UNIVERSITY OF CALGARY

Evaluating the Performance of Direct Injection and TUIO-based Protocols for Multi-Touch Data Transfer

by

Darren Andreychuk

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 2012

Abstract

In the past two years, hardware vendors have released drivers for multi-touch devices running on Windows 7. Since direct injection of multi-touch input uses the Human Interface Device (HID) for USB protocol, it represents a new way of transferring touch data to client applications acting as an alternative to the Tangible User Interface protocol (TUIO) previously introduced. Anecdotally, TUIO has been criticized in the past for slowing applications down because of noticeable latency gaps between touch interactions and visual feedback from the system. This problem can result in poor user experiences and software quality degradation, both of which are contributing factors to device rejection in the marketplace. Over time touch-based systems have evolved into modular systems making it challenging to track performance bottlenecks in the overall system. This thesis focuses on high-performance multi-touch software systems by comparing HID and TUIO to determine which protocol contributes the lowest latency.

A semi-automated benchmark harness was constructed for an existing system to simulate and monitor different scenarios commonly observed in multi-touch interactions, with the goal of stress-testing both protocols without the presence of hardware. A performance evaluation was conducted on the modified system and the results indicate that, when compared to TUIO, HID is a faster protocol for multi-touch data transfer under constantly changing and often strenuous conditions.

Acknowledgements

I would like to thank my father, Ronald. The pursuit of this degree has been an exciting and exhausting journey. You have been there for me every step of the way. I could not have done it without your love, encouragement, and never-ending support. Thank you for always believing in me. To my extended families: the Ashcroft family, the Krakiwsky family, and the Pashulka family. Thank you for the love, support, guidance, and periodic comedy relief during this time.

To my professors at Mount Royal and the University of Calgary: Marc Schroeder, Paul Pospisil, Indy Lagu, Collette Lemieux, Charles Hepler, Ehud Sharlin, Craig Schock, Ruth Ablett, and Saul Greenberg. Thank you for the guidance and inspiration.

I would like to thank Rylan Cottrell, Theodore D. Hellmann, Yaser Ghanam, Bradley Cossette, Ali Hosseini Khayat, and my colleges in the Software Engineering Lab at the University of Calgary for their endless advice and offering different perspectives on my research. I would also like to thank Tak Fung at the University of Calgary for his statistical expertise and his good humor during the final stages of my data analysis.

I would like to thank SMART Technologies Inc. for funding of my research. Specifically, I would like to thank Benjamin Bozsa, Joseph Goethals, Taco van Ieperen, Erik Benner, and Edward Tse for their technical expertise and insight in various aspects of this work during my time at SMART. I would also like to thank Reed Townsend and Todd Landstad at Microsoft for their technical expertise regarding Windows 7 Touch.

Finally, I would like to thank my supervisor, Dr. Frank Maurer, for patiently guiding me through this process and for giving me a lot of freedom to explore. I learned so much during this time.

Dedication

To my late mother, Paulette, you gave me so much encouragement to pursue this degree during your final days. I wish you could have lived long enough to see this.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
API	Application Programming Interface
DI	Diffused Illumination
FPS	Frames Per Second
FTIR	Frustