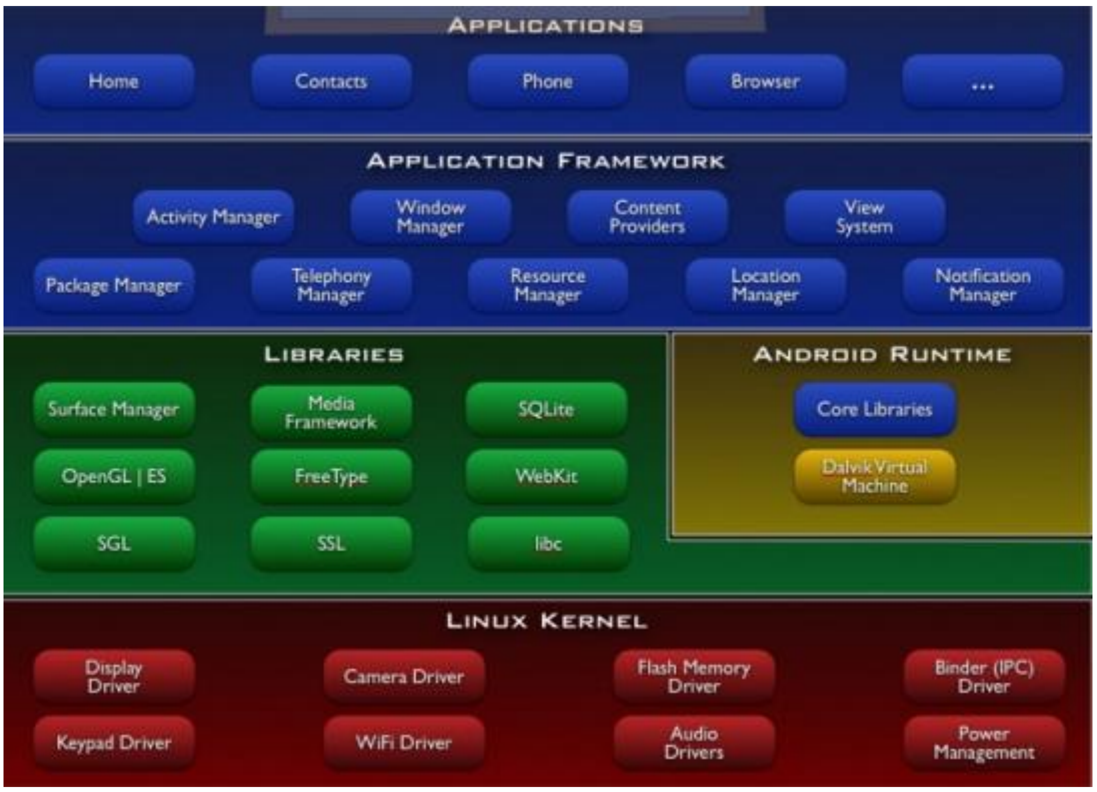


FIGURE 1- ANDROID MAIN COMPONENTS

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 Android 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, 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. 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).

Android SDK exposes several ways to deal with touch:

- Use of GestureDetector. Application developers have an option to use GestureDetector class for detecting common gestures.
- Override OnTouchEvent method of View class and define what to do when certain touch events occur.

4. Implementation Requirements

Introducing new features is not an easy task. Current users need to be able to get accustomed to them easily, they must be intuitive and easy to use. Users have physiological differences which must be taken into account at planning stage, e.g. differences in thumb dimensions.

Application developers, have to be taken into account as well. Developing applications that use new feature need to be made as easy as possible with minimal performance impact. New applications shall not be forced to use the feature. As for existing applications, the feature need not to require any changes in current implementation.

At this project we made changes in Android source code. For these changes to be integrated easily into Android repositories, we need to take into account current Android architecture and add our feature without changing it.

5.Android input system

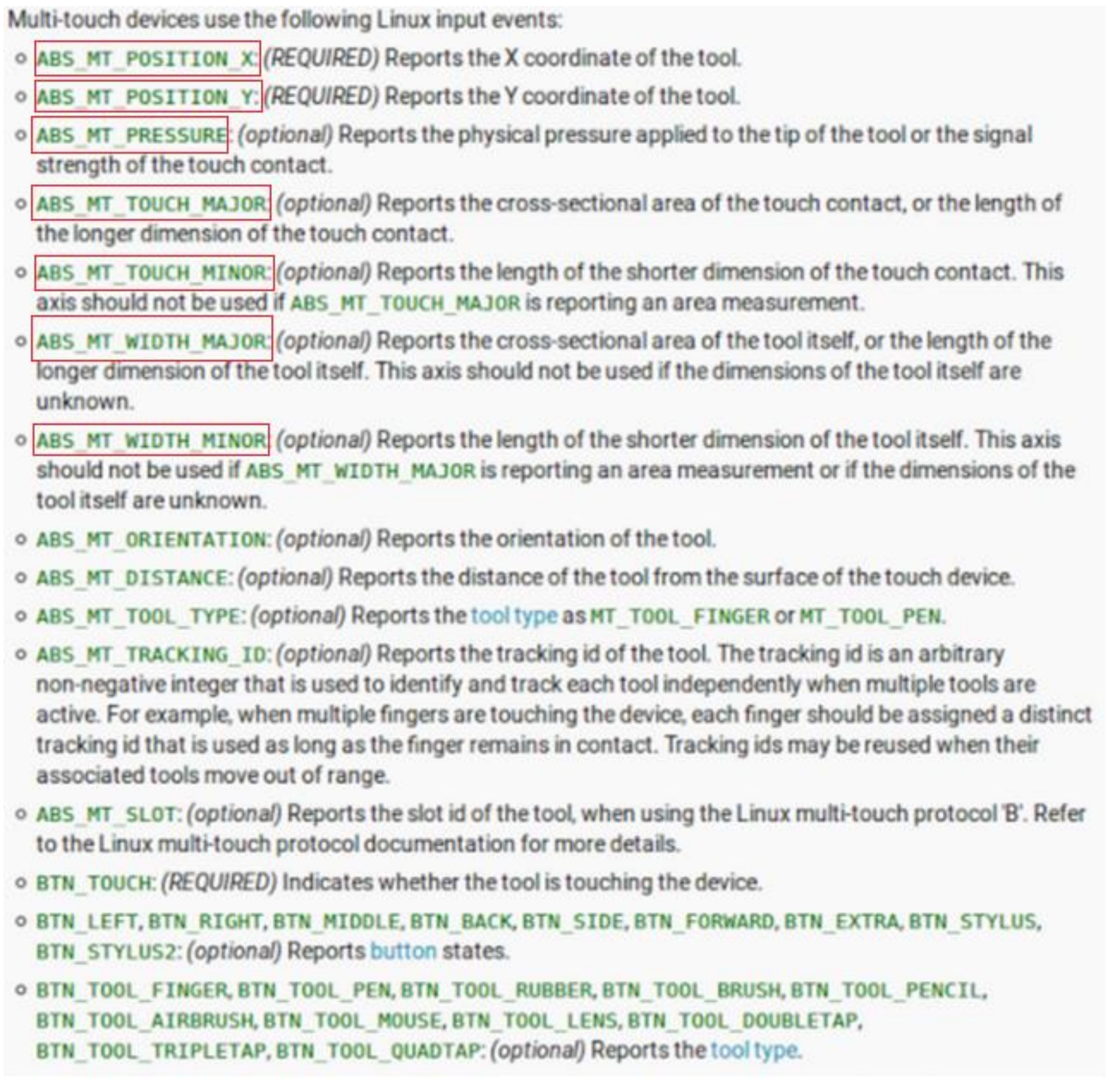
5.1 Linux touchscreen driver

All Android OS versions are based on Linux kernel. Linux kernel is the lowest level of Android OS which interfaces with the hardware of host device. It is responsible for interfacing all of applications that are running in “user mode” down to the physical hardware, and allowing processes to get information from each other using inter-process communication (IPC).

Each device that use touch screen has a driver, which is compiled into the Linux kernel. The driver reads physical drivers and translates the data to events. These events are reported to the Android lower layers.

There is a vast variety of touch screens available which are integrated into many touch based devices. Each device has Linux kernel. For every such device a touch screen driver must be implemented.

Android OS requires touchscreen driver of meeting certain criterias , (see figure 2- Events which are potentially relevant to detecting FatTouch are marked)


Multi-touch devices use the following Linux input events:

- ABS_MT_POSITION_X: *(REQUIRED)* Reports the X coordinate of the tool.
- ABS_MT_POSITION_Y: *(REQUIRED)* Reports the Y coordinate of the tool.
- ABS_MT_PRESSURE: *(optional)* Reports the physical pressure applied to the tip of the tool or the signal strength of the touch contact.
- ABS_MT_TOUCH_MAJOR: *(optional)* Reports the cross-sectional area of the touch contact, or the length of the longer dimension of the touch contact.
- ABS_MT_TOUCH_MINOR: *(optional)* Reports the length of the shorter dimension of the touch contact. This axis should not be used if ABS_MT_TOUCH_MAJOR is reporting an area measurement.
- ABS_MT_WIDTH_MAJOR: *(optional)* Reports the cross-sectional area of the tool itself, or the length of the longer dimension of the tool itself. This axis should not be used if the dimensions of the tool itself are unknown.
- ABS_MT_WIDTH_MINOR: *(optional)* Reports the length of the shorter dimension of the tool itself. This axis should not be used if ABS_MT_WIDTH_MAJOR is reporting an area measurement or if the dimensions of the tool itself are unknown.
- ABS_MT_ORIENTATION: *(optional)* Reports the orientation of the tool.
- ABS_MT_DISTANCE: *(optional)* Reports the distance of the tool from the surface of the touch device.
- ABS_MT_TOOL_TYPE: *(optional)* Reports the tool type as MT_TOOL_FINGER or MT_TOOL_PEN.
- ABS_MT_TRACKING_ID: *(optional)* Reports the tracking id of the tool. The tracking id is an arbitrary non-negative integer that is used to identify and track each tool independently when multiple tools are active. For example, when multiple fingers are touching the device, each finger should be assigned a distinct tracking id that is used as long as the finger remains in contact. Tracking ids may be reused when their associated tools move out of range.
- ABS_MT_SLOT: *(optional)* Reports the slot id of the tool, when using the Linux multi-touch protocol 'B'. Refer to the Linux multi-touch protocol documentation for more details.
- BTN_TOUCH: *(REQUIRED)* Indicates whether the tool is touching the device.
- BTN_LEFT, BTN_RIGHT, BTN_MIDDLE, BTN_BACK, BTN_SIDE, BTN_FORWARD, BTN_EXTRA, BTN_STYLUS, BTN_STYLUS2: *(optional)* Reports button states.
- BTN_TOOL_FINGER, BTN_TOOL_PEN, BTN_TOOL_RUBBER, BTN_TOOL_BRUSH, BTN_TOOL_PENCIL, BTN_TOOL_AIRBRUSH, BTN_TOOL_MOUSE, BTN_TOOL_LENS, BTN_TOOL_DOUBLETAP, BTN_TOOL_TRIPLETAP, BTN_TOOL_QUADTAP: *(optional)* Reports the tool type.

FIGURE 2-ANDROID DEFINED REQUIREMENTS FOR THE DATA REPORTED BY A TOUCHSCREEN DRIVER , PARTIAL REQUIREMENTS LIST:

Our initial thought was to assist in ABS_TOUCH_MAJOR, ABS_TOUCH_MINOR, ABS_WIDTH_MAJOR and ABS_WIDTH_MINOR for touch analysis. Unfortunately we found out later that these values are not actually reported by the driver. Pressure event is reported, and we elaborate on this property in the following chapters.

The device used for this project is Nexus7. The touch screen driver or this device is located at /tegra/drivers/input/touchscreen/ektf3k.c.

The function responsible of reporting touch events:

```
    static void elan_ktf3k_ts_report_data(struct i2c_client *client, uint8_t *buf)
{
   struct elan_ktf3k_ts_data *ts = i2c_get_clientdata(client);
   struct input_dev *idev = ts->input_dev;
   uint16_t x, y, touch_size, pressure_size;
   uint16_t fbits=0, checksum=0;
```

```
    uint8_t i, num;
    static uint8_t size_index[10] = {35, 35, 36, 36, 37, 37, 38, 38, 39, 39};
    uint16_t active = 0;
    uint8_t idx=IDX_FINGER;
    num = buf[2] & 0xf;
    for (i=0; i<34;i++)
        checksum +=buf[i];

      if ((num < 3) || ((checksum & 0x00ff) == buf[34])) {
        fbits = buf[2] & 0x30;
        fbits = (fbits << 4) | buf[1];
         for(i = 0; i < FINGER_NUM; i++){
            active = fbits & 0x1;
            if(active || mTouchStatus[i]){
           input_mt_slot(ts->input_dev, i);
                input_mt_report_slot_state(ts->input_dev, MT_TOOL_FINGER, active);
                if(active){
                    elan_ktf3k_ts_parse_xy(&buf[idx], &x, &y);
             x = x > ts->abs_x_max ? 0 : ts->abs_x_max - x;
             y = y > ts->abs_y_max ? ts->abs_y_max : y;
               touch_size = ((i & 0x01) ? buf[size_index[i]] : (buf[size_index[i]] >> 4)) & 0x0F;
             pressure_size = touch_size << 4; // shift left touch size value to 4 bits for max pressure value
255
                    input_report_abs(idev, ABS_MT_TOUCH_MAJOR, touch_size);
                    input_report_abs(idev, ABS_MT_PRESSURE, pressure_size);
                    input_report_abs(idev, ABS_MT_POSITION_X, y);
                    input_report_abs(idev, ABS_MT_POSITION_Y, x);
                    if(unlikely(gPrint_point)) touch_debug(DEBUG_INFO, "[elan] finger id=%d X=%d
y=%d size=%d pressure=%d\n", i, x, y, touch_size, pressure_size);
             }
         }
         mTouchStatus[i] = active;
            fbits = fbits >> 1;
            idx += 3;
        }
          input_sync(idev);
    } // checksum
    else {
        checksum_err +=1;
        touch_debug(DEBUG_ERROR, "[elan] Checksum Error %d byte[2]=%X\n", checksum_err,
buf[2]);
    }

    return;
}
```

Reported data potentially relevant for FatTouch detection is marked.

After researching the possibility of integrating FatTouch detection at touch screen driver we came to following conclusions:

- Adding FatTouch analysis to touch screen driver ~~will~~ requires adding a new input event type. Each step of input processing system would need to recognize this event and pass it to the next step respectively; thus requires changes at most of touch reporting pipeline.
- Would harm performance- touch analysis here will cause major overhead.

Not all touch events reported by the driver are relevant to FatTouch detection. Trying to detect FatTouch here would cause the driver to analyze each touch screen event.

- Touch screen drivers are implemented for a variety of devices. Each driver implementation would have to be changed to support FatTouch.

5.2- Android Native

From the Linux kernel the touch information proceeds to the Android Native layer.

The Android native input subsystem is made up of a series of C++ classes that forwards the semi-processed information to the Java classes that completes the input system as an Android framework. The input framework detects, filters, categorizes, and injects input events into the currently running Activity or system component.

Input events are detected and read by the EventHub (see section 5.2.5).

The InputReader (section 5.2.3) continually acquires new events from the EventHub and performs initial filtering and categorization on input events based on the device that event is from. This essentially turns “raw” input events into “cooked” events. These “cooked” events are then added to the InputDispatcher (section 5.2.4) queue. The InputDispatcher 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.

5.2.1 Starting up the input native pipeline

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.

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. 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.

5.2.2 InputManager

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

5.2.3 InputReader

The role of the InputReader is to processes raw input events and to send the cooked event data to an input dispatcher.

The InputReader continuously retrieves a raw event from the EventHub and processes it. It keeps track of a list of devices that are currently connected along with their capabilities. Each 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. 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. After processing the raw input event, each InputMapper notifies the InputDispatcher of its input event.

InputMapper examples:

- 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)

The InputMapper which is relevant to FatTouch is SingleTouchInputMapper.

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 native constants are used by the InputReader and InputDispatcher

5.2.4 InputDispatcher

The InputDispatcher can be broken down into two main parts: one part continually reads from the internal queue of “EventEntry” objects 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.

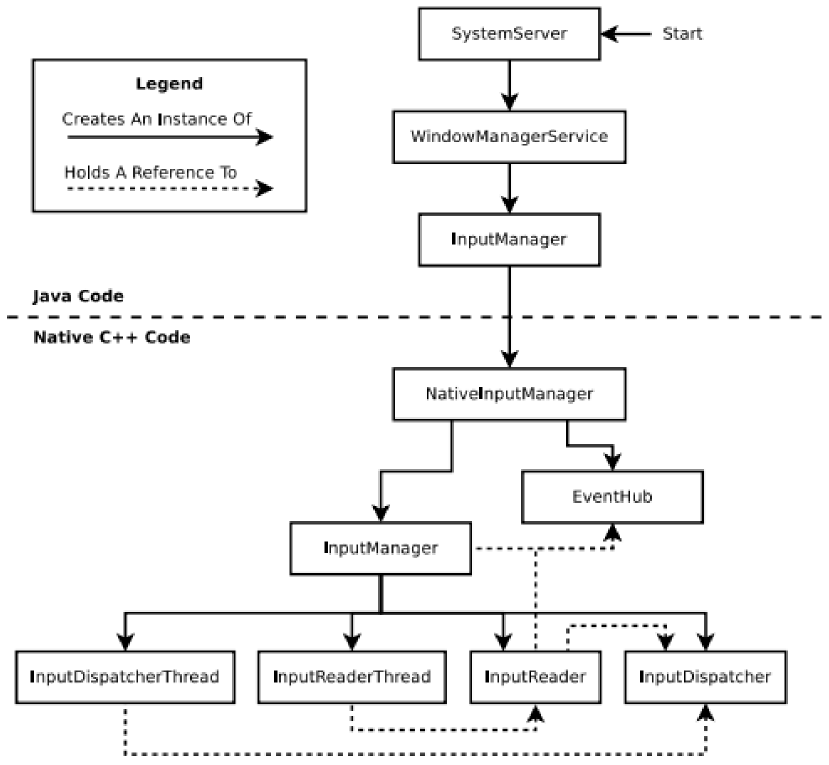


FIGURE 3-EVENT PROPAGATION BETWEEN C++ AND JAVA

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).

5.2.5 EventHub.cpp

First class to access touch information is the EventHub. 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. 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 (see appenix). This is called by the InputReader and is guaranteed to be called by a single caller. 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.

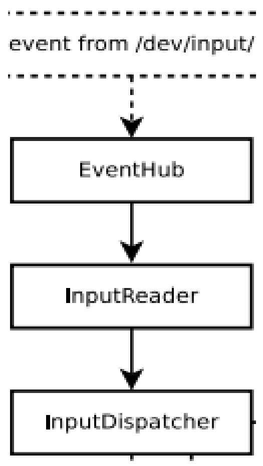


FIGURE 4-INPUT NATIVE FLOW

5.3 - Android low levels

All touch events are received at Android lower layers as instance of MotionEvent class. MotionEvent class serves as a wrapper for events dispatched from Native Android. Motion events describe movements in terms of an action code and a set of axis values. The action code specifies the state change that occurred such as a pointer going down or up. The axis values describe the position and other movement properties.

For example, when the user first touches the screen, the system delivers a touch event to the appropriate View with the action code ACTION_DOWN and a set of axis values that include the X and Y coordinates of the touch and information about the pressure, size and orientation of the contact area.

Some devices can report multiple movement traces at the same time. Multi-touch screens emit one movement trace for each finger. The individual fingers or other objects that generate movement traces are referred to as pointers. Motion events contain information about all of the pointers that are currently active even if some of them have not moved since the last event was delivered.

The number of pointers only ever changes by one as individual pointers go up and down, except when the gesture is canceled.

Each pointer has a unique id that is assigned when it first goes down (indicated by ACTION_DOWN or ACTION_POINTER_DOWN). A pointer id remains valid until the pointer eventually goes up (indicated by ACTION_UP or ACTION_POINTER_UP) or when the gesture is canceled (indicated by ACTION_CANCEL).

Touch resolution is performed at ACTION_UP or ACTION_POINTER_UP motion event with an exception of Long Press touch timeout expiration.

The MotionEvent class provides many methods to query the position and other properties of pointers, such as getX(int), getY(int), getAxisValue(int), getPointerId(int), getToolType(int), and many others. Most of these methods accept the pointer index as a parameter rather than the pointer id. The pointer index of each pointer in the event ranges from 0 to one less than the value returned by getPointerCount().

After motionEvent is generated, it is passed to relevant View class, and the view's OnTouchEvent method invoked.

View class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The ViewGroup subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties.

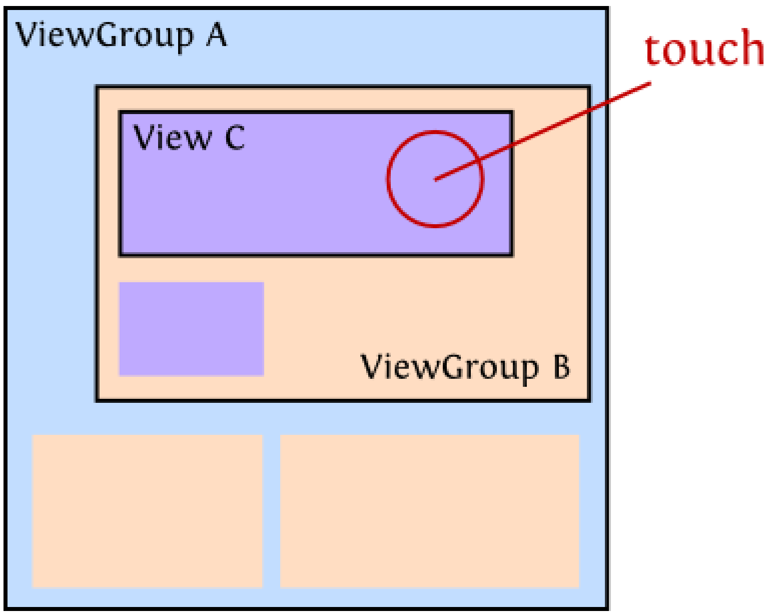
All user interface elements in an Android app are built using View and ViewGroup objects.

5.3.1 Application usage of touch events

Applications use touch to interact with users. In order to create flow for certain touch events, application developers need to intercept and analyze them. The UI layout is comprised of several View and VewGroup classes. Each ViewGroup can contain several ViewGoup or View classes thus creating hierarchy of user interface components.

When a user touches screen of the device, an ACTION_DOWN event is sent to the View class in which the touch occurred. This class can decide if it is interested in receiving the rest of events related to this touch (e.g. changes in position, pressure etc.). In order to be able to decide, the class needs to override onTouchEvent(MotionEvent e) method. In case the view is interested in rest of the events, it shall return true. If some View or ViewGroup class returns true for onTouchEvent(MotionEvent e) ,classes above it will not receive the event.

For example, if we have the following layout:



The touch occurred at the red circle. As the touch occurs, View C receives a MotionEvent of ACTION_DOWN. If View is interested in rest of events regarding this touch, it would return true in onTouchEvent(MotionEvent e). If View C is interested in such events, upper layers of the UI will not receive the event.

Each event contains information regarding the touch- location, pressure, size etc. View class which is interested in certain touch events can analyze them or use GestureDetector class for detecting gestures.

Since every touch event is processed in View.onTouchEvent() it is possible to add FatTouch resolution at the View class, but this method is very unpractical. Every application that overrides View.onTouchEvent method, will not be able to use current FatTouch resolution implementation. Thus such applications are forced either to use the whole onTouchEvent method, or implement the resolution in its source code. In other words, FatTouch is not modular enough for an easy use.

5.4 Android Framework and Gestures

Touchscreens of touch based devices, such as the iPad, utilize multi-touch technology, with gestures acting as the main form of user interface. Many touchpads, which in laptops replace the traditional mouse, have similar gesture support. For example, a common gesture is to use two fingers in a downwards or upwards motion to scroll the currently active page. The rising popularity of touchscreen interfaces has led to gestures becoming a more standard feature in computing. An increasing number of products like smartphones, tablets, laptops or desktop computers have functions that are triggered by multi-touch gestures.

5.4.1 Gesture detector

Android provides the GestureDetector class for detecting common gestures.

Currently this class provides detection of the following gestures: Scroll, Long press, Fling and Double Tap.

Nested Classes

The use of the GestureDetector is through its nested interfaces and classes, an application has to implement or override.

interface	GestureDetector.OnContextClickListener	The listener that is used to notify when a context click occurs.
interface	GestureDetector.OnDoubleTapListener	The listener that is used to notify when a double-tap or a confirmed single-tap occur.
interface	GestureDetector.OnGestureListener	The listener that is used to notify when gestures occur
class	GestureDetector.SimpleOnGestureListener	A convenience class to extend when you only want to listen for a subset of all the gestures.

5.4.1.1 Usage

To use GestureDetector, we need to do 3 things

Use of GestureDetector is accomplished by:

- Implementing Listeners
- Overriding all the callback methods of listener interface
- Binding the GestureDetector to gesture listener

5.4.1.2 Implementing Listeners and Overriding methods

The first thing to do in order to start detecting the gesture events is to implement the listeners. There are 2 ways to do that:

1. Make our Activity class implement GestureDetector.OnDoubleTapListener (for double tap gesture detection) and GestureDetector.OnGestureListener interfaces and implement all the abstract methods.
2. Write a custom class as a nested class of our Activity or some other external class that extends the aforementioned interfaces or extends the GestureDetector.SimpleOnGestureListener class. The advantage of SimpleOnGestureListener nested class is that it implements all the methods from those 2 interfaces but does nothing. So if we want to deal with and process only a few (subset) gestures then we can take this approach.

Example of the 1st approach:

```
class CustomGestureDetector implements
GestureDetector.OnGestureListener,

GestureDetector.OnDoubleTapListener {

    @Override
    public boolean onSingleTapConfirmed(MotionEvent e) {
        mGestureText.setText("onSingleTapConfirmed");
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent e) {
        mGestureText.setText("onDoubleTap");
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent e) {
        mGestureText.setText("onDoubleTapEvent");
        return true;
    }

    @Override
```

```
        public boolean onDown(MotionEvent e) {
            mGestureText.setText("onDown");
            return true;
        }

        @Override
        public void onShowPress(MotionEvent e) {
            mGestureText.setText("onShowPress");
        }

        @Override
        public boolean onSingleTapUp(MotionEvent e) {
            mGestureText.setText("onSingleTapUp");
            return true;
        }

        @Override
        public boolean onScroll(MotionEvent e1, MotionEvent e2, float
        distanceX, float distanceY) {
            mGestureText.setText("onScroll");
            return true;
        }

        @Override
        public void onLongPress(MotionEvent e) {
            mGestureText.setText("onLongPress");
        }

        @Override
        public boolean onFling(MotionEvent e1, MotionEvent e2, float
        velocityX, float velocityY) {
            mGestureText.setText("onFling");
            return true;
        }
    }
```

5.4.1.3 Binding

Next steps are to create a `GestureDetector` object and attach the listener objects to it. This will can be done in the `onCreate()` method of an `Activity` class:

```
private TextView mGestureText;
private GestureDetector mGestureDetector;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_gesture);
    mGestureText = (TextView) findViewById(R.id.gestureStatus);

    // Create an object of our Custom Gesture Detector Class
    CustomGestureDetector customGestureDetector = new
CustomGestureDetector();
    // Create a GestureDetector
    mGestureDetector = new GestureDetector(this,
customGestureDetector);
    // Attach listeners that'll be called for double-tap and
related gestures

mGestureDetector.setOnDoubleTapListener(customGestureDetector);
}
```

Implementing onTouchEvent()

The gesture detectors won't fire yet. This is because the touch events are not yet re-routed to the gesture detectors. In order to do that, `onTouchEvent()` of the Activity must be overridden.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    mGestureDetector.onTouchEvent(event);

    return super.onTouchEvent(event);
}
```

5.4.2 FatTouch and Gestures

In the beginning of the project, our perception of FatTouch was as of a separate kind of input, as much as touching the screen with a finger, or with a pointer tool are different types of input, but after we became familiar with the whole android input system - Gesture detector in particular we changed our approach.

Long press is a gesture we found to have a certain similarity to Fat Touch – both of these touches are performed with one finger, and both do not involve movement of the finger on the screen surface. This led us to understanding that Fat touch can be also perceived as a gesture. This approach has a number of clear benefits and we will present them in section 5.5 .

5.5 Implementation

After we grasped the android touch input system in its whole, from physical touch driver to the application layer, we came to a conclusion regarding the most suitable place to insert FatTouch detection and resolution

- × Linux Driver

  Time consuming, all Linux kernels need to be changed. Will affect the whole input pipeline

- × Android Native
  Suffers from the same problem as implementation at Linux driver. Might suffer even larger overhead since here the classes handle all types of events.
- × Low level
  Forces application developers wanting the option of FatTouch, also to use View.onTouchEvent ()
- ✓ Framework-
  Implementing FatTouch here, allows developers a modular usage of FatTouch. Using FatTouch here will not harm performance.
  Another important benefit, is the similarity of use between FatTouch and another gestures.

We came to conclusion that integrating FatTouch into the GestureDetector is the optimal solution

5.5.1 Resolving FatTouch

We needed to find characteristics which distinguish the FatTouch from regular touch.
We printed all touch events characteristics to the android log, for each touch event occurred.
From the first look couple of features stood out- touch size and pressure. For further and more precise analysis we used a group of people. Each person performed 100 regular and 100 FatTouches.
It became apparent that pressure and size have much higher values for FatTouch. The difference in size value between regular and fat touch was an expected result, as the FatTouch is a touch of a full thumb, and for obvious reasons it has a bigger touch size. On the other hand, the difference in pressure value was a surprise, since even for a light Fat touch the value was much higher than regular touch.
Based on these results, it became obvious that our problem of FatTouch resolution can be reduced to a statistical classification problem.

5.5.1.1 SVM

In machine learning, **support vector machines** (**SVM**, also **support vector networks**) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked for belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier.

An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

Given two categories of point sets, there are multiple ways to divide the space into the two categories, so each point in this given space will fall into one and only category. In our case we have two dimensional space, in which the axis are touch size and touch pressure. SVM offers an optional solution in a sense that it finds the dividing line (hyperplane), which is as far as possible from its nearest points. This way the categorization will be less noise sensitive, and abnormal measurements will still stay in the right class.

SVM Linear Algorithm

Let's introduce the notation used to define formally a hyperplane:

$$f(x) = \beta_0 + \beta^T x,$$

where β is known as the *weight vector* and β_0 as the *bias*.

The optimal hyperplane can be represented in an infinite number of different ways by scaling of β and β_0. As a matter of convention, among all the possible representations of the hyperplane, the one chosen is

$$|\beta_0 + \beta^T x| = 1$$

where x symbolizes the training examples closest to the hyperplane. In general, the training examples that are closest to the hyperplane are called **support vectors**. This representation is known as the **canonical hyperplane**.

Now, we use the result of geometry that gives the distance between a point x and a hyperplane (β, β_0):

$$\mathrm{distance} = \frac{|\beta_0 + \beta^T x|}{||\beta||}.$$

In particular, for the canonical hyperplane, the numerator is equal to one and the distance to the support vectors is

$$\mathrm{distance}_{\text{ support vectors}} = \frac{|\beta_0 + \beta^T x|}{||\beta||} = \frac{1}{||\beta||}.$$

Recall that the margin introduced in the previous section, here denoted as M, is twice the distance to the closest examples:

$$M = \frac{2}{||\beta||}$$

Finally, the problem of maximizing M is equivalent to the problem of minimizing a function $L(\beta)$ subject to some constraints. The constraints model the requirement for the hyperplane to classify correctly all the training examples x_i. Formally,

$$\min_{\beta, \beta_0} L(\beta) = \frac{1}{2}||\beta||^2 \text{ subject to } y_i(\beta^T x_i + \beta_0) \geq 1 \ \forall i,$$

where y_i represents each of the labels of the training examples.

This is a problem of Lagrangian optimization that can be solved using Lagrange multipliers to obtain the weight vector β and the bias β_0 of the optimal hyperplane.

5.5.1.2 SVM FatTouch Resolution

The most critical conclusion from our perspective is the fact that if the two sets of points cannot be separated (and are in fact belong to the same category), the SVM algorithm will not converge, and no solution will be offered.

Using MatLab, we defined two vectors- Regular touch and fat touch. We applied SVMTRAIN MatLab function on the vectors and got the following results:

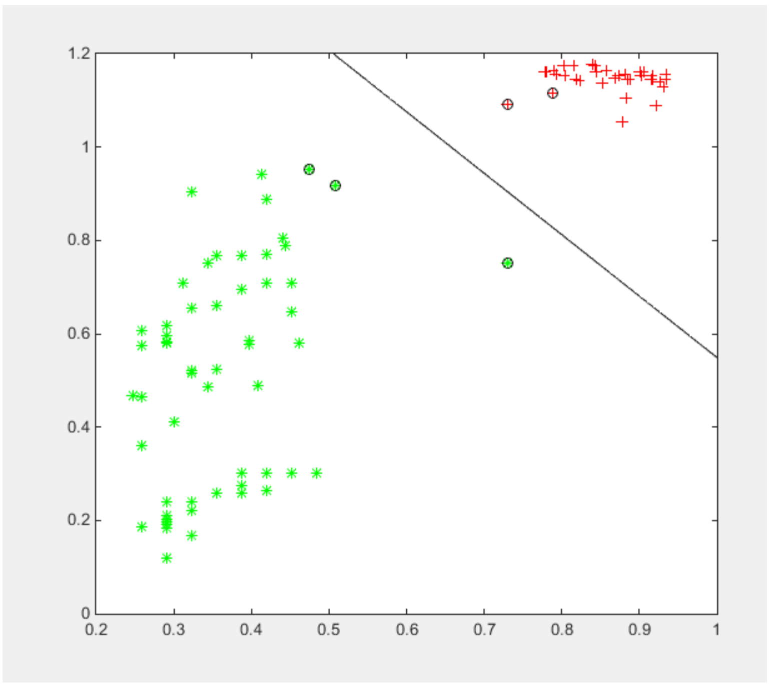


FIGURE 5 - SVM RESULTS

Where Red marks are FatTouches, and green marks are regular. X axis are touch size and Y axis are touch pressure.

Here it will be a good place to mention that the high pressure values in regular touches were received when the user deliberately applied force to the touch.

From the graph it is clear that in fact, FatTouch and regular touch features can be separated into two categories.

From the results of SVM we came to a conclusion that pressure feature might be sufficient for detecting FatTouch. In order to avoid as much as possible relying on the size of the thumb, since it varies depending on person's age height etc., we tested the FatTouch while examining pressure only.

5.5.2 Implementation in GestureDetector

Our implementation of FatTouch detector was added to the GestureDetector class in a way that is in line with the current Android structure, and so the implementation was integrated in a natural way with the android framework and GestureDetector class.

We have added an interface OnFatTouchListener to GestureDetector.java,

```
public interface OnFatTouchListener{
    boolean OnFatTouch(MotionEvent) {
}
```

Now nested classes of GestureDetector look like that

interface	GestureDetector.OnContextClickListener	The listener that is used to notify when a context click occurs.
interface	GestureDetector.OnDoubleTapListener	The listener that is used to notify when a double-tap or a confirmed single-tap occur.
interface	GestureDetector.OnGestureListener	The listener that is used to notify when gestures occur
class	GestureDetector.SimpleOnGestureListener	A convenience class to extend when you only want to listen for a subset of all the gestures.
interface	GestureDetector.OnFatTouchListener	The listener that is used to notify when Fattouch occur

We added a member of the interface to the GestureDetector

```
    private final Handler mHandler;
    private final OnGestureListener mListener;
    private OnDoubleTapListener mDoubleTapListener;
    private OnContextClickListener mContextClickListener;
    private OnFatTouchListener mOnFatTouchListener;
```

We added FatTouch resolution to the onTouchEvent() function

```
 private const double FATTOUCH_LIMIT  = 0.85;
 public boolean onTouchEvent(MotionEvent ev) {
....
 case MotionEvent.ACTION_UP:
    mAvgPressure = mAvgPressure / mPressureCounter;

    if(mAvgPressure >= FATTOUCH_LIMIT && !mInLongPress &&
                mOnFatTouchListener!=null)
    {
        mOnFatTouchListener.OnFatTouch(ev);

    }
....

 case MotionEvent.ACTION_DOWN:
        mAvgPressure = ev.getPressure();
        mPressureCounter = 1;

 case MotionEvent.ACTION_MOVE:
    mAvgPressure += ev.getPressure();
    mPressureCounter++;
```

Lastly we added a function

```
public void setOnFatTouchListener(OnFatTouchListener
onFatTouchListener) {
         mDoubleTapListener = onFatTouchListener;
    }
```

The usage of FatTouch, is identical to the usage of the other gestures.

In order to handle FatTouch event, the application will need to implement OnFatTouchListener interface, and use gestureDetector's setOnFatTouchListener function in order to register for the events.

5.6 Proof of Concept

After integrating FatTouch in GestureDetector class, we have created an example application. The application is based on one of example applications available in the Android SDK. We altered an application which performs image manipulation. On FatTouch, the image is changed between fisheye and negative, while in long press it changes back to the original image.

Instantiation of GestureDetector and OnFatTouchListener:

```
private MyOnFatTouchListener mOnFatTouchListener ;
  private GestureDetector mGesture;
  public void onCreate(Bundle savedInstanceState) {
...
  mGesture = new GestureDetector(getApplicationContext(), this);
  mOnFatTouchListener = new MyOnFatTouchListener ();
  mGesture.setOnFatTouchListener(mOnFatTouchListener );
...
  public boolean onTouchEvent(MotionEvent event) {
       mGesture.onTouchEvent(event);
        return true;
 };
```

Implementation of OnFatTouchListener:

```
 private class MyOnFatTouchListener implements OnFatTouchListener {
       @Override
       public void OnFatTouch(MotionEvent e) {
           if(mCurrentEffect==R.id.fisheye){
               mCurrentEffect=R.id.negative;
            }else {
               mCurrentEffect=R.id.fisheye;
            }
            // onDrawFrame(GL10 gl);
            setCurrentEffect(mCurrentEffect);
            mEffectView.requestRender();
       }
   }
```

The application was tested by several people, each of them successfully used the FatTouch feature.

6. Conclusions

During our work, we have considered adding FatTouch detection and resolution to several stages in the input pipeline. The stage we found most suitable is the GestureDetector class located in the android framework.

This solution offers number of benefits. It has backwards computability and will not harm any applications using motion events, view and GestureDetector. It is aligned with the current android architecture, and naturally blends with the other existing gestures (such as long press). There is almost no overhead at all, in adding FatTouch to GestureDetector. Furthermore our implementation of FatTouch resolution is very easy to use for any application developer

We had succeeded in our goal of identifying FatTouch. The FatTouch feture was integrated into an existing application and successfully tested by several people!

We believe that FatTouch is a useful feature, and can promote developing diverse applications.

8. Bibliography

[1] http://codetheory.in/android-gesturedetector/

[2] - Andrew Hughes , the Faculty of California Polytechnic State University. ACTIVE PEN INPUT AND THE ANDROID INPUT FRAMEWORK, 2011

[3] http://developer.android.com/guide/index.html

[4] https://en.wikipedia.org/

[5] http://balpha.de/2013/07/android-development-what-i-wish-i-had-known-earlier/

[6] http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

7. APPENDIX

7.1 EventHub.getEvent()

```
bool EventHub::getEvent(RawEvent* outEvent) {
    outEvent->deviceId = 0;
    outEvent->type = 0;
    outEvent->scanCode = 0;
    outEvent->keyCode = 0;
    outEvent->flags = 0;
    outEvent->value = 0;
    outEvent->when = 0;
    // Note that we only allow one caller to getEvent(), so don't
need
    // to do locking here...  only when adding/removing devices.
    if (!mOpened) {
        mError = openPlatformInput() ? NO_ERROR : UNKNOWN_ERROR;
        mOpened = true;
        mNeedToSendFinishedDeviceScan = true;
    }
    for (;;) {
        // Report any devices that had last been added/removed.
        if (mClosingDevices != NULL) {
            Device* device = mClosingDevices;
            LOGV("Reporting device closed: id=%d, name=%s\n",
                 device->id, device->path.string());
            mClosingDevices = device->next;
            if (device->id == mBuiltInKeyboardId) {
                outEvent->deviceId = 0;
            } else {
                outEvent->deviceId = device->id;
            }
            outEvent->type = DEVICE_REMOVED;
            outEvent->when = systemTime(SYSTEM_TIME_MONOTONIC);
            delete device;
            mNeedToSendFinishedDeviceScan = true;
            return true;
        }
        if (mOpeningDevices != NULL) {
            Device* device = mOpeningDevices;
            LOGV("Reporting device opened: id=%d, name=%s\n",
                 device->id, device->path.string());
            mOpeningDevices = device->next;
            if (device->id == mBuiltInKeyboardId) {
                outEvent->deviceId = 0;
            } else {
                outEvent->deviceId = device->id;
            }
            outEvent->type = DEVICE_ADDED;
            outEvent->when = systemTime(SYSTEM_TIME_MONOTONIC);
            mNeedToSendFinishedDeviceScan = true;
            return true;
        }
        if (mNeedToSendFinishedDeviceScan) {
            mNeedToSendFinishedDeviceScan = false;
            outEvent->type = FINISHED_DEVICE_SCAN;
            outEvent->when = systemTime(SYSTEM_TIME_MONOTONIC);
            return true;
        }
```

```
            // Grab the next input event.
            for (;;) {
                // Consume buffered input events, if any.
                if (mInputBufferIndex < mInputBufferCount) {
                    const struct input_event& iev =
mInputBufferData[mInputBufferIndex++];
                    const Device* device = mDevices[mInputFdIndex];
                    LOGV("%s got: t0=%d, t1=%d, type=%d, code=%d, v=%d",
device->path.string(),
                         (int) iev.time.tv_sec, (int) iev.time.tv_usec,
iev.type, iev.code, iev.value);
                    if (device->id == mBuiltInKeyboardId) {
                        outEvent->deviceId = 0;
                    } else {
                        outEvent->deviceId = device->id;
                    }
                    outEvent->type = iev.type;
                    outEvent->scanCode = iev.code;
                    outEvent->flags = 0;
                    if (iev.type == EV_KEY) {
                        outEvent->keyCode = AKEYCODE_UNKNOWN;
                        if (device->keyMap.haveKeyLayout()) {
                            status_t err = device->keyMap.keyLayoutMap-
>map(iev.code,
                                    &outEvent->keyCode, &outEvent-
>flags);
                            LOGV("iev.code=%d keyCode=%d flags=0x%08x
err=%d\n",
                                    iev.code, outEvent->keyCode,
outEvent->flags, err);
                        }
                    } else {
                        outEvent->keyCode = iev.code;
                    }
                    outEvent->value = iev.value;
                    // Use an event timestamp in the same timebase as
                    // java.lang.System.nanoTime() and
android.os.SystemClock.uptimeMillis()
                    // as expected by the rest of the system.
                    outEvent->when = systemTime(SYSTEM_TIME_MONOTONIC);
                    return true;
                }
                // Finish reading all events from devices identified in
previous poll().
                // This code assumes that mInputDeviceIndex is initially
0 and that the
                // revents member of pollfd is initialized to 0 when the
device is first added.
                // Since mFds[0] is used for inotify, we process regular
events starting at index 1.
                mInputFdIndex += 1;
                if (mInputFdIndex >= mFds.size()) {
                    break;
                }
                const struct pollfd& pfd = mFds[mInputFdIndex];
                if (pfd.revents & POLLIN) {
                    int32_t readSize = read(pfd.fd, mInputBufferData,
                            sizeof(struct input_event) *
INPUT_BUFFER_SIZE);
                    if (readSize < 0) {
                        if (errno != EAGAIN && errno != EINTR) {
```

```
                        LOGW("could not get event (errno=%d)",
errno);
                    }
                } else if ((readSize % sizeof(struct input_event)) !=
0) {
                    LOGE("could not get event (wrong size: %d)",
readSize);
                } else {
                    mInputBufferCount = size_t(readSize) /
sizeof(struct input_event);
                    mInputBufferIndex = 0;
                }
            }
        }
#if HAVE_INOTIFY
        // readNotify() will modify mFDs and mFDCount, so this must
be done after
        // processing all other events.
        if(mFds[0].revents & POLLIN) {
            readNotify(mFds[0].fd);
            mFds.editItemAt(0).revents = 0;
            continue; // report added or removed devices immediately
        }
#endif
        mInputFdIndex = 0;
        // Poll for events.  Mind the wake lock dance!
        // We hold a wake lock at all times except during poll().
This works due to some
        // subtle choreography.  When a device driver has pending
(unread) events, it acquires
        // a kernel wake lock.  However, once the last pending event
has been read, the device
        // driver will release the kernel wake lock.  To prevent the
system from going to sleep
        // when this happens, the EventHub holds onto its own user
wake lock while the client
        // is processing events.  Thus the system can only sleep if
there are no events
        // pending or currently being processed.
        release_wake_lock(WAKE_LOCK_ID);
        int pollResult = poll(mFds.editArray(), mFds.size(), -1);
        acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_ID);
        if (pollResult <= 0) {
            if (errno != EINTR) {
                LOGW("poll failed (errno=%d)\n", errno);
                usleep(100000);
            }
        }
    }
}
```

7.2 InputDispatcher.dispatchOnce()

```
void InputDispatcher::dispatchOnce() {

  nsecs_t nextWakeupTime = LONG_LONG_MAX;
  { // acquire lock
    AutoMutex _l(mLock);
```

```
        mDispatcherIsAliveCondition.broadcast();
        // Run a dispatch loop if there are no pending commands.
        // The dispatch loop might enqueue commands to run afterwards.
        if (!haveCommandsLocked()) {
            dispatchOnceInnerLocked(&nextWakeupTime);
        }
        // Run all pending commands if there are any.
        // If any commands were run then force the next poll to wake up immediately.
        if (runCommandsLockedInterruptible()) {
            nextWakeupTime = LONG_LONG_MIN;
        }
    } // release lock
    // Wait for callback or timeout or wake.  (make sure we round up, not down)
    nsecs_t currentTime = now();
    int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);
    mLooper->pollOnce(timeoutMillis);
}
```

7.3 InputReader::loopOnce()

```
void InputReader::loopOnce() {

    int32_t oldGeneration;
    int32_t timeoutMillis;
    bool inputDevicesChanged = false;
    Vector<InputDeviceInfo> inputDevices;
    { // acquire lock
        AutoMutex _l(mLock);
        oldGeneration = mGeneration;
        timeoutMillis = -1;
        uint32_t changes = mConfigurationChangesToRefresh;
        if (changes) {
            mConfigurationChangesToRefresh = 0;
            timeoutMillis = 0;
            refreshConfigurationLocked(changes);
        } else if (mNextTimeout != LLONG_MAX) {
            nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
            timeoutMillis = toMillisecondTimeoutDelay(now, mNextTimeout);
        }
    } // release lock
    size_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT_BUFFER_SIZE);
    { // acquire lock
        AutoMutex _l(mLock);
        mReaderIsAliveCondition.broadcast();
        if (count) {
            processEventsLocked(mEventBuffer, count);
        }
        if (mNextTimeout != LLONG_MAX) {
            nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
            if (now >= mNextTimeout) {
#if DEBUG_RAW_EVENTS
                ALOGD("Timeout expired, latency=%0.3fms", (now - mNextTimeout) * 0.000001f);
#endif
                mNextTimeout = LLONG_MAX;
                timeoutExpiredLocked(now);
            }
        }
        if (oldGeneration != mGeneration) {
            inputDevicesChanged = true;
            getInputDevicesLocked(inputDevices);
        }
    } // release lock
    // Send out a message that the describes the changed input devices.
    if (inputDevicesChanged) {
        mPolicy->notifyInputDevicesChanged(inputDevices);
    }
    // Flush queued events out to the listener.
    // This must happen outside of the lock because the listener could potentially call
    // back into the InputReader's methods, such as getScanCodeState, or become blocked
    // on another thread similarly waiting to acquire the InputReader lock thereby
    // resulting in a deadlock.  This situation is actually quite plausible because the
    // listener is actually the input dispatcher, which calls into the window manager,
    // which occasionally calls into the input reader.
    mQueuedListener->flush();
}
```

7.4 View.onTouchEvent(MotionEvent event

```
public Boolean onTouchEvent(MotionEvent event) {

        final int viewFlags = mViewFlags;

        if ((viewFlags & ENABLED_MASK) == DISABLED) {
            // A disabled view that is clickable still consumes the touch
            // events, it just doesn't respond to them.
            return (((viewFlags & CLICKABLE) == CLICKABLE ||
                    (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));
        }

        if (mTouchDelegate != null) {
            if (mTouchDelegate.onTouchEvent(event)) {
                return true;
            }
        }

        if (((viewFlags & CLICKABLE) == CLICKABLE ||
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
            switch (event.getAction()) {
                case MotionEvent.ACTION_UP:
                    if ((mPrivateFlags & PRESSED) != 0) {
                        // take focus if we don't have it already and we should in
                        // touch mode.
                        boolean focusTaken = false;
                        if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
                            focusTaken = requestFocus();
                        }

                        if (!mHasPerformedLongPress) {
                            // This is a tap, so remove the longpress check
                            if (mPendingCheckForLongPress != null) {
                                removeCallbacks(mPendingCheckForLongPress);
                            }

                            // Only perform take click actions if we were in the pressed state
                            if (!focusTaken) {
                                performClick();
                            }
                        }

                        if (mUnsetPressedState == null) {
                            mUnsetPressedState = new UnsetPressedState();
                        }

                        if (!post(mUnsetPressedState)) {
                            // If the post failed, unpress right now
                            mUnsetPressedState.run();
                        }
                    }
                    break;

                case MotionEvent.ACTION_DOWN:
                    mPrivateFlags |= PRESSED;
                    refreshDrawableState();
                    if ((mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) {
                        postCheckForLongClick();
                    }
                    break;

                case MotionEvent.ACTION_CANCEL:
                    mPrivateFlags &= ~PRESSED;
                    refreshDrawableState();
                    break;

                case MotionEvent.ACTION_MOVE:
                    final int x = (int) event.getX();
                    final int y = (int) event.getY();

                    // Be lenient about moving outside of buttons
                    int slop = ViewConfiguration.get(mContext).getScaledTouchSlop();
                    if ((x < 0 - slop) || (x >= getWidth() + slop) ||
                            (y < 0 - slop) || (y >= getHeight() + slop)) {
                        // Outside button
                        if ((mPrivateFlags & PRESSED) != 0) {
                            // Remove any future long press checks
                            if (mPendingCheckForLongPress != null) {
                                removeCallbacks(mPendingCheckForLongPress);
                            }

                            // Need to switch from pressed to not pressed
                            mPrivateFlags &= ~PRESSED;
                            refreshDrawableState();
                        }
                    } else {
                        // Inside button
                        if ((mPrivateFlags & PRESSED) == 0) {
                            // Need to switch from not pressed to pressed
                            mPrivateFlags |= PRESSED;
                            refreshDrawableState();
                        }
                    }
                    break;
            }
            return true;
        }
```

```
        return false;
    }
```

7.5 Table of Figure