Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Markus Kousa

Design, implementation and evaluation of a low-cost, high accuracy feedback latency measurement system

Master's Thesis
Espoo, November 30, 2017

Supervisor: Professor Antti Ylä-Jääski, Aalto University
Advisor: Teemu Kämäräinen M.Sc. Tech.

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Markus Kousa

Title:
Design, implementation and evaluation of a low-cost, high accuracy feedback latency measurement system

Date: November 30, 2017 **Pages:** 64

Major: Computer Science **Code:** SCI3042

Supervisor: Professor Antti Ylä-Jääski

Advisor: Teemu Kämäräinen M.Sc. Tech.

A touchscreen is a commonly used medium for the interaction between a user and a device. Response to user's action is often indicated visually on the screen after a certain delay. This interface latency is inherent in any computer system. Studies indicate that the latency has a major contribution on how users perceive the interaction with the device. While modern commercial touchscreen devices manifest latencies ranging between 50 ms and 200 ms, research indicates that the user performance for tapping tasks deteriorates at considerably lower levels and users are able to discern the latency as low as 3 ms.

In this Thesis we present a novel solution for Android operated mobile devices to expose factors behind the feedback latency of a tap event. We start by reviewing the main components of the Android operating system. Next we describe the internal system elements which partake in the interaction between the user's touch input event and its corresponding visual presentation on the screen of the device. Propelled by the obtained information, we implement an affordable, fully automated system that is capable of collecting both temporal and environmental data.

The constructed measurement system provided revealing results. We discovered that most of the feedback latency on a mobile device is accumulated by the internal components which are involved in presenting the visual feedback to the user. We also identified two main user action patterns which impose a huge effect upon system's responsiveness. Firstly, the location of touch is reflected in the amount of feedback latency. Secondly, the interval between two consecutive touch events might cause even unexpected results. Our study demonstrated that the latency can vary a lot between different devices by ranging from no effect on one device to a five-fold difference on another device.

The study concludes that, despite the feedback latency is affected by multiple factors, the latency can be measured very precisely with the system that can be built even by an average Joe.

Keywords: Touchscreen, latency, Android, Teensy, Arduino

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Markus Kousa		
Työn nimi:	Vasteajan mittausjärjestelmän suunnittelu, toteutus ja testaus		
Päiväys:	30. marraskuuta 2017	**Sivumäärä:**	64
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Antti Ylä-Jääski		
Ohjaaja:	Diplomi-insinööri Teemu Kämäräinen		

Kosketusnäyttö on yleisesti käytetty kanava käyttäjän ja laitteen välisessä vuorovaikutuksessa. Järjestelmän palaute käyttäjän antamaan syötteeseen esitetään usein visuaalisesti laitteen näytöllä. Vasteen tuottamisessa syntyy kuitenkin jonkin verran viivettä eli latenssia. Tutkimusten mukaan viiveellä on suuri vaikutus käyttäjäkokemukseen. Nykyisten kosketuslaitteiden latenssi vaihtelee yleensä 50 ja 200 millisekunnin välillä. Kosketuspohjaisten tapahtumien suorittamisen on todettu heikentyvät jo huomattavasti pienemmän viiveen johdosta ja jopa alle kolme millisekuntia kestävä viive on vielä havaittavissa.

Tässä diplomityössä esitetään Android-pohjaisille mobiililaitteille luotu edullinen järjestelmä, jonka avulla pystytään mittaamaan käyttäjän näytölle luoman kosketuksen ja sitä vastaavan järjestelmän antaman visuaalisen palautteen välistä viivettä. Työssä esitetellään ensin Android-käyttöjärjestelmän komponentit, jotka osallistuvat tämän tapahtumaketjun suorittamiseksi vaadittaviin toimintoihin. Tietojen pohjalta luodaan järjestelmä, jolla voidaan kerätä automaattisesti dataa viiveen eri syntykohdista ja sen ympäristöön littyvistä seikoista. Datan avulla pystytään aiempaa paremmin arvioimaan viiveen syntyyn vaikuttavia tekijöitä. Saatua tietoa voidaan hyödyntää yleisesti viiveen hallitsemiseen tähtääviin toimenpiteisiin ja siten lopulta käyttäjäkokemuksen parantamiseen.

Järjestelmällä mitatuista tuloksista selviää, että suurin osa tapahtumaketjun latenssista syntyy käyttäjälle esitettävän visuaalisen palautteen vaatimiin toimenpiteisiin. Lisäksi työ tuo esille kaksi käyttäjän syötteen antamiseen liittyvää toimintatapaa, joilla on suuri vaikutus latenssiin. Kosketuksen sijainti ruudulla ja kahden peräkkäisen kosketuksen välinen aika vaikuttavat vasteaikaan. Latenssi ei aina muodostu suoraviivaisesti ja se voi ilmentää jopa yllättäviä piirteitä eri laitteiden välillä: toimintatapa yhdessä laitteessa ei vaikuta tulokseen, mutta saattaa toisessa laitteessa näkyä moninkertaisena erona.

Vaikka latenssin syntyyn vaikuttaa monta eri tekijää, sitä voidaan onneksi mitata erittäin tarkasti järjestelmällä, jonka jopa Matti Meikäläinen pystyy rakentamaan.

Asiasanat:	Kosketusnäyttö, vasteaika, Android, Teensy, Arduino
Kieli:	Englanti

Acknowledgements

It took over 15 years to finalize my studies in Aalto University. Now this project has finally been completed!

I would like to thank my supervisor Antti Ylä-Jääski and my instructor Teemu Kämäräinen for giving me the chance to work on such an interesting topic and especially for their support and guidance during the process of writing this thesis.

Also I would like to express my deepest gratitude to the members of my family, especially to my beloved spouse Riikka for keeping me sane and motivated during the journey.

Espoo, November 30, 2017

Markus Kousa

Abbreviations and Acronyms

ADC	Analog-to-Digital Converter
ADB	Android Debug Bridge
AFE	Analog Front-End
AOSP	Android Open Source Project
API	Application Programming Interface
ART	Android RunTime
Ashmem	Anonymous Shared Memory
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
DSP	Digital Signal Processor
DVM	Dalvik Virtual Machine
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
HID	Human Interface Device
HWC	Hardware Composer
I^2C	Inter-Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter-Process Communication
IRQ	Interrupt Request
ITO	Indium Tin Oxide
JNI	Java Native Interface
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
NDK	Native Development Kit
OEM	Original Equipment Manufacturer
OLED	Organic Light Emitting Diode
OTG	On-The-Go
PIN	Positive Intrinsic Negative
RTT	Round-Trip-Time
SD	Standard Deviation

SPI	Serial Peripheral Interface
TIA	TransImpedance Amplifier
UI	User Interface
USB	Universal Serial Bus
VSYNC	Vertical synchronization

Contents

Abbreviations and Acronyms **5**

1 Introduction **9**
1.1 Problem statement 10
1.2 Objectives 10
1.3 Structure 11

2 Background **12**
2.1 Perception of end-to-end latency 12
2.2 Related work 13

3 Environment **16**
3.1 Android Architecture 16
3.2 From touch to display 19
3.2.1 Touch panel 19
3.2.1.1 Touch sensor 20
3.2.1.2 Touch Controller 23
3.2.2 Input Subsystem 24
3.2.2.1 Linux input pipeline 24
3.2.2.2 Android input framework 26
3.2.3 Application 27
3.2.4 Graphics framework 28
3.2.5 Display 29

4 Implementation **30**
4.1 Theory of operation 31
4.2 Hardware implementation 32
4.2.1 Microcontroller 32
4.2.2 Touch generation 32
4.2.3 Feedback detection 34
4.3 Application 36

4.3.1 Data communication 36
4.3.2 User interface 37
4.3.3 End-to-end latency measurement 38
4.3.3.1 Time synchronisation 38
4.3.3.2 Data collection 40
4.3.3.3 Touch generation 41
4.3.3.4 onTouch and frame callback 42
4.3.3.5 Feedback detection 43
4.3.3.6 Environmental data 44
4.3.3.7 Data assurance 45

5 Evaluation and results **47**
5.1 Validation 47
5.2 Accuracy and testing considerations 48
5.3 Results 49
5.3.1 Frame processing 49
5.3.2 Detection location 50
5.3.3 Light intensity 50
5.3.4 Touch location 51
5.3.5 Inter-touch interval 53
5.3.6 CPU usage 53

6 Conclusions **57**

A Results: Inter-touch interval **64**

Chapter 1

Introduction

A touchscreen is a display capable of recognizing a touch to its surface area. It is currently most widely used in mobile devices. Users can interact with mobile applications in touchscreen devices by performing various gestures on the screen. The major interaction method to date is a single tap: A finger is touched on a specific interface object briefly (*tap down*) and removed (*tap up*) to select or activate the element. This research focuses on the tap down part of the touch event.

Touch responsiveness is subject to some degree of latency which is inherent in all computer systems. Since mobile devices are basically computers, the processing introduces delay into the interaction. The end-to-end touch latency describes the delay between the user's action on the screen and the corresponding feedback on the screen provided by the system. It is a cumulative effect of the individual latencies along the end-to-end path. This latency originates from various elements pertaining to the input device, the operating system, user application and the output device.

Long latency can render the user experience useless. Studies on touchscreen devices indicate that a delay of 580 ms holds the upmost barrier after which the user experience is considered unacceptable [1]. Latency has a strong negative effect on user's performance in task completion already at the level of 75 ms [33]. However, users can perceive latency even at levels under 3 ms [40]. Current commercial touch devices exhibit latencies estimated in the range of 50 ms to 200 ms [40] which is far from the level of perception the users are capable of. To establish an understanding of how to control and improve the negative impact caused by the latency, a procedure to measure the latency is needed.

Various methods to address the latency have been studied. Until the recent years the use of an external high-speed camera has been the most common procedure to measure the end-to-end latency. However, accuracy of

the method depends on the frame rate of the camera. Due to the manual effort, post-processing the data frame by frame is time-consuming and subject to errors. High precision industrial systems are often unreachable, not least because of the amount of capital needed for the investment. Low-cost solutions capable of extracting latency data have been introduced very recently. Some approaches require active user-interaction throughout the measurement process which comes at the expense of increased user effort and achievable measurement accuracy.

The ability to split the end-to-end latency into smaller parts is vital for establishing an understanding of the elements contributing to latency. Information about the origins of latency is needed for the grounds in assessing the potential effect on user experience. Conceptually this idea is not unique and some research and implementations on the subject have already been established. However, to the best of our knowledge, none of the currently proposed solutions provide an unattended mechanism that allows measurement of the end-to-end latency in finer detail combined with collection of environmental data. Thus, this thesis mostly focuses on developing a system capable of measuring the end-to-end latency of a tap event on Android operated mobile devices.

1.1 Problem statement

In this work we aim to answer the following inter-related research questions:

- Which factors contribute to touch latency on a mobile device?
- How these factors can be measured?

1.2 Objectives

Our first goal is to create a low-cost system that is capable of collecting data on the end-to-end latency of a *tap down* event on Android operated mobile devices. The system should be able to collect latency data for individual elements along the end-to-end path and provide also the combined end-to-end latency result. In addition extraction of environmental data such as the current CPU utilization for each measurement is required. After initial setup the system should not need to interact with a human-operator and should reach total accuracy of under 1 ms.

With the aid of the constructed measurement system our second goal is to study which factors contribute to the system latency of a touch event.

1.3 Structure

The rest of this Master's thesis is structured as follows. Chapter 2 presents the background and related work. Chapter 3 provides an overview of the environment in which the measurement system is operated. Chapter 4 describes the developed implementation. Chapter 5 discusses about the evaluation of the built system and provides measurement results. Chapter 6 concludes the thesis by revisiting the main contributions.

Chapter 2

Background

In this chapter we provide a review of related work focusing on two fields. In the first section we examine studies on human perception which demonstrate the value in reducing the latency. Human factors and performance related to the end-to-end latency are also discussed. Next we introduce past efforts done by other researches in the field of measuring the end-to-end latency.

2.1 Perception of end-to-end latency

End-to-end latency can be defined as the elapsed time between a user initiating an activity and the system presenting a response [49]. It has been studied at least since the 1960s when Miller [37] described the threshold levels of human attention to computer responsiveness in the three orders of magnitude. He stated that a response time of 100 ms is vieved as instantaneous. Response times under 1 second are fast enough for users to percieve that they are interacting freely with the information. User's attention is completely lost for response times over 10 seconds. Anderson et al. [1] studied the amount of acceptable level of latency when performing common tasks with touchscreen devices. They reported that delays exceeding 580 ms were considered unacceptable for most of the users. MacKenzie et al. [33] measured the effects of latency on user's performance in motor-sensory tasks on interactive systems. They reported a strong degradation of user's performance in movement time and error rate even at levels as low as 75 ms. Studies on user's response to latency conducted by Jota et al. [24] indicate a correlation between reduced latency and improved performance. They observed that even though the minimum threshold level at which a user is able to discriminate the difference between different touch latencies is 20 ms, most users are not able to notice improvements in latency under 40 ms. Results by Deber

et al. [12] indicate similar results providing the detectable threshold levels for tapping and dragging as 69 ms and 11 ms, respectively. Ng et al. [40] discovered that the latency on dragging events is still perceivable at levels under 3 ms when comparing against a 1 ms referent.

Human perception of latency is influenced by multiple factors. Deber et al. [12] observed that the latency of a system's response to direct-touch input (such as a touchscreen) is easier to detect than the latency in response to indirect input (for example a mouse). A reason, as discussed by Ng at al., is that the perception of latency in indirect touch relies in the comparison of motion-to-visual perception which is considerably slower than using only the visual sensing modality [40]. With direct input users can observe the finger and it's graphical response simultaneously thus establishing a unified comparison point to perceive the visual difference easier than with indirect touch [12]. Sight plays an essential role also in performance. Jota et al. noticed that the size of a target affects on the performance. Under high latency small targets are harder to hit and users have to rely more on the feedback to ensure accuracy [24]. Ng et al. [41] made an observation that the perception of latency may be influenced also by the level of cognitive demands and the attention required to complete the task.

Latency perception for tapping and dragging mechanics has been characterized by Deber et al. Their results indicate that the trailing effect on the screen caused by the increased latency while dragging provide more visible manifestation of latency than the momentary difference between a tap and it's graphical response [12]. As the threshold level for the perception of latency is lower for dragging events, the desire to hide the latency from the user is frequently regarded as a catalyst to favor tap-based interaction primitives on interactive applications [24].

The importance of low system response has been widely acknowledged. As outlined by previous studies, latency should be lower than the current commercial values of 50-200 ms as reported by Ng et al.[40]. Considering the proposed threshold levels we target to achieve the accuracy of under 1 ms with our measurement tool.

2.2 Related work

Various different methods to measure the end-to-end latency have been proposed. Studies [26][40] have utilized an external high-speed camera to capture the time between the physical user action on the input device and the corresponding response on the screen. A vast amount of manual work is required to obtain results. Setting up the system and counting the frames one by

one to extract the latency requires a lot of time. Precision of the method is constrained by the speed of the camera. Also proper lightning conditions and a careful camera placement are needed to ensure proper measurements as outlined by Casiez et al. [7].

Cattan et al. [9] created a method that is based on prediction of finger movement and the ability of the operator to perceive a mismatch between the finger and the lagging response. This method requires the operator to move the finger at a constant speed in straight lines on the screen. Despite not needing an external device, an active user-operator is needed to accomplish the measurement task.

Solutions utilizing software and hardware components have emerged recently. All of them provide means to simulate a touch electronically and use a photosensor to detect the system's visual response. Introduced by Beyer et al.[4] and followed by Grau [19], microcontroller based setups were introduced to aid with collection of end-to-end latency data.

Instead of using a coin to inject a touch event on the screen (like [4][19]), the approach developed by Deber et al [11] created a touch generation module using a piece of brass contact attached to a mechanical relay. The reported accuracy of the tool is 1 ms. The approach generates only one measure to express the end-to-end latency (also like [4][19]) and therefore prevents inspecting the latency in finer detail.

The WALT [18] provided by Google is capable of measuring latency for various elements including tap, drag, screen draw, audio and MIDI. The overall end-to-end latency value is not provided because the approach uses different setups for testing the input and output latency. A tap is generated by a human-operator with a special stylus that is equipped with an accelerometer. Contact on the surface of a display considered when a shock above 3G is sensed. The output latency is monitored with a photodiode. The tool is capable of extracting the latency between various measurement points in the operating system level. Google has open-sourced the code which has inspired us to complement our approach with similar data collection points. The tool is not able to characterize the origins of the latency by means of environmental data, requires constant manual input and does not allow to measure the end-to-end latency in a single operation.

The method presented recently by Casiez et al [8] uses a vibration sensor attached to user's finger to simulate a touch. The approach is reported to allow a collection of intermediate events (such as the reception of the HID reports) via external toolkit libraries. Collection of data from different locations on the screen is made flexible by separate sensor modules. A human-operator is required to operate the system thus not allowing induction of touch signals in a precise, automated manner.

While the reviewed methods provide latency measures, they either require high amount of human interaction or lack measures to establish a connection between the latency and its contributing factors. We believe that there is a need for an affordable, fully automated solution which can help us to understand the main factors behind the feeback latency. With the aid of additional information our aim is to have a better control on the latency and to improve the user experience.

Chapter 3

Environment

This chapter provides an introduction of the components involved in the input touch event processing in the Android operating system. First we start by introducing Android and then delve deeper into the input event processing flow on Android.

The end-to-end touch event flow in Android is as follows. A touch on the screen of a device is sensed by the underlying touchpanel hardware. It signals the Linux kernel to transform the resulted device specific messages into generic input events. Android decodes and dispatches the input events to the focused user application window. The application requests the operating system to present a visual feedback to the user through the display of the device. More detailed information is provided in the following sections.

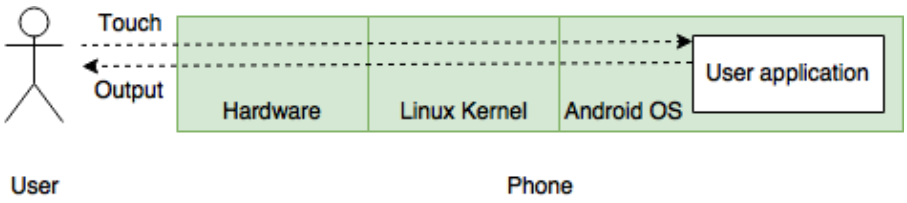

Figure 3.1: Interaction between user and application.

3.1 Android Architecture

Android is an open-source mobile operating system developed by Google. It is built on top of the Linux kernel and is designed primarily for resource constrained embedded platforms such as mobile smartphones and tablet computers. The core Android platform, known as the *Android Open Source*

Project (AOSP), contains the Android source code which is freely available for everyone [16]. Each version of the Android platform is identified by an Application Programming Interface (API) level which identifies the unique revision of the offered framework API [15]. That helps developers to manage application compatibility across different Android versions. Information outlined in this thesis is mainly based on the version 7 (Nougat). Android operating system is structured into different layers as depicted in Figure 3.2.

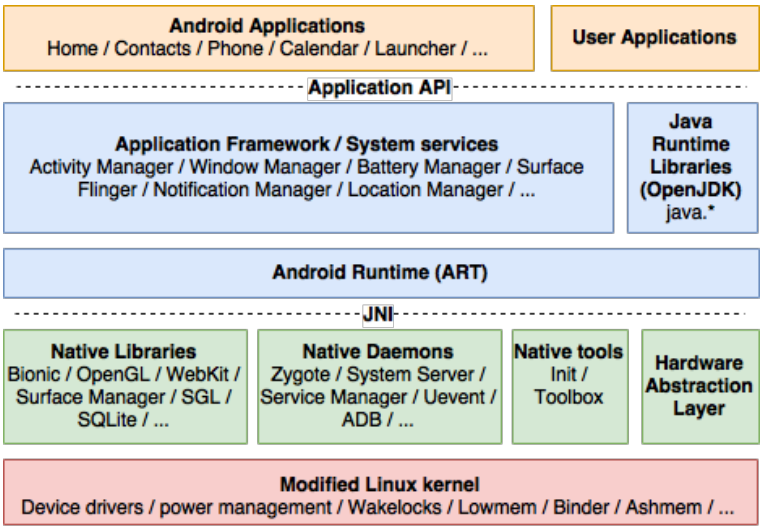

Figure 3.2: The Android architecture (modified from [56])

Applications are computer programs designed to help users to perform an activity. Android provides a set of core applications (for example Contacts, Phone, Calendar). Applications are developed mainly in the Java programming language. The Android platform is exposed to applications by the *Java framework APIs* which offer a set of components and services for building and running applications. [15]

Each Java application is executed in a restricted environment to ensure high control over the actions performed by the application. Resource constrained environments (such as mobile devices) introduce high demand for minimal memory footprint, low battery usage and effective usage of resources. Google has developed a specialized virtual machine for Android because the Java Virtual Machine (JVM) is not an optimal solution for embedded systems. *Dalvik Virtual Machine* (DVM) was the default runtime environment in Android until the version 5.0 (API level 21) after it was replaced by *Android RunTime* (ART) [15].

Android contains C/C++ based native libraries which are required by

many components built from native code. Some of the functionality provided by these libraries is exposed through the Java framework APIs. Code written in C/C++ can be integrated into native libraries with the *Native Development Kit* (NDK) and accessed from the Java code through the *Java Native Interface* (JNI) framework. [15]

In modern operating systems (like Android) the system memory is usually divided into two distinct regions: the *kernel-space* and the *user-space*. The core operating system components are executed in the kernel-space with full access to the whole environment. User processes are executed in a restricted user-space with no direct access to the underlying hardware or kernel memory. [51] The user-space environment of the Android platform covers all components on top of the Linux kernel-space. Android has introduced the *Hardware Abstraction Layer* (HAL) to allow user-space applications to communicate with kernel device drivers. A HAL provides a standard interface between the Android framework and the hardware-specific software allowing Android to be agnostic about lower-level driver implementations. [15][56]

Linux kernel is the heart of the Android architecture and exists at the very bottom of the Android software stack. Linux kernel manages the computer resources such as memory, processes, file system, network, security and power. It also provides device drivers as a communication layer between hardware and software components. Android is based on Linux kernel version 2.6. Mobile devices are typically battery-driven and have limited resources such as memory and processing power. To overcome these limitations Android has introduced several modifications to the Linux kernel. The Android Power Manager component provides aggressive power management to preserve the battery state. It instructs the kernel to go to sleep as soon and as often as possible. *Wakelocks* prevent Android devices from entering into power saving mode during critical tasks. The *Low-Memory Killer* prevents the system from running out of memory by eliminating processes of the hosting components that are not high priority and haven't been used for a long time. *Binder* is an integral part of Android. It handles inter-process communication (IPC) thus allowing applications to interact with each other and Android system components. *Ashmem* (Anonymous Shared Memory) is another IPC mechanism that uses shared memory regions to share data between processes. The alarm mechanism in Android is capable of waking the system even during the power save mode. Log events are stored into circular memory buffers instead of files thus enabling rapid logging even through the read-only filesystem types. [56]

3.2 From touch to display

Components involved in transforming a physical touch into a visual feedback on the screen of an input device can be categorized into three major parts: the touch panel, the system software and the display as illustrated in Figure 3.3. The touchpanel identifies several characteristics of the physical touch and exposes the resulted signals for the operating system. The Android framework acts as the orchestrator between the components. Visual response is actualised by the display system.

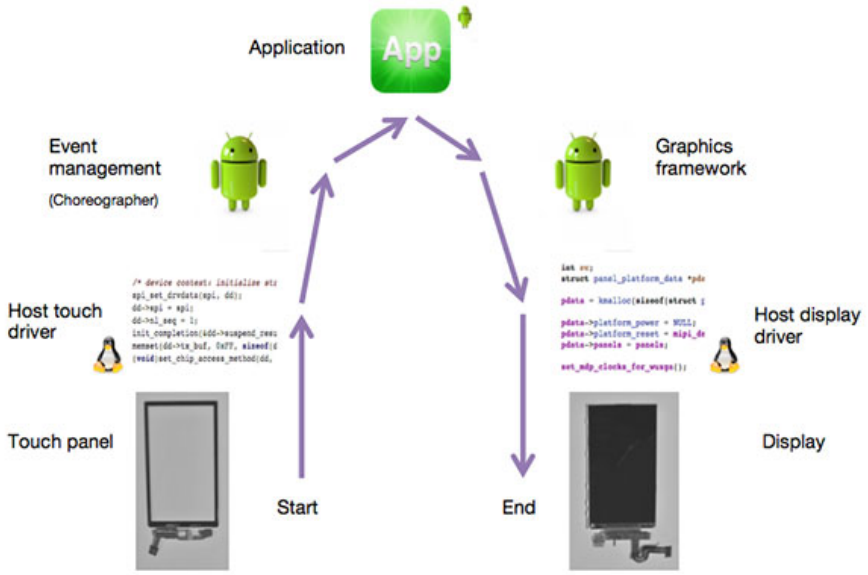

Figure 3.3: The touch ecosystem in Android. [45]

3.2.1 Touch panel

Most users interact with the phone through its touchscreen. A touchscreen is often perceived as an interactive display that is able to detect and respond to user input. From a technical perspective display and touch detection mechanisms are separate subsystems. The term touchpanel used in this thesis describes the electronic subsystem that detects and processes user touch input into a computer readable format. The term touchscreen however is used to cover the combination of a touchpanel and a display. A touch is sensed by a touchpanel which typically consists of three main components:

a touch sensor, a controller and a driver interface to interact with the host system.

3.2.1.1 Touch sensor

A touch sensor is a transparent touch sensitive surface which enables the measurement of user touch inputs. The sensor resides between a protective top layer and the display component. There are various touch sensing technologies on the market, each utilizing a different method to detect touch input. The most common touch sensing technologies in today's smartphones exploit the *resistive* and *capacitive* methods. Operation of the resistive touchpanel is based on voltage difference at the location of touch. [30]

Unlike for the resistive touch sensor, no pressure force is needed for the capacitive sensor to detect touch. It is sensitive to changes in electrostatic capacitance and exploit electronical properties of a human body. Capacitance is formed across electrode plates. [30] Capacitance is the ability of a component to store an electrical charge. Energy can be stored by separating conductive plates with an insulating material. The amount of capacitance is affected by the surface area of the plates as well as the distance and the dielectric material used between the plates. [38] Capacitance (C) is a measure of the charge (Q) stored in a capacitor at a given voltage (V) formulated as:

$$C = \frac{Q}{V} \tag{3.1}$$

The generalized equation of capacitance in a parallel-plate capacitor is formed as:

$$C = \varepsilon_r \varepsilon_0 \frac{A}{d} \tag{3.2}$$

where

- C is the capacitance (farad, F)
- ε_r is the relative static permittivity (dielectric constant) of the material between the plates
- ε_0 is the electric constant (permittivity of free space) ($\approx 8.854 \times 10^{-12}$ $\mathrm{F}m^{-1}$)
- A is the overlapping surface area of the plates
- d is the distance between the plates

Projected capacitive technology has become the most widespread capacitive touch sensing method used on smartphones. [30]. There are multiple different constructions available and various materials to form a projected capacitive sensor. Typically the sensor is constructed of a dielectric layer coated with conductive electrode lines on each side. The electrodes may also be etched on the same layer using a dielectric material to separate the overlapping electrodes. The electrodes are often made of highly transparent and conductive materials such as Indium Tin Oxide. The horizontal electrodes, usually on the top side, are linked to the Y-axis and are called the sensing electrodes. The transmitting electrodes on the other side contribute to the X-axis in the perpendicular (vertical) direction. [30] The electrode pattern geometries are an important factor in the overall resolution and touch sensitivity of the sensor [31][42]. A common approach to interleave the electrode layers and provide an optimal surface area and higher touch sensitivity is to use a single-layer diamond shaped electrode grid pattern as shown in Figure 3.4.

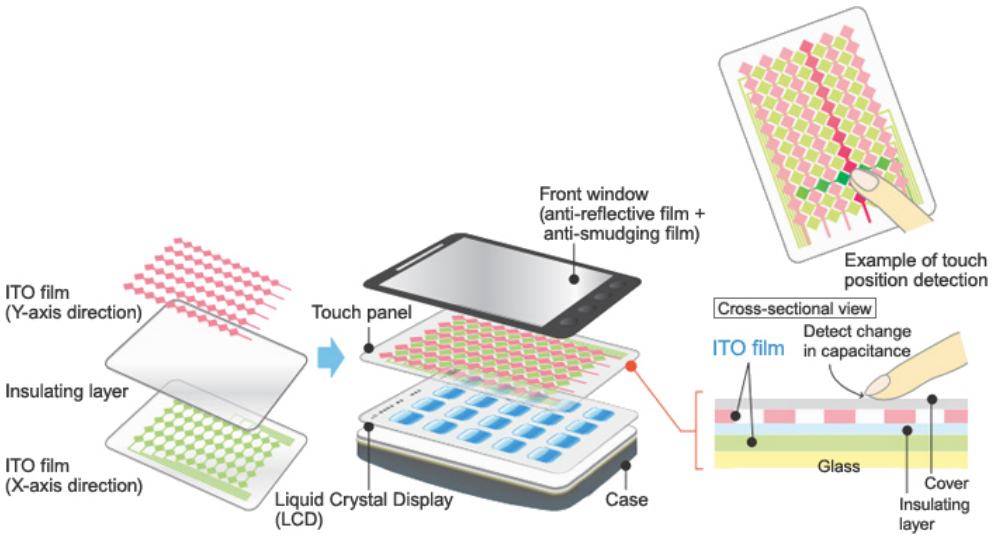

Figure 3.4: Capacitive touch sensor electrode layout. [14]

Touch detection in projected capacitive technologies is based on measuring the capacitance at each addressable electrode in the panel. A finger approaching an electrode causes a disturbance in the electromagnetic field projected above the electrode altering the capacitance on the electrode. The change in capacitance is sensed by the electronics and transformed into touch data. There are two types of capacitive sensing systems, self-capacitance and mutual capacitance [30]. *Self-capacitance* is the ratio of electrostatic charge

to a voltage potential and is exhibited by any object that can be electrically charged. *Mutual capacitance* (also known as leakage and parasitic capacitance) is the proportional factor by which a charge on a conductor at different electronic potential is induced on another conductor in close proximity as in response to the voltage potential difference. [39] The key difference between the two methods is how the electrodes are measured. The self-capacitance method is based on measuring the capacitance of a *single* electrode, one at a time. When a finger approaches an electrode, the self-capacitive load on the electrode increases due to the additional human-body capacitance with respect to ground (Figure 3.5 (a)). However, measurement of the capacitance level in the mutual capacitance method is made between a *pair* of electrodes. A finger near an intersection of electrodes causes some of the mutual capacitance between the row and column electrodes to couple with the finger thus reducing the capacitance at the intersection (Figure 3.5 (b)). [30]

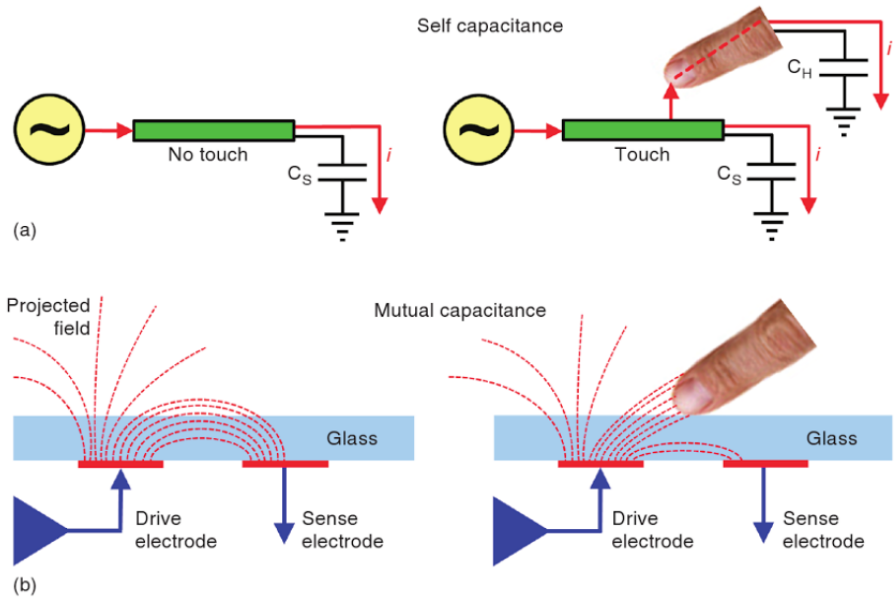

Figure 3.5: Self-capacitance and mutual capacitance touch sensing methods.[30]

The self-capacitance method is not ideal for a multi-touch functionality because of the inability to provide unambiguous touch coordinates if the screen is touched with two or more fingers that are diagonally separated. This problem known as *ghosting* is demonstrated in Figure 3.6. The more commonly utilized mutual capacitance method allows an unlimited number

of unambiguous touches, produces higher resolution and is less sensitive to electromagnetic interference [3].

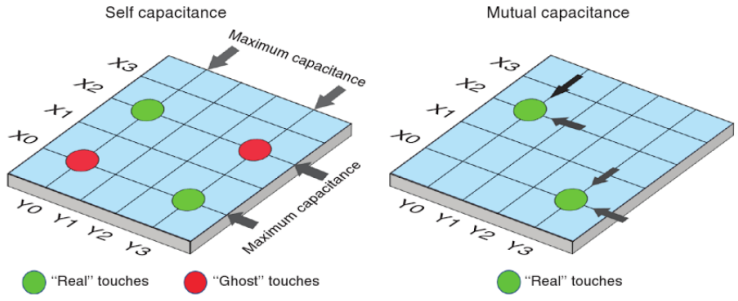

Figure 3.6: Touch coordinate detection.[30]

Many manufacturers combine the two methods by extracting multi-touch and location data with the mutual capacitive method and sense proximity using a self-capacitive solution [35].

3.2.1.2 Touch Controller

A touch controller is an application-specific integrated circuit (ASIC) designed for driving the touch sensor electrodes, measuring sensor node capacitance changes and exchanging touch data with the host computer. Figure 3.7 illustrates the basic components in a mutual-capacitance based touchpanel controller.

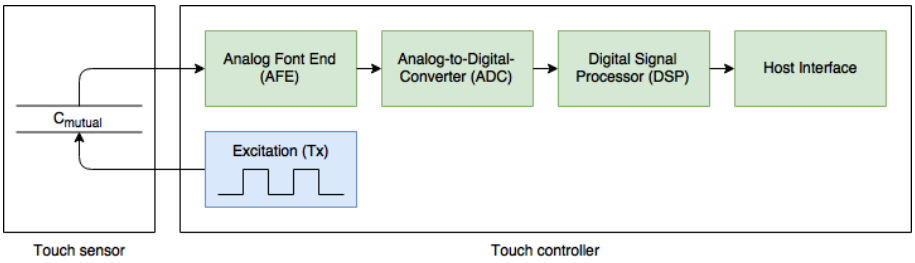

Figure 3.7: Mutual capacitance touchpanel controller.[30]

Typically a touch controller scans the touch panel one row at a time, starting from the top of the panel. The sensor transmits an excitation pulse alternately to each X electrode. Capacitance at the intersection of the excited X electrode and each Y electrode is measured by an analog front-end

(AFE). A touch is reflected as an increased capacitance level around the touch location. An analog-to-digital converter (ADC) transforms the analog values to a digital format. A digital signal processor (DSP) uses complex algorithms such as interpolation [42] to convert the array of digital capacitance values into touch specific data such as coordinate positions and touch strength values. [30]

The report rate at which processed touch events are published by the device firmware depend on several aspects. A large sensor needs usually more time to scan because more electrodes are required to provide a high touch detection resolution. The touch controller has the task to suppress noise injected by the components around the touch panel. The use of filters improves signal-to-noise ratio at the cost of increased latency. Also the number of fingers used simultaneously on the screen affect on the rate.[45] A typical report rate is in the range of 60 Hz and 140 Hz. [40][45][47]

A touch controller might enter into a sleep mode when the touch panel is not used for some time. Reduced scan frequency helps to save power but increases tap latency usually by 50 ms to 200 ms [45].

3.2.2 Input Subsystem

3.2.2.1 Linux input pipeline

The open-source Linux kernel provisions the foundation upon which the Android OS is built. Linux provides a well-maintained platform to access the under-lying peripherals. Devices are usually accessed through special hardware interfaces (such as serial ports, SPI, I^2C and USB), which are protected and managed by the kernel. The Linux kernel input subsystem is an abstraction layer which interacts between the input hardware and user space applications by exposing the user input to user space in a device-independent way through a set of various interfaces. The input subsystem, standardized in Linux kernel version 2.6, is composed of input device driver, input core and input event handler layers (Figure 3.8) [21].

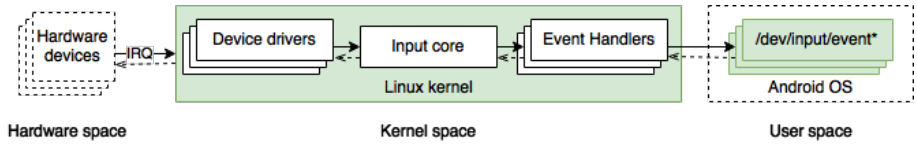

Figure 3.8: Linux kernel input subsystem

A hardware device issues an interrupt request (IRQ) to the host device CPU when data are ready to be processed by the kernel [36]. The CPU responds to the interrupts by calling the kernel which will invoke a callback to the respective device driver to claim the interrupt. The *input device drivers* act between the hardware and Linux kernel input core by responding to the respective interrupts and abstracting low-level hardware signals into standard input event messages. The Linux kernel provides drivers for a wide variety of peripheral input devices (especially for HID compliant devices) but for some built-in embedded devices such as touchscreens the original equipment manufacturer (OEM) must often provide custom drivers. [16] A touch driver processing a single touch pointer data on the 400kHz I^2C bus introduces a latency of a few milliseconds [45].

Data management within the input subsystem is done through input event messages [21]. Device specific signals are translated by device drivers into a standard input event format to provide a uniform event structure for components outside the kernel [16]. The input event message structure as shown in Listing 3.1 as defined in the Linux input protocol specification [53].

```
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```

Listing 3.1: Input event structure

Time defines the timestamp when the event was generated in the kernel. *Type* identifies the generic type of value (like a key press or an absolute motion) which subsets a group of codes applicable for the input event. *Code* defines the precise type of event (such as the axes being manipulated) and *value* contains the actual event data value raised by the device. Time is generated by the evdev driver ([17] *EventHub.h*) and other values are supplied by the calling input device driver. A single hardware event generates multiple input events. For example a touch event consists of separate X and Y-coordinate events and data about various touch properties. A special event type *EV_SYN* is used as a marker to tag input events occurring at the same moment into separate input data packets. Each input event contains new value of a single data item. Events are emitted only when values of event codes have changed. [29] Table 3.1 provides an example of input events generated for a single-touch hardware event.

Input drivers report input events to the kernel *Input core* to be passed to associated *event handlers* [21]. In Linux a special *device file* is used to

Time	Type	Code	Value	Description
1493395433.732284	EV_KEY	BTN_TOUCH	1	physical touch
1493395433.732284	EV_ABS	ABS_PRESSURE	42	contact pressure
1493395433.732284	EV_ABS	ABS_X	355	X-axis value
1493395433.732284	EV_ABS	ABS_Y	841	Y-axis value
1493395433.732284	EV_SYN	0	0	data end
1493395434.002979	EV_KEY	BTN_TOUCH	0	no touch
1493395434.002979	EV_ABS	ABS_PRESSURE	0	contact pressure
1493395434.002979	EV_SYN	0	0	data end

Table 3.1: Report of input events of a single-touch press and release.

allow user space applications to interact with device drivers via standard system calls (like read and write) just like processing regular files [36]. The device files are typically located under the */dev/input* folder where *evdev*, the generic input event handler interface eventually exposes input events to [29].

3.2.2.2 Android input framework

The Android input framework consists of a set of classes that transform raw input events into Android specific events and deliver them to interested targets. The main components managing incoming events in the Android InputManager system service are the EventHub, the InputReader and the InputDispatcher.

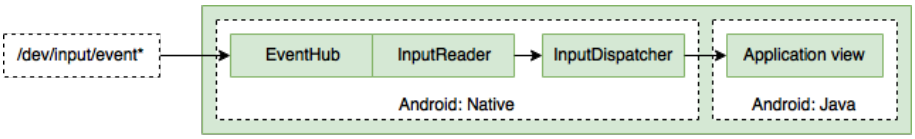

Figure 3.9: Android input event flow.

The *EventHub* component detects and reads input events from the kernel through the evdev interface associated with each input device [16]. It aggregates input events received across all known input devices on the system and keeps track of the capabilities of individual input devices such as the class and supported key codes ([17] *EventHub.h*).

The *InputReader* acquires the raw events from the EventHub and converts them based on input device specific policies into a stream of "cooked"

Android input events [16]. For example initial filtering and categorization on input events and translation of touch screen coordinates into display coordinates is performed ([17] *InputReader.h*).

The *InputDispatcher* delivers input events received from the InputReader to all active input targets registered for the events such as the currently focused window [16].

3.2.3 Application

An *activity* is a fundamental building block of an Android application. It presents a single screen with a user interface thus providing an entry point for interacting with the user. All elements in the user interface are built using *View* and *ViewGroup* objects as demonstrated in Figure 3.10. Views are responsible for drawing on the screen and handling events. The layout of the user interface is defined by the ViewGroup that is a container of child views and view groups. [15]

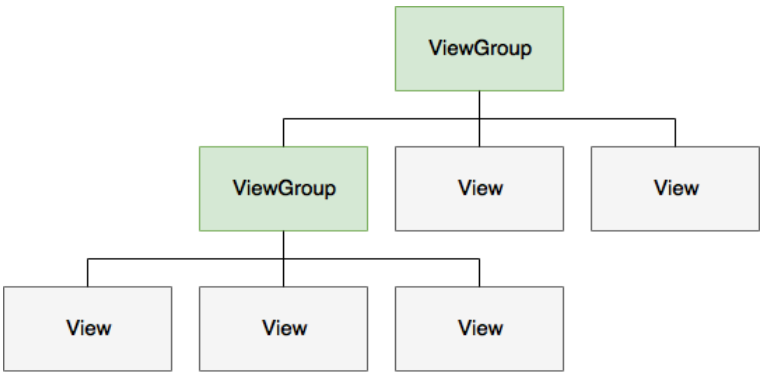

Figure 3.10: View hierarchy in Android.

Android provides several ways for an application to capture the events from a users's interaction. An *Event Listener* is an interface that is used to catch a notification when a specific event occurs. The Event Listener is associated with an *Event Handler* method which will be called by the Android framework when the respective action occurs for the registered View. For example the *onTouch()* callback method included in the *OnTouchListener* interface is called when the user performs a touch event on the associated item. [15]

Touch events are wrapped as *MotionEvent* objects. The MotionEvent class provides many methods to query the properties of the object, such as the type and a set of axis values. [15]

3.2.4 Graphics framework

Each activity is given a *Window* which is a rectangular area attached with a single view hierarchy. There can be one or more windows on the screen such as the status bar on top of the screen, the navigation bar at the bottom or side and the application user interface. Each window is associated with a *Surface* onto which the contents of the window are rendered. [16] Graphics data processing flow in Android is depicted in Figure 3.11.

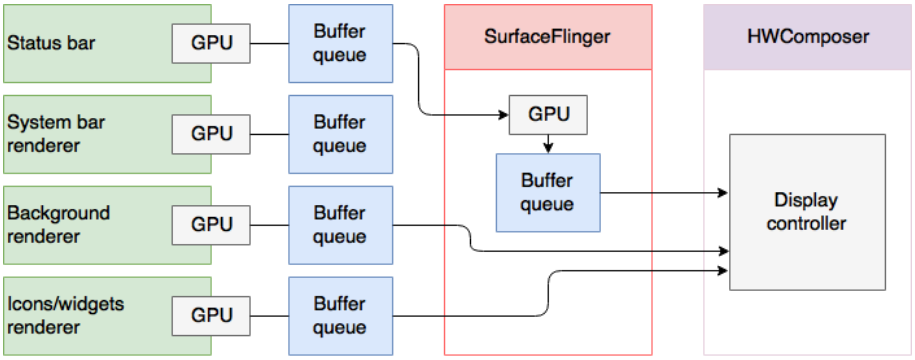

Figure 3.11: Graphics data flow.

Rendered data is populated into a buffer and returned to the buffer queue. The *BufferQueue* for a viewable Surface is typically configured with three buffers but buffers are allocated on demand to minimize memory consumption. There might be only two allocated buffers in the queue if the buffers are populated slowly enough. [16]

Composition of the surfaces is orchestrated by *SurfaceFlinger*. Frequency of processing the surfaces is limited by the refresh rate of the display. The display contents are updated only between the refresh cycles to prevent tearing the image. The display issues a periodic vertical synchronization (*VSYNC*) signal to the system stating the moment when it's safe to update the contents. Typically the signal is generated at the rate of 60 Hz. Timing operations are coordinated by the *Choreographer*. Applications always start drawing and SurfaceFlinger operations are initiated when the VSYNC signal arrives. SurfaceFlinger collects all buffers of processed data for visible layers and interacts with the *Hardware Composer HAL* (HWC) to determine the most efficient way to composite and pass the buffers to the display through the hardware driver. The HWC abstracts objects and helps to offload some work that would normally be done with OpenGL.[16]

The interval defined by the VSYNC rate sets the boundary for the application to complete the execution of its internal logic and frame drawing.

Exceeding the boundary will add the overall latency by another 16.7 ms because the application has to wait for the next VSYNC to start processing the next frame. A drop in the frame rate will result stuttering in the output visualization resulting in a bad user experience.

3.2.5 Display

At the end the image processed by an application is presented to user on a display. The most common technologies used for the displays in the current smartphones are Liquid Crystal Display (LCD) and Organic Light Emitting Diodes (OLED). [5]

Several aspects pertain to latency originating from the display side. The *response time* describes the time needed to transition a pixel from one *gray value* to another. The gray value defines the intensity of the pixel ranging between 0% (black) and 100% (white). [5] Typical *response times* of the LCD displays differ between 2 ms and 100 ms whereas the OLED displays tend to react considerably faster, usually within a few microseconds. [45]

Another matter is the internal *refresh rate* that describes the interval between the times the display redraws its contents. The refresh rate is typically about 60 Hz which translates roughtly to 16.7 ms. [5]

An internal *frame buffer* can be used on the display to store the latest frame data. The benefit of using the memory buffer is the need for the host to transmit data to the display only at times when the content of the frame has changed. The lower amount of data transfer helps to reduce power consumption. However, additional memory storage introduces another buffer to fill before the frame can be displayed yielding an additional latency of 16.7 ms. [45]

Chapter 4

Implementation

At this point we have defined the top-level end-to-end flow and the elements that comprise the flow. In this chapter we describe the implementation and related design considerations of our measurement instrument. The measurement setup is depicted in Figure 4.1.

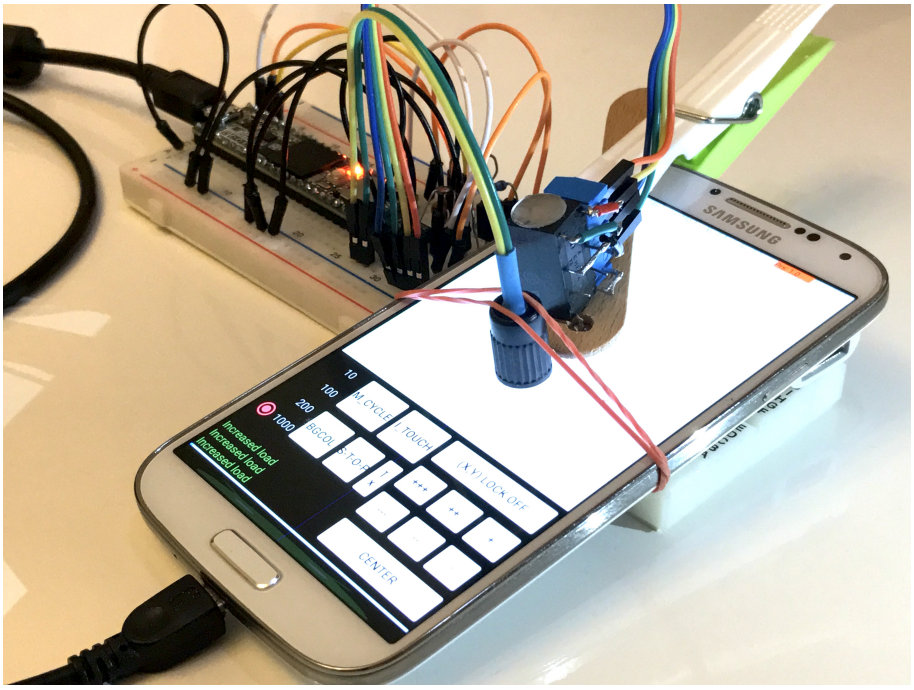

Figure 4.1: Measurement setup for end-to-end delay analysis.

4.1 Theory of operation

The goal of the implementation is to accurately measure the time consumed between a triggered touch event and its corresponding visual counterpart on the screen. The tool traces the end-to-end path and provides latencies for both the individual elements along the event flow and the total cumulative end-to-end latency value. Variability in latency is expected due to timely inconsistent factors such as the injection time of the touch signal and the constantly alternating utilization rate of the CPU. The variability can not be explained comprehensively with only a single end-to-end measure and therefore the tool collects also environmental data (such as the CPU utilization and battery state) during latency measurement.

Measurement is applicable once the application is installed on the inspected device. The application is automatically started right after the device is attached to the measurement instrument via a USB connection. The touch signalling and photosensor modules are placed on top of the device. User is given an option to choose a measurement program and select the amount of measurements performed per each measurement cycle. The end-to-end program collects latency data along the whole end-to-end path. The phone application acts as the host and interacts with the measuring instrument during the whole measurement cycle.

At the beginning of each measurement cycle the screen is analysed to obtain reference values for the black and white brightness levels. This is needed to define appropriate threshold values for the visual feedback detection. Time synchronization between the phone and the instrument is performed automatically during the measurement cycle to ensure a high control over the inherent drift in time between the devices. Data collection is started after the initialization phase has finished. At first the instrument activates the touch pad to inject a touch signal on the screen of the inspected device. Timestamp corresponding the time of touch initiation is recorded and the instrument starts monitoring the screen with the photosensor to detect a visual response. Once the operating system has processed and delivered the corresponding touch event to the application, the framework is instructed to provide a visual feedback by changing the color of the background from black to white. In addition the phone collects timestamps at intermediary collection points which are detailed later in this chapter. The instrument records timestamps when a change in screen brightness is detected and transmits the collected data to the application. At the end of each measurement the application collects environmental data such as the usage, frequency and temperature of the CPU. After each measurement the need for time syn-

chronization is verified and an arbitrary delay is added to generate timely variation between consecutive touch signals. After the measurement cycle has finished the operator is provided an option to receive the collected data wirelessly.

4.2 Hardware implementation

Creating an artificial touch signal, capturing feedback from the display and providing accurate timestamps for the captured events raise a requirement to harness external hardware components to generate the needed functionalities. In this section we describe the hardware implementation of the measurement instrument.

4.2.1 Microcontroller

Functions must be operated in a precise, controlled manner. A Teensy 3.5 development board [50] featuring a 32-bit 120 MHz ARM Cortex-M4 processor was chosen because it is very easy to program, contains an effective CPU enabling a high-resolution data sampling, provides a native USB interface and in addition is very affordable.

The USB interface in Teensy was initialized to appear as a serial device type to enable date exchange with Android using the *bulk transfer* communication type. The Analog-to-Digital converter (ADC) was configured to output data as an average of 8 samples (*analogReadAveraging*) in the 12-bit resolution (*analogReadRes*). This combination defines the required processing time needed to extract photodiode data values from an analog input (*analogRead*). The obtained duration of 17μs ($\approx$ 58 kHz) is used as a dimensioning parameter for the feedback detection module.

4.2.2 Touch generation

Various methods in creating an artificial touch signal have been proposed by other researchers. Attaching a conductive object (typically a small copper plate or a coin) on the screen hooked with a wire to a controlling unit appears to be the most common solution. Some methods involve movement, such as poking the screen with a conductive object either manually [8][18] or using a robotic arm. The moment when a moving object touches the screen surface can be detected with various sensors such as a vibration sensor [8] or an accelerometer [18]. Resolving the relation between the location of touch and it's respective latency profile is considerably easier if touch signals can be

injected in a controlled manner always exactly to the same physical location of the screen. For that reason we pursued to implement a stationary, yet a flexible solution.

As discussed in the previous chapter, the basis of capacitive touch detection is the ability to identify a change in capacitance. Altering capacitance is relatively straightforward. Since capacitance depends on the amount of charge as defined by formula 3.1, a change in capacitance can be achieved by dissipating charge via a conductive material [44].

We examined various different materials and shapes during the development process of the touch signal contact. Grau [19] used copper tape to trigger a touch signal on an iPhone. Based on our experiments with Samsung S4 even a tiny piece of copper tape itself, without an attached signalling wire, resulted in a reported touch event. Our trials with a copper coin (used in [4][19][27]) revealed a high amount of dispersion in location among the resulted touch events. Therefore the characteristics of a coin, such as the relatively large, rugged surface contact area, were considered to be non-optimal for the purpose.

We noticed big differences in touch sensitivity among different mobile devices. Sensitivity is affected by the amount of capacitance and in the context of capacitive touch detection the magnitude of change introduced by a touch is on the order of picofarads or only hundreds of femtofarads [20][42]. Managing sensitivity becomes a bigger challenge when a conductive material is attached onto the contact pad to establish a path to a control unit for signal generation. Often even a short wire alone, with the other end of the wire disconnected, was enough to trigger a touch signal. This was also remarked by Deber at al [11]. The more sensitive the touch sensor is, the more attention must be paid on the materials used. Also the length of the connection wire should be kept at the bare minimum to prevent an unintentional dissipation of charge and also to diminish electrical noise.

We obtained most promising results with a brass contact (like [12]) of 8 mm in diameter and 1 mm thick. Brass is recognized as a suitable material also in many other display testing systems [13][46]. We paid special attention on attaining a well polished contact surface to limit the distribution of injected signals into a narrow spot on the pad.

Due to aforementioned sensitivity considerations there was a need for a switching component capable of enabling and disabling the signal reliably. Multiple components such as solid-state switches, transistors and analog relays were examined. Unfortunately components providing lowest level switching-times (in the nanosecond-level) could not be utilised because they were leaking too much current and thus caused unintentional touch signals. We obtained the best results with an Omron G5V-2-H1 [43] electromechani-

cal relay. Although the sensitivity issue was tackled and the signal production was very reliable, a new issue emerged with the relay. Due to the mechanical nature of the selected component, the switching delay of the relay was close to 6ms. This time is needed to energize a coil to magnetically actuate an iron contact inside the relay [23]. The relay holds two switches which are controlled simultaneously by a single coil. This enabled us to feed one pole of the relay to the touch pad while the other pole was attached to the microcontroller allowing an accurate way to deduct the actual delay in every single measurement. The contacts of a relay have a tendency to bounce on closing [23]. That produces rapid vibrations in the signal over a short period of time. Based on our measurements an average bounce time of $\pm 59\mu$s (SD=70 μs, N=1000) was introduced, allowing us ignore the impact of the bounce time.

Our contact pad produced pretty stationary touch signals. Approximately 99%, 99.8% and 99.9% of all triggered touch signals were within ± 1, ± 2 and ± 10 pixels apart from each other, respectively. Considering the pixel density of 441 pixels per inch in our test device (Samsung S4), the width of 3 and 5 pixels yield respective signal dispersion diameters of approximately 0.17 mm and 0.29 mm. The implemented solution was considered very reliable also because no unintended touch signals occurred during the operation.

4.2.3 Feedback detection

The feedback is visualised as a change in background color on the phone display. The change in brightness can be detected with a photosensor component coupled with analog and digital signal processing as depicted in Figure 4.2.

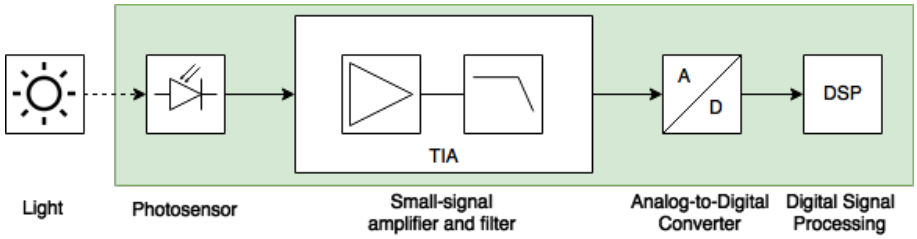

Figure 4.2: Change detection block diagram (modified from [52])

A Vishay BPW34 PIN photodiode [54] was selected to detect changes in brightness because it provides a wide spectral sensitivity, fast response time (the reported rise and fall time of 100ns) and comes in a flat packaging.

The low-level current sourced by the photodiode must be amplified and transformed into a voltage in order to enable measurement using a microcontroller. This type of conversion is usually implemented using a *Transimpedance amplifier* (TIA) circuit which typically consists of a photodiode, an amplifier and a resistor/capacitor feedback pair. An LTC1050 operational amplifier [32] was chosen due to excellent characteristics such as low input bias current, offset voltage, noise and voltage drift. Figure 4.3 presents the TIA circuit diagram used in our application.

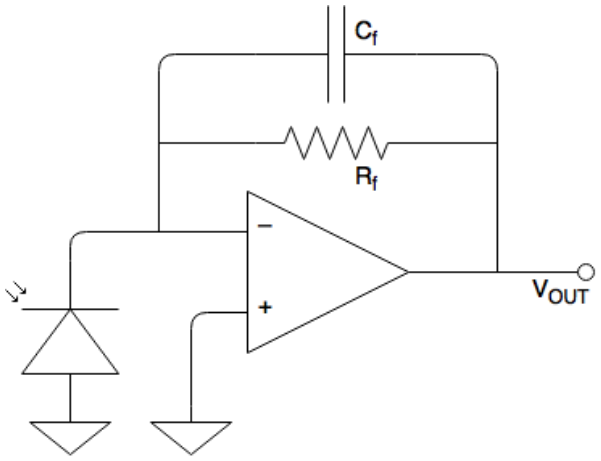

Figure 4.3: Photodiode amplifier circuit diagram.

Amplification of the photodiode current is defined by the value of the feedback resistor R_f. The maximum light brightness on a smartphone display is typically around 250-750 cd/m^2 [22][25] thus the circuit was dimensioned covering the operation range between 0 and 1000 cd/m^2. According to the photodiode datasheet [54] current I_{PDmax} produced by the photodiode at the maximum dimensioned level is 75 uA. The maximum voltage V_{OUT} fed into the microcontroller should be in the range between 0 and 3.3V. The nearest standard resistor value of 47kΩ was selected for the feedback resistor R_f on the basis of the theoretical value calculated using formula 4.1.

$$R_f(Ohm) = \frac{V_{OUTmax}}{I_{PDmax}} = \frac{3.3V}{75\mu A} = 44k\Omega \tag{4.1}$$

A *cutoff frequency* defines the point after which energy flow through the device starts to reduce thus preventing it from working as designed. The frequency depends on the used feedback resistor and capacitor values based on formula 4.2

$$f_{op}(Hz) = \frac{1}{2\pi R_f C_f} \tag{4.2}$$

The analog voltage signal produced by the circuitry is processed by the ADC of the microcontroller. Value of the feedback capacitor C_f when using the maximum operation frequency of about 58 kHz f_{base} as used in our experiments is

$$C_f(F) = \frac{1}{2\pi R_f f_{base}} = \frac{1}{2\pi 47k\Omega 58kHz} \approx 58.4pF \tag{4.3}$$

However, the capacitor value of 30 pF was selected to leave some room for further testing. The photodiode was capped to limit the ambient light from entering the sensor. The light emission of the display on measured devices was set at the maximum brighness level to achieve a stronger detection signal. A small amount of noise was picked by the sensor. That was expected because the wire used to connect the sensor to the microcontroller was not properly shielded. The effect of noise was managed programmatically to ensure that only valid readings were accepted.

4.3 Application

In this section we report the software part of implementation. We start with a discussion about the USB because the application starts automatically when a device to be inspected is connected via the USB connection. Next the main parts of the user interface are described. Finally the measurement cycles are detailed.

4.3.1 Data communication

Data communication between the phone and the measurement instrument is performed over the USB connection. The *Universal Serial Bus* (USB) is a standard for a serial communication interface. Communication is made between the host and one or more client devices. The USB is a polled bus, where the host initiates all data communication. A smartphone acts normally as the USB client device. The *On-The-Go* (OTG) is a standard that enables mobile devices to act as a host. [2] A smartphone can be equipped with the OTG support by utilizing an external OTG adapter cable to allow data communication with a microcontroller.

Android phone acts as the USB host when communicating with Teensy. USB connection establishment is initiated in the onCreate() method of the

main activity. An instance of *UsbManager* class is obtained to access the state of USB and establish connection with USB devices. A HashMap containing all currently attached USB devices is iterated to discover and retrieve the desired *UsbDevice*. *UsbInterface* of the device object containing Teensy specific device vendor ID is accessed to map separate *UsbEndpoints* for sending and receiving data. Finally an instance of *UsbDeviceConnection* is created to open the device for synchronous data transfer. The *bulkTransfer()* method is used to perform data transfer for both directions.

An intent action *ACTION_USB_DEVICE_ATTACHED* is broadcasted by Android when a USB accessory is connected to the USB bus. This intent contains the *UsbDevice* object of the connected device. Main activity of the application is mapped with an intent filter for the same USB attached action. The intent is processed in the *onResume()* method. Main activity is started automatically if the vendor ID attribute conveyed in the UsbDevice object corresponds with the vendor ID of Teensy defined in application's manifest file. Permission to communicate with the USB device is obtained if the user grants the application to handle the intent in the initial USB device connection establishment dialog with the user.

4.3.2 User interface

The user interface allows user to interact with the application. Application contains only one activity, the main activity (*MainActivity*). The user interface window (depicted in figure 4.4) consists of a *ScrollView* and two *TextView* elements, *Button* widgets and *RadioButtons*. The ScrollView element is used for providing textual feedback to user, TextView elements for user touch detection and response visualisation. The amount of measurements made in each measurement cycle is specified by the radio buttons. The time between consecutive touch events can be defined in steps of 1μs, 10μs and 100μs.

The screen was triggered at and measured from various different coordinate positions during the development process. Each UI element was therefore constructed into separate components to provide a flexible way to change the layout of UI components during our experiments. When the end-to-end latency was measured, the touch detection and response UI elements were combined to enable placing both sensor modules next to each other in the middle of the screen.

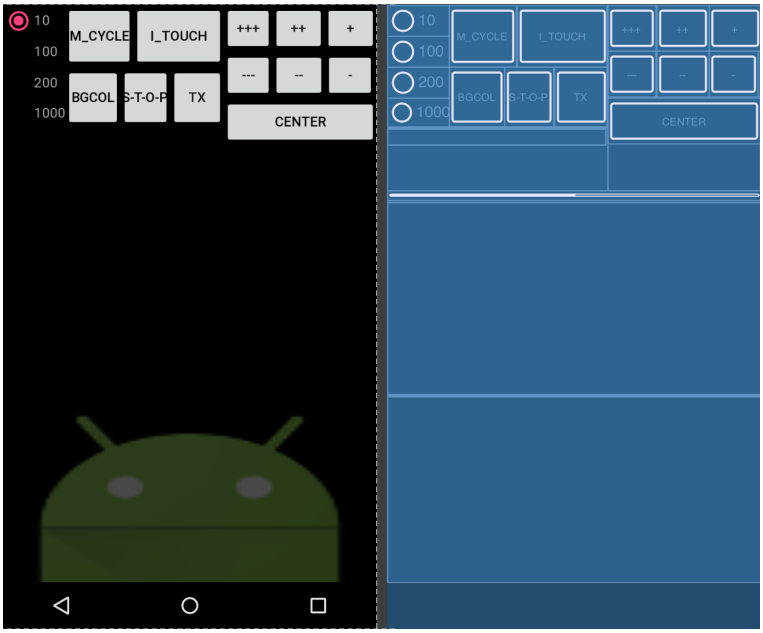

Figure 4.4: The user interface and design components.

4.3.3 End-to-end latency measurement

Measurement process starts with an analysis of screen reference values (detailed in chapters covering touch generation and feedback detection) and the clock synchronization followed by automated data collection.

The measurement program is accessed though the user interface. An *OnTouchListener* is registered on the RUN button object in the Main activity. When the button is pressed the system executes the code written in *onClick* on the main thread. A new thread is instantiated at start of the code thus releasing the main thread for other activities. Two separate threads are used to execute the measurement: the main (UI) thread is responsible for updating the user interface and handling touch event callback procedures whereas the other thread executes the rest of the activities. At first new objects are instantiated to store measurement values. The current battery level is queried from the *BatteryManager* class via an intent filter created for the *ACTION_BATTERY_CHANGED* action.

4.3.3.1 Time synchronisation

Syncronization of time is performed at the beginning of each measurement cycle. Each event is mapped with a distinct timestamp at the time of the event.

The monotonic *nanoTime* method is used in Android and *elapsedMicros* in Teensy to extract current timestamps. Timestamps originating from different sources can not be compared directly per se, so a mechanism is needed to synchronize the clocks between external sources. To our knowledge there is no robust method for providing a sub-millisecond precision in synchronizing clock signals between a microcontroller and a host node without the need to access an external network or the GPS. Our solution implements time management between the devices through the USB connection.

Time synchronization through the USB interface is based on measuring the minimum *Round-Trip-Time* (RTT) which identifies the time needed to transmit a USB packet from phone to Teensy and back. The RTT process is managed by the phone application and is performed as follows:

1. The current timestamp $phone_{start}$ is stored and immediately after that a byte of data is transmitted to Teensy. Teensy responds with it's current timestamp $teensy_{time}$.
2. The current timestamp $phone_{end}$ is stored immediately after reception of the response. The RTT is calculated ($phone_{end}$ - $phone_{start}$) and stored.
3. The RTT cycle is performed 1000 times. A random delay of 2-5 ms is applied between each transmission to prevent message accumulation in the USB buffer and therefore an evident distortion in results.

Response processing time in Teensy is subtracted from the elapsed RTT to extract an estimate of the USB data transfer delay. After the RTT cycle has finished, timestamps mapped to the smallest RTT are used for defining the reference time point, all relative to the same point in time, as follows:

- RTT = $phone_{end}$ - $phone_{start}$ - $teensy_{processing}$
- $phone_{reference} = phone_{start} + \frac{RTT}{2}$
- $teensy_{reference} = teensy_{time}$

When measuring the latency, the timestamps are converted into duration calculated from the reference times. This allows us to link the timestamps originating from different sources. The minimum RTT defines the tolerance level of accuracy when estimating the actual teensy timestamps. Received Teensy time can be is estimated as $teensy_{phone\_time} = phone_{reference} \pm \frac{RTT}{2}$. However, latency is likely asymmetric due to asymmetric nature of the USB. Based on our measurements an average minimum RTT (excluding processing time in Teensy) was 173 μs (N=500, SD=12.1 μs) yielding a typical time synchronisation difference of $\pm$87 μs between the two devices.

Most electronic devices are equipped with a hardware oscillator to create a signal with a precise resonance frequency to provide a stable clock signal. Accuracy of the oscillator is defined as ppm (parts per million) which indicates the maximum deviation of device's crystal from the nominal value. Due to variation in clock frequency, separate clock signals will drift apart over time. Our application compensates clock drift by performing time synchronization throughout the measurement cycle. A typical stability requirement for mobile phones is about 10 ppm [28]. As the frequency tolerance of the crystal used in Teensy is 18 ppm [34], the maximum clock drift of 90 μs is ensured by performing time synchronization every 5 seconds.

4.3.3.2 Data collection

We have identified six data collection points which allow us to split the end-to-end path into five sections as illustrated in Figure 4.5.

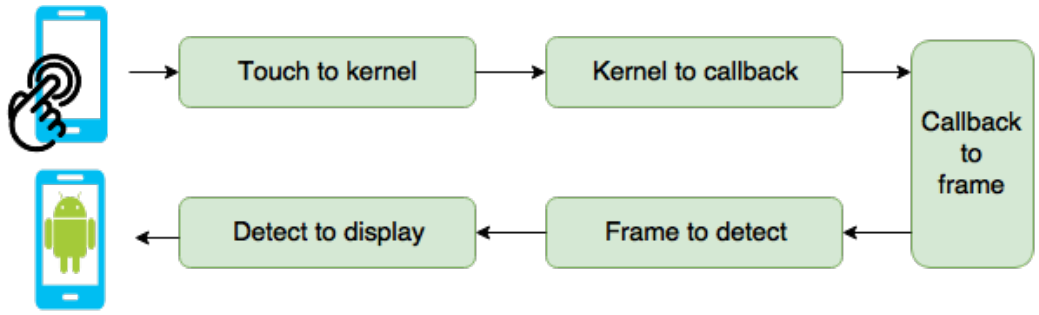

Figure 4.5: End-to-end latency split into sections in the data collection sequence.

Touch to kernel delay is the time between a touch signal is activated on the screen and the corresponding touch event is created in the kernel.

Kernel to callback defines the duration between the kernel event timestamp and when the touch event was dispatched to application via the onTouch() callback method.

Callback to frame specifies the delay between the time the touch event was dispatched to application via the onTouch() callback method and the time when rendering a display frame started. The display frame is processed as a consequence to the request to change the background color for the view.

Frame to detect states the period that is spent between the graphics framework initiated by the frame draw and when a change on the display color is detected.

Detect to display represents the duration needed to draw the frame on the display.

End-to-end latency provides the overall delay between the moment the touch signal is activated on the screen and the time the corresponding visual feedback has been drawn on the display.

The following items were recognized as potential factors to have an influence on the end-to-end latency:

- Action: Position of touch
- Action: Position of feedback detection
- Action: Time interval between consecutive touch signals
- Data: The CPU (utilization, frequency and heat)
- Data: The level of available charge in the battery
- Collect: Phone attributes (the manufacturer, the model, the OS version)

The items prefixed with "Action" are based on different user action patterns. The items marked with "Data" were collected for each measurement and their relationship with the latency was analysed. The "Collect" item states the characteristics which we believe are essential in characterizing the constant environmental factors which should be collected as a part of the measurement phase and reported along the measurement results. Tests and results for the assessed item are reported in the next chapter.

The end-to-end measurement cycle is depicted in Figure 4.6. The steps of the cycle are described in the following sections.

4.3.3.3 Touch generation

Generation of a touch signal is the first step in the measurement cycle. A digital output pin on the microcontroller is connected to the relay. Changing the pin to high state energizes the relay. However, operation of the relay is not instant and therefore needs to be monitored in order to determine the exact moment when the switch has actually closed. A change in signal is reflected simultanously in both poles of the relay. Therefore monitoring can be done with an analog input port connected to the other pole of the relay. A connection to signal ground is needed to initiate touch generation. To detect the ground signal, the input port is initiated by the *INPUT_PULLUP* mode, setting the pin signal state to high. Immediately after the relay is triggered, the analog port is monited for change. The closure of the switch has actualized once the ground signal level is identified. The current timestamp

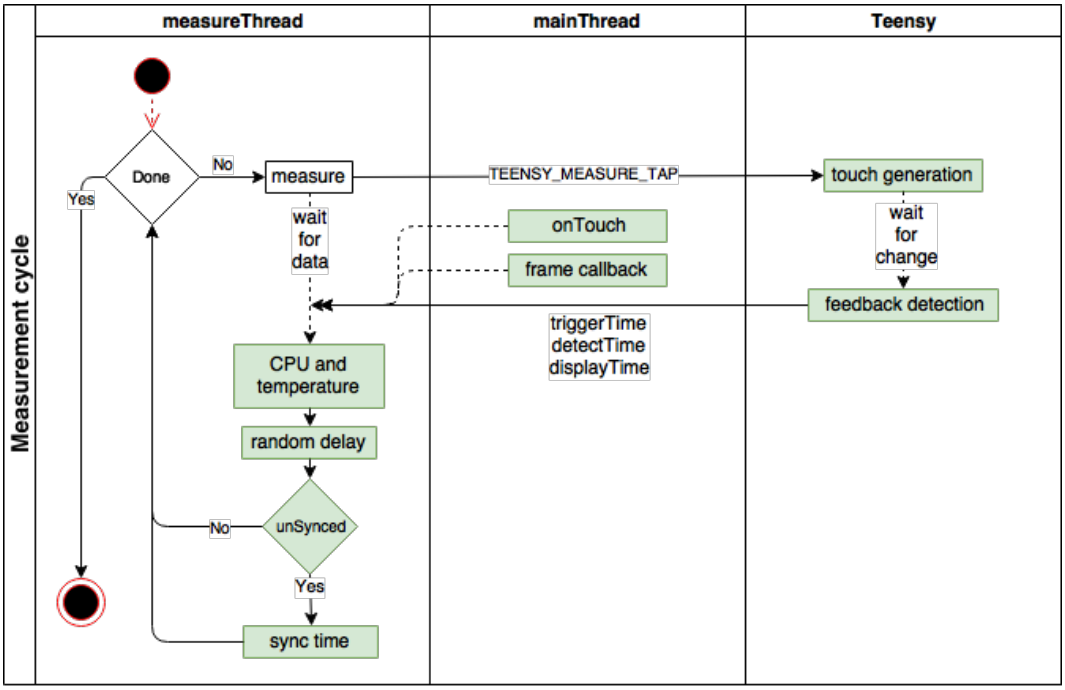

Figure 4.6: The end-to-end measurement cycle.

is identified as the *physical* touch activation time. Next the Teensy proceeds to the feedback detection phase.

4.3.3.4 onTouch and frame callback

When user touches the screen, Android delivers the resulted touch event to registered input targets. An *OnTouchListener* is registered for the touch detection view element to receive touch events. A callback to the *onTouch()* method is invoked by Android when touch events are dispatched to the registered view. Therefore all application logic related to touch event handling is implemented in the overridden onTouch() method.

At first the type of the event is obtained. If the event action code equals to *MotionEvent.ACTION_DOWN*, denoting the touch going down, the current timestamp is stored (*callback time*). Immediately after that the *setBackgroundColor* method is used to change the background color to white for the view.

The time when the next frame started being rendering (*frame time*), is obtained with a callback to the *doFrame* method.

The officially documented *MotionEvent.getEventTime()* method provides

time data only in the millisecond resolution. Therefore the hidden *MotionEvent.getEventTimeNano()* method was preferred to retrieve the kernel level event time (*kernel time*).

4.3.3.5 Feedback detection

Response to touch is presented as a change in the screen brightness level. Our application utilizes the transition from black to white but in practise a change between any levels can be used. To detect change between two states, reference values for the corresponding levels must be acquired. This acquisition is performed before a run of measurements is executed. The average value of the white level, sampled for the period of 100 ms, was used as the threshold value for the white level. However, the control for the black level required more effort.

The signal analysis conducted at the time of development revealed a small amount of noise and random irregular peaks in the signal level measured for the black state. That was probably caused by the unshielded cables used for prototyping the instrument. These issues could not be addressed at the time of the development phase, raising a special need for the detection algorithm.

With the constraints in mind we implemented a novel approach for detecting change in the black level. At the initialization phase the black screen is first sampled for the period of 300 ms to extract the mean, the maximum and the standard deviation of the sample. The standard deviation (SD) provides a measure to quantify the amount of variation or dispersion between values in a set of data. Only a random sample of the population is studied because it is not possible to sample every member within the entire population. The corrected sample standard deviation is calculated using the formula 4.4

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{4.4}$$

, where x_i is one sample value, $\overline{x}$ is the sample mean and N is the sample size.

The process is considered out of control if a data point exceeds the control limit [10]. The control limit for the maximum black screen value is defined as *maxValue + L*SD* where maxValue is the maximum value seen in the sample and the sigma limit L is equal to 3 as in the design of the Shewhart control chart limits [10].

Accuracy of the implemented algorithm was validated manually. We compared the difference of the timestamps between the data point reported by the algorithm and the first actual occurrence in the sample exceeding the *maxValue* value. The obtained accuracy of 0.08 ms (SD 0.03 ms, N=100) was considered adequate. No false alarms were reported during the use of the instrument.

Detection of the white level is illustrated in Figure 4.7 which demonstrates the luminance curve of the AMOLED display used in the inspected Samsung S4. For large gray transitions, the final level is obtained only in the subsequent time frames [6]. This is shown as an intermediate gray level which is visible in Figure 5.1. Timestamps of the generated touch event and the detected screen transition events are sent to phone immediately after the white level has been detected.

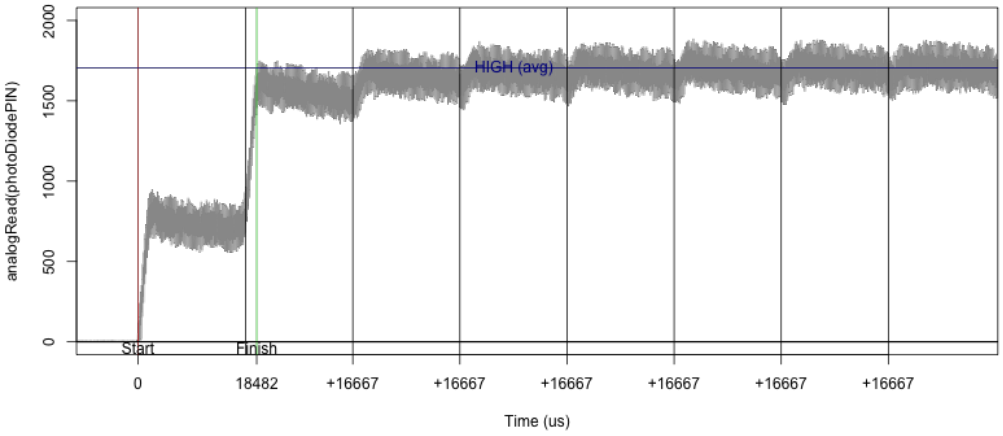

Figure 4.7: Luminance curve at the transition from black to white. Illustrated time frames are drawn since the time of detection. Sampled at the rate of 58 kHz.

4.3.3.6 Environmental data

Data for the CPU utilization is collected in each measurement. The amount of time the CPU has spent performing work is measured in clock ticks and is available in the */proc/stat* file. The monotonically increasing numbers in the file are aggregated per type of work since the system boot time and contain separate events for each CPU core. Delta between the statistics collected

at the start and end of each round denote the amount of work being made during the collected samples. The *CPU load* is a percentage of the CPU's capacity over time which is used by the CPU for processing instructions other than waiting for input/output operations (*iowait*) or entering the low-power (*idle*) mode.

The CPU load level alone does not provide enough information on the full utilization of system resources. Depending on the workload the operating frequency of CPUs may be scaled down. The CPU usage in /proc/stat is reported with respect to the scaled operating frequency. The CPU load value must be normalised with respect to full CPU potential by taking into account the amount of time spent in each frequency state. Normalized CPU load is calculated as [48]

$$CPUload\frac{weightedFrequency}{maximumFrequency} \tag{4.5}$$

The amount of time consumed per frequency is extracted from *time_in_state* file. Statistics from the file are collected along with corresponding CPU load data and delta times are calculated. Relative usage percent of each frequency is calculated by dividing frequency specific delta time over the total consumed delta time. Frequency values are multiplied by corresponding relative usage percentage. Weighted frequency is the sum of resulting values. Maximum operating frequency of the CPU is read from *cpuinfo_max_freq*. Normalized CPU load is calculated separately for each CPU core. In case a core is offline the time_in_state file might not be available and thus corresponding normalized CPU load is reported as a value of 0.

Temperature data of all available thermal zones is read next. Current temperature is provided by the thermal framework. Separate data files for each thermal zone are available through the sysfs under */sys/class/thermal/thermal_zone* directories. The type of thermal zone is declared in the *type* file. Current temperature in (milli)degree Celcius as reported by each thermal zone sensor can be read from the *temp* file.

4.3.3.7 Data assurance

Based on our findings the location of touch impacts on the *touch to kernel* delay. To ensure a better control over the location of induced signals on the screen, only samples within ±3 pixels apart from the inspected center point were accepted at the measurement phase.

The interval between consecutive touch events on the screen has an impact on the latency. The effect was discovered for the intervals below 660

ms. A random delay between each measurement was applied to ensure the distribution of touch signals over the interval range of 150 ms and 1000 ms.

At the end of each measurement round the state of time synchronization is validated. The current time is compared against the last synchronization time and another synchronization step is executed if the maximum allowed unsynchronized time limit is exceeded.

Chapter 5

Evaluation and results

In this chapter we describe how our implementation was evaluated and discuss the obtained results based on the data collected with the developed system.

5.1 Validation

Operation accuracy of the measurement tool was validated with a high-speed camera Sony NEX-FS700E. A run of 10 consecutive end-to-end measurements was performed while recording the operation with the camera at the speed of 800 frames per second (equals to 1.25 ms per frame). Figure 5.1 illustrates the most essential phases of the measurement cycle. The led in the microcontroller was illuminated between the moment the touch signal was initiated (step 1) and the time when the threshold value for the white level was reached (step 4). The frames displaying the illuminated led were counted for each measurement and the resulting frame time was compared against the data collected by our measuring instrument.

Accuracy of the validation was limited by the frame rate of the video equipment. The duration of a single frame is significantly longer than the time needed by the microcontroller to detect change and to control the led. Due to this limitation the validation involves an uncertainty of two camera frames (2.5 ms). All measured samples and their corresponding frame counterparts were within one frame duration of each other (N=10, mean 0.41 ms, SD 0.28 ms). Correlation between the variables in the two data sets was calculated with *the Pearson's correlation coefficient*. The obtained R value of 0.9995 indicates almost a perfect positive correlation between the data sets. The results gave us confidence of the accuracy of our device and spurred us to start exploiting the full potential of the tool by collecting data.

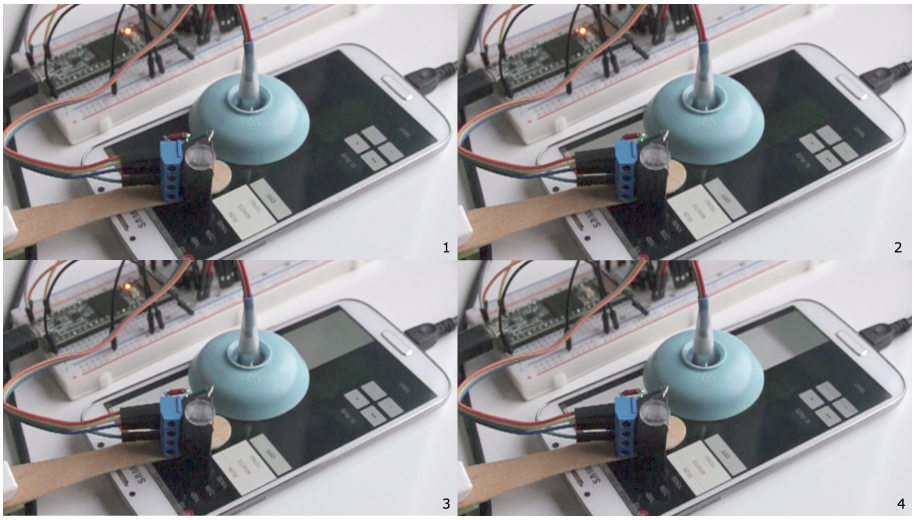

Figure 5.1: Measuring the end-to-end latency. Step 1: The state when a touch was triggered. Step 2: A change in the black level occurs. Step 3: A change in the white level occurs. Step 4: The detection cycle has just completed.

5.2 Accuracy and testing considerations

Accuracy of the developed system is composed mainly by the elements interfacing between the application and the external measurement apparatus. The USB communication introduces an average latency of ±87 μs (SD=12.1 μs). The maximum delay of 90 μs is adduced in the cycles of 5 secods as the consequence of the task in compensating the potential clock drift between the phone and the microcontroller. The average bounce time in the relay used for triggering a touch signal was measured as ±59μs (SD=70 μs). The method used for detecting the change on the initial black level on the screen was identified to achieve the precision of ±80 μs (SD=30 μs). Accuracy on detecting the white level is evaluated based on the validation made with the high-speed camera. Bearing in mind the uncertainty factor of 2.5 ms as introduced by the limited frame rate of the camera, the calculated precision of 0.41 ms (SD=0.28 ms) was reached. In the light of aforementioned measurement results, we can conclude that the initial requirement of the overall accuracy of under one millisecond was achieved.

The time synchronisation phase is executed frequently during all measurements to ensure precise timing between the connected devices. However, the synchronization procedure introduces varying delay between individual mea-

surements. The delay sourced by the USB connection latency was controlled automatically between measurements by limiting the maximum round-trip-time to 200 μs ($\pm$100 μs). As discussed next in this chapter, the latency of touch events is heavily influenced by the time interval between consecutive touch signals. To ensure control on valid results, the first set of data after each time synchronisation phase was therefore discarded in the tests regarding the touch location and the inter-touch interval.

Measured devices were Samsung S4 (Android 5.0.1) and Huawei Nexus 6P (Android 8.0.0).

5.3 Results

In this section we discuss the results and make conclusions about the main aspects affecting the end-to-end touch latency. Our measurement results are shown in Table 5.1.

	Samsung S4		**Nexus 6P**	
	Avg	**SD**	**Avg**	**SD**
Touch to kernel (ms)	15.4	1.0	29.5	1.2
Kernel to callback (ms)	1.8	0.4	1.8	0.4
Callback to frame (ms)	8.8	1.5	8.7	0.9
Frame to detection (ms)	40.8	0.8	39.2	1.5
Screen update (ms)	17.6	0	17.5	0.1
End-to-end (ms)	**84.4**	**2.0**	**96.6**	**2.2**

Table 5.1: End-to-end latency. 1000 x 10 pcs. Inter-touch interval max 200 ms.

5.3.1 Frame processing

The results indicate clearly that most of the end-to-end latency originates from frame display processes. This observation is shared also by other researchers [8][27]. The result was already anticipated because frame processing is tightly coupled with the amount of utilized frame buffers and the vertical refresh rate of the display. The use of multiple buffers helps in ensuring a smooth visualization of moving images but comes at the expense of increased delay.

5.3.2 Detection location

The position where the inspected object is located on the screen introduces a delay which is directly proportional to the vertical position of the object and the refresh rate of the display. To examine the effect on the latency, the photo detection module was placed at each corner of the screen. The physical size of the module introduces a constraint on measuring the absolute minimum and maximum coordinate locations (0,0), (x_{max},0), (0,y_{max}) and (x_{max},y_{max}). As a consequence the latency range between the measured top and bottom locations is smaller than the full potential of the actual screen resolution.

Position	**Avg**	**SD**
Top left	33.4 ms	0.2 ms
Top right	33.4 ms	0.2 ms
Bottom left	49.1 ms	0.2 ms
Bottom right	49.1 ms	0.2 ms

Table 5.2: The relation between detection location and frame processing. (Samsung S4, N=1000).

The results as shown in Table 5.2 demonstrate the operation of the display. The display draws the given data in vertical direction at the speed dictated by the refresh rate. Depending on the location of the inspected object, the delay of up to one full draw is introduced which is approximately 16.7 ms at the refresh rate of 60 Hz.

There is no set standard on which location the measurement should be performed on the screen. It is essential to position the photo detection module at the same location on the screen to obtain comparable results. We believe that the average screen draw latency is generally produced at the center of the screen and therefore all of our measurements were obtained from that location.

5.3.3 Light intensity

The difference of gray values (the lightness of color) in transition between the initial and the final colors defines how quickly the transition occurs on the display. Figure 5.2 demonstrates the difference. Transition from black (0% gray) to white (100% gray) took approximately 18.0 ms while transition from middle gray (50% gray) to white was reached in only 1.3 ms. The difference of 16.7 ms corresponds roughly to the VSYNC pulse duration.

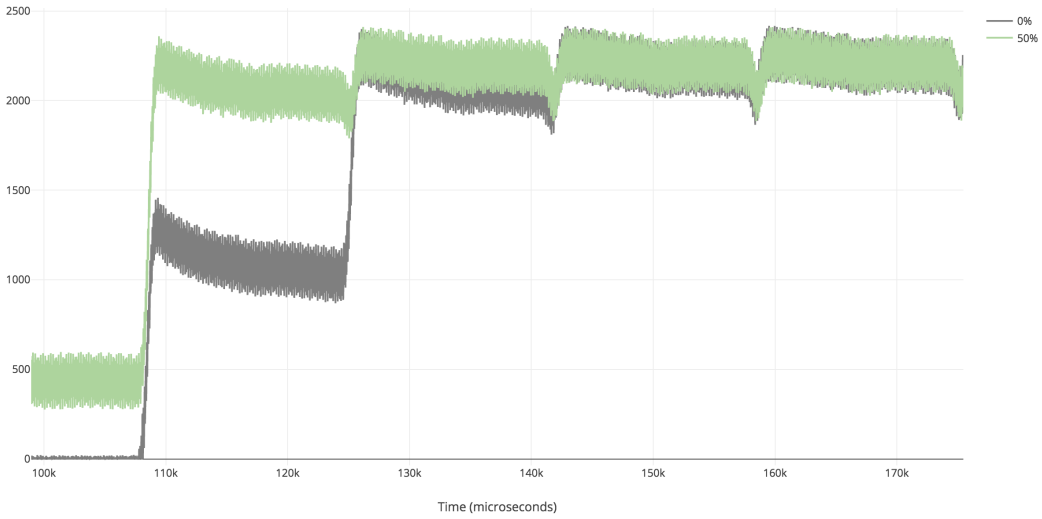

Figure 5.2: Luminance curve at different base levels of gray value in transition to white (Samsung S4, sample rate 58 kHz).

5.3.4 Touch location

Touch input processing has a major role in the overall end-to-end latency distribution and is mostly determined by the location of touch and the time between consecutive touch events. This section concentrates on the location of touch.

Induction of a touch signal initiates the end-to-end latency path. Touch signals are often fed onto the system at the center of the screen [4][11][19][27]. Discussion about the effect of touch location is however often disregarded. We experimented the impact of location by injecting touch signals at the corners of the display. 10 series of run were taken at each location. A single run consisted of 1000 samples. The interval between consecutive touch signals was kept constant at 100 ms.

Our results reveal that the latency is dependent on the touch location and is induced mainly by the touch panel. The relation was already expected because a touchpanel is typically scanned line by line starting from one side of the panel and collected samples are processed only after the entire panel has been completely scanned. While experimenting with Samsung S4 we anticipated that the screen scanning would occur in vertical direction thus resulting the smallest delay at the bottom of the screen. However, the results (as shown in Table 5.3) indicate that touch signal scanning is performed horizontally in that particular device, starting from the top left corner. As

indicated by the results, the latency decreases along the x-axis being at the minimum on the right side of the panel. In vertical direction the difference was considerably smaller.

Phone	Position (X,Y)	Average range	Mean	SD
Samsung S4	(57,61)	16.0 - 27.1 ms	21.3 ms	2.9 ms
Samsung S4	(58,1863)	15.8 - 26.9 ms	21.1 ms	3.0 ms
Samsung S4	(1020,55)	7.9 - 19.0 ms	13.0 ms	2.9 ms
Samsung S4	(1025,1873)	8.1 - 19.2 ms	13.2 ms	2.9 ms
Samsung S4	(540,960)	14.0 - 21.2 ms	17.6 ms	2.0 ms

Table 5.3: Touch-to-kernel times measured at each corner of display using the inter-touch interval of 80 ms. (Samsung S4 [1080x1920], N=1000).

The *latency jitter* describes the range between the minimum and maximum latency values. The jitter measured for Samsung S4 varied usually between the range of 10 ms to 11.1 ms which correspond roughly to the operation frequencies of 100 Hz and 90 Hz, respectively. The jitter was not affected by the location of touch or the time between two consecutive touch signals. Similar observation was made by Vu et al. [55]. They injected touch signals onto a touchpanel at the rate of 1 kHz. Their experiment revealed that 98% of the time the resulted latency jitter was less than 40 ms. The limit is probably introduced for practical reasons to accomodate the maximum possible event rate that can be generated by a human [55].

An unexpected behavior requiring future research was observed. The latency jitter of 7.2 ms, measured at the center screen coordinate on Samsung S4, is considerably smaller than the jitter for the other measured locations as shown in Table 5.3. Similar behavior was not observed with Huawei Nexus P6.

The duration of one panel scan, performed by the touch controller, can be estimated based on the difference of the latency jitter values between the farthest corners of the display. The time difference of 8.0 ms, as measured for Samsung S4, implies the touch scanning frequency of 120 Hz. However, considering an estimated average time of 1 ms [45] required for signalling operations between the touch controller and the kernel, the actual scanning frequency is probably closer to 140 Hz. Without access to the datasheet of the touch controller (Synaptics S5000B) we were not able to verify our estimated touch scan rate.

5.3.5 Inter-touch interval

Based on the literature [45] a touch controller is likely to enter into a power-saving mode when a device hasn't been actively used for a certain amount of time. This function is reflected in the touch controller as a slower operating frequency which appears as a longer touch event reporting delay.

We examined the effect of the power-save mode by feeding the touch panel with signals at different intervals. Location of the touch signal contact was carefully placed at the center of the screen. Touch signals were produced in multiple batches, each at a fixed interval. At first the range of 1 s to 5 s was examined in one second intervals. Next the range between 100 ms and 1000 ms was measured in steps of 100 ms. A run of 200 measurements was executed for each interval. Data were analysed after each run to find any deviation between the samples. Another set of measurement runs was conducted in steps of 10 ms for the ranges in the close proximity of deviating time intervals. Finally a run of 1000 measurements was performed for the intervals having distinct characteristics.

Results obtained from our test devices were not foreseen. We did not observe a difference in the latency jitter on Huawei Nexus P6 across the inspected interval range as the jitter remained at a constant level of 29.4 ms. However, we discovered peculiarities in the results reported by Samsung S4 for the intervals between 350 - 410 ms and 650 - 660 ms as depicted in Figure 5.3. The measurement results are provided in Appendix A. Interestingly the fastest average delay of 14.8 ms resulted as the response for the inter-touch interval of 400 ms. The longest results of 45.1 ms were obtained at the interval of 650 ms.

Unfortunately we could not test another copy of S4 to conclude whether the phenomenom is the result of an extraordinary design or just an indication of a defective touch sensor. However, based on our results measured for S4 at the inter-touch interval of 200 ms, the obtained end-to-end latency of 84.4 ms (SD 2.0 ms) is very similar to the result of 85 ms reported by Beyer et al. [4].

5.3.6 CPU usage

CPU utilization during the latency measurement cycle in a normal system condition was typically at around 40%. To find the relationship between latency and CPU usage, a method to simulate high CPU usage up to the level of 100% is needed. Our test device Samsung S4 (model GT-I9506) contains four processor cores. As one process can utilize only one core at a time, three additional concurrent processes are required to reach the full

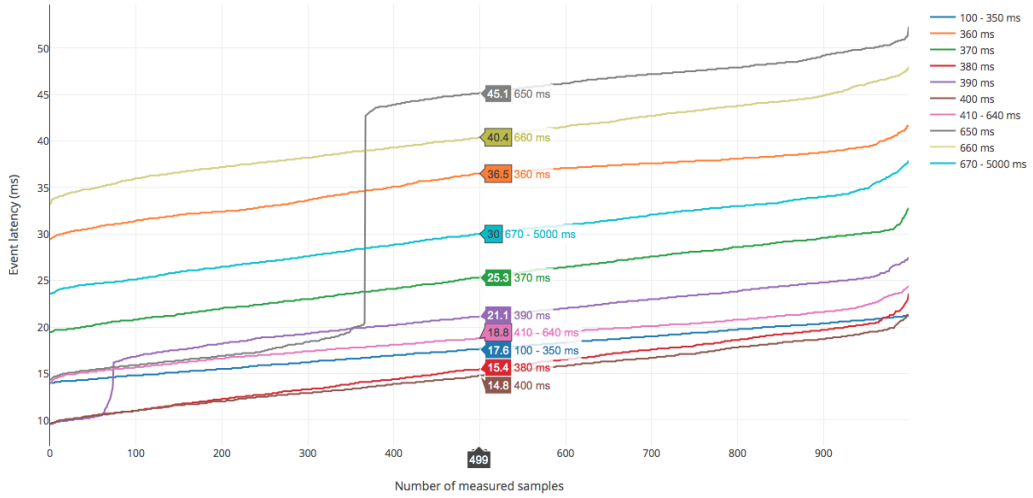

Figure 5.3: Inter-touch interval vs. *touch to kernel* latency (Samsung S4, measured at 540x960).

processing potential of the CPU.

A batch of 500 end-to-end measurements was executed. Each batch contained 10 samples. Utilization on the CPU was increased in three phases during the measurement to simulate increase in the CPU usage. The *adb tool* was used to connect wirelessly to the phone and to execute a *while loop* to increase the CPU usage. The Android Debug Bridge (adb) is a command-line tool that enables communication with the device [15].

Results shown in Table 5.4 and depicted in Figure 5.4 present a strong relationship between the CPU usage and the latency. During the first 200 measurements the average end-to-end latency remained at around 84.5 ms (SD 2.9 ms). A major increase in latency can be observed at the point when CPU utilization reached the average level of 80%. Similarly higher latency is reflected at the next CPU utilization phase when all four CPU cores were being utilized.

Measurements	**CPU %**	**Avg**	**SD**
1-76	39.2 %	84.4 ms	3.4 ms
77-199	61.6 %	84.6 ms	2.6 ms
200-279	81.5 %	91.7 ms	9.1 ms
280-500	94.5 %	105.4 ms	15.6 ms

Table 5.4: CPU utilization vs. *end-to-end* latency.

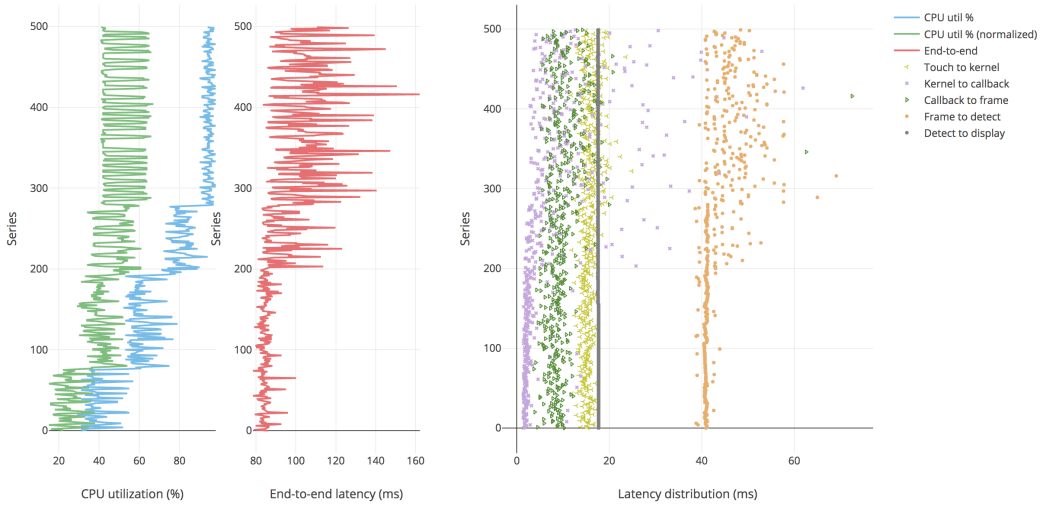

Figure 5.4: CPU % vs. *end-to-end* latency distribution (Samsung S4).

Measurement results for the extracted sections of the end-to-end latency are shown in Table 5.5. The results indicate that the higher the CPU utilization is, the more time is needed to dispatch touch events to application (*kernel to callback*). Results pinpoint that up to 50 % of the overall latency increment may fall upon the Android input framework. The touch input processing side, including touch controller and driver, was not considerably affected (*touch to kernel*). As expected, CPU usage does not seem to have any effect on the screen redrawing time (*screen update*).

	39.2%		**61.6%**		**81.5%**		**94.5%**	
	Avg	**SD**	**Avg**	**SD**	**Avg**	**SD**	**Avg**	**SD**
Touch to kernel	15.1	1.0	15.0	1.0	15.6	1.2	16.4	1.9
Kernel to callback	2.5	2.3	2.8	1.8	6.9	6.8	12.9	11.6
Callback to frame	8.2	1.7	8.3	1.8	9.1	2.2	11.4	6.2
Frame to detection	40.9	0.5	40.9	0.7	42.6	2.7	47.0	5.6
Screen update	17.7	2.5	17.7	0.1	17.6	0	17.6	0

Table 5.5: CPU utilization % vs. *end-to-end* latency distribution (ms).

The operation of dynamic frequency scaling (CPU throttling) is also demonstrated in Figure 5.4. The normalized CPU utilization, which considers also the operating *frequency* of the CPU, shows clearly that throttling is increased at both transition points of CPU utilization, starting at the

point of 81.5%. As a consequence, the normalized CPU utilization remains approximately at the same level even though CPU utilization increases. The frequency is decreased to conserve power and to produce less heat.

Chapter 6

Conclusions

Touchscreens are a commonly used medium for the interaction between a user and a device. Response to user action is often provided by the means of a visual response on the screen. The corresponding response time is subject to the inherent delay induced by the system actions performed along the end-to-end processing path in accomplishing the interaction. Studies indicate that the latency has a major contribution on how users perceive the interaction with the device. While modern commercial touchscreen devices manifest latencies ranging between 50 to 200 ms, research indicates that user performance for tapping tasks deteriorate at considerably lower levels and user's are able to discern latency as low as 3 ms. The need to reduce the delay has been widely acknowledged and several methods for characterizing the latency have been introduced. Even though current solutions may be capable of extracting the delay in the finer details, some constrains do apply. They either lack the ability to conduct the measurement process without a human-operator or are not capable of retrieving additional environmental data to help assessing the factors behind the latency.

In this Thesis we presented a novel solution for Android operated mobile devices to expose factors behind the feedback latency of a tap event. We started by reviewing the main components of the Android operating system. Next we described the internal system elements which partake in the interaction between the user's touch input event and its corresponding visual presentation on the screen of the device. Propelled by the obtained information, we implemented an affordable, fully automated system that is capable of collecting both temporal and environmental data. The system was utilized to examine which factors contribute to touch latency on a mobile device.

We found that most of the end-to-end latency is composed by the processes involved in presenting the visual feedback to user. This output latency is mainly dictated by frame rendering and buffering elements which can con-

tribute over half of the overall latency. The amount of delay imposed by drawing the final image on the screen is dependent on the refresh rate of the display and the location where the item is observed at. The incurred delay was at its smallest at the top of the display and increased along the vertical direction. Also the amount of light intensity in transition between one background color to another might affect on latency.

Touch input processing is another major source of latency. We identified two main contributing user action patterns. Latency induced by the touch-panel is directly proportional to the location of touch in conjunction with the operation rate of the touch controller. Also the interval between two consecutive touch events was demonstrated to impose unexpected results ranging from having no effect on one device to resulting in a five-fold difference in the latency on another device.

Our study pinpointed that high CPU usage has most effect on event dispatching. Delay of that latency section was shown to increase by five-fold and eventually contribute up to 50% of the overall latency increment.

We hope that the proposed system and the results provided in this Thesis help to develop even better interactive applications and create more positive user experience.

Bibliography

[1] Anderson, G., Doherty, R., and Ganapathy, S. User perception of touch screen latency. In *Design, User Experience, and Usability.* (2011), A. Marcus, Ed.

[2] Axelson, J. *USB Complete: The Developer's Guide, Fifth Edition.* Lakeview Research LLC, 2015.

[3] Barrett, G., and Omote, R. Projected-capacitive touch technology. *Information Display*, 3 (2010).

[4] Beyer, J. A method for feedback delay measurement using a low-cost arduino microcontroller. Master's thesis, 2015.

[5] Bhowmik, A. K. Mobile displays: Technology and applications. John Wiley and Sons, 07 2008.

[6] Boher, P. New methods for optical characterization of oled displays. Electronic Display Conference, 25-26 February 2015, Nuremberg.

[7] Casiez, G., Conversy, S., Falce, M., Huot, S., and Roussel, N. Looking through the eye of the mouse: A simple method for measuring end-to-end latency using an optical mouse. UIST '15 Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, 2015.

[8] Casiez, G., Pietrzak, T., Marchal, D., Poulmane, S., Falce, M., and Roussel, N. Characterizing latency in touch and button-equipped interactive systems. Proceedings of UIST'17, the 30th ACM Symposium on User Interface Software and Technology., 2017.

[9] Cattan, E., Rochet-Capellan, A., and Berard, F. A predictive approach for an end-to-end touch-latency measurement. ITS 2015, 2015.

[10] Coleman, S., Greenfield, T., Stewardson, D., and Montgomery, D. C. *Statistical Practice in Business and Industry*. John Wiley & Sons, 2008.

[11] Deber, J., Araujo, B., Jota, R., Forlines, C., Leigh, D., Sanders, S., and Wigdor, D. Hammer time! a low-cost, high precision, high accuracy tool to measure the latency of touchscreen devices. In *chi4good, CHI 2016, San Jose, CA, USA* (2016), Tactual Labs / University of Toronto.

[12] Deber, J., Jota, R., Forlines, C., and Wigdor, D. How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch. CHI 2015, 2015.

[13] Epson. Scara http://global.epson.com/products/robots/products/scara/g.html, 2017.

[14] Geomatec. The structure of a projective touch panel http://geomatec-sputtering.com/tech/ito/electricity/, 2017.

[15] Google. Android developer. https://developer.android.com, 2017.

[16] Google. Android open source project. https://source.android.com, 2017.

[17] Google. Git on android. https://android.googlesource.com, 2017.

[18] Google. Walt https://github.com/google/walt, 04 2017.

[19] Grau, P. Development and evaluation of an arduino-based feedback delay measurement instrument, 2016.

[20] Gu, H., and Sterzik., C. Capacitive touch hardware design guide. http://www.ti.com/lit/an/slaa576a/slaa576a.pdf. Texas Instruments, 2015.

[21] Hards, B. The linux usb input subsystem http://www.linuxjournal.com/article/6396, 04 2017.

[22] Ho, J., and Chester, B. Understanding brightness in amoled and lcd displays. https://www.anandtech.com/show/8795/understanding-brightness-in-amoled-and-lcd-displays, 2014.

[23] Hughes, J. M. *Practical Electronics: Components and Techniques*. O'Reilly Media, Inc., 2015.

[24] Jota, R., Ng, A., Dietz, P., and Wigdor, D. How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks. CHI2013, 2013.

[25] K., P. Let there be light - here are 2015's smartphones with the brightest displays. https://www.phonearena.com/news/let-there-be-light—here-are-2015s-smartphones-with-the-brightest-displays_id75128.

[26] Kaaresoja, T. *Latency guidelines for touchscreen virtual button feedback. PhD thesis.* PhD thesis, University of Glasgow, 2016.

[27] Kämäräinen, T., Siekkinen, M., Ylä-Jääski, A., Zhang, W., and Hui, P. Dissecting the end-to-end latency of interactive mobile video applications. ACM, 2 2017.

[28] Kaul, A. B. *Microelectronics to Nanoelectronics: Materials, Devices & Manufacturability.* CRC Press, 2012.

[29] Kernel.org. https://www.kernel.org/doc/documentation/input, 2010.

[30] Kriss, M. *Handbook of Digital Imaging.* John Wiley and Sons, 2015.

[31] Lee, J., Cole, M. T., Lai, J. C. S., and Nathan, A. An analysis of electrode patterns in capacitive touch screen panels. *Journal of display technology, Vol. 10, no. 5* (5 2014).

[32] Linear. Ltc 1050 http://cds.linear.com/docs/en/datasheet/1050fb.pdf, 2017.

[33] MacKenzie, I. S., and Ware, C. Lag as a determinant of human performance in interactive systems. Interchi'93, 1993.

[34] Manitou48. Crystals https://github.com/manitou48/crystals/blob/master/crystals.txt, 2017.

[35] Matthew Trend, P. C. Touch sensor including mutual capacitance electrodes and self-capacitance electrodes. Google Patents (US20130154996 A1), 2011.

[36] Mauerer, W. *Professional Linux Kernel Architecture.* Wiley Publishing, Inc., 2008.

[37] Miller, R. Response time in man-computer conversational transactions. In *Proceeding AFIPS '68 (Fall, part I) Proceedings of the December 9-11, 1968, fall joint computer conference, part I* (1968), pp. 267–277.

[38] Morrison, R. *Digital Circuit Boards: Mach 1 GHz*. John Wiley and Sons, 2012.

[39] Morrison, R. *Grounding and Shielding: Circuits and Interference*, 6 ed. John Wiley and Sons, 2016.

[40] Ng, A. Designing for low-latency direct-touch input. UIST '12, 10 2012.

[41] Ng, A., Annett, M., Dietz, P., Gupta, A., and Bischof, W. F. In the blink of an eye: Investigating latency perception during stylus interaction. CHI 2014, 2014.

[42] O'Connor, T. mtouch projected capacitive touch screen sensing theory of operation https://www.microchip.com/stellent/groups/techpub_sg/documents /devicedoc/en550192.pdf, 2010.

[43] Omron. Omron g5v-2-h1 http://omronfs.omron.com/en_us/ecb /products/pdf/en-g5v_2.pdf, 2017.

[44] Ott, H. W. *Electromagnetic Compatibility Engineering*. John Wiley & Sons, 2011.

[45] Padre, J. Understanding the touchscreen ecosystem in smartphones, 2014. https://developer.sonymobile.com.

[46] Park, Y. B. Touch quality test robot. http://www.google.com /patents/us20130345864. Google Patents, 2013.

[47] Samsung. Touch controllers http://www.samsung.com/semiconductor /products/display-solution/touch-controller/, 2017.

[48] Schwartz, R. Normalized cpu load https://developer.qualcomm.com /forum/qdevnet-forums/performance-tools/trepn-profiler/26812, 2017.

[49] Shneiderman, B. Response time and display rate in human performance with computers. ACM Computing Surveys Volume 16 Issue 3, 1984.

[50] Stoffregen, P. Teensy 3.5 usb development board. https://www.pjrc.com/store/teensy35.html.

[51] Tanenbaum, A. S. *Modern Operating System*, 3 ed. Prentice Hall, 2007.

[52] Tietze, U. Electronic circuits: Handbook for design and application. Springer, 2015.

[53] Torvalds, L. input.h https://github.com/torvalds/linux, 08 2016.

[54] Vishay. Bpw34 https://www.vishay.com/docs/81521/bpw34.pdf.

[55] Vu, T., Baid, A., Gao, S., Gruteser, M., Howard, R., Lindqvist, J., Spasojevic, P., and Walling, J. Capacitive touch communication: A technique to input data through devices' touchscreen. IEEE Transactions on Mobile Computing (Volume: 13, Issue: 1), 2014.

[56] Yaghmour, K. *Embedded Android: Porting, Extending, and Customizing*, 1 ed. O'Reilly Media, 2013.

Appendix A

Results: Inter-touch interval

Phone	Interval	Result range	Mean	SD
Samsung S4 (540x960)	100 - 350 ms	14.0 - 21.2	17.6	2.0
Samsung S4 (540x960)	360 - ms	24.9 - 41.7	35.6	2.9
Samsung S4 (540x960)	370 - ms	19.4 - 32.8	25.3	3.2
Samsung S4 (540x960)	380 - ms	9.6 - 23.6	15.5	3.2
Samsung S4 (540x960)	390 - ms	9.6 - 27.4	20.6	3.8
Samsung S4 (540x960)	400 - ms	9.6 - 21.3	14.8	2.9
Samsung S4 (540x960)	410 - 640 ms	13.9 - 24.4	18.8	2.2
Samsung S4 (540x960)	650 ms	14.3 - 52.3	35.9	14.6
Samsung S4 (540x960)	660 ms	33.2 - 48.0	40.4	3.4
Samsung S4 (540x960)	670 - 5000 ms	23.6 - 37.9	29.8	3.3

Table A.1: Inter-touch interval vs. *touch to kernel* latency. (Samsung S4, measured at 540x960).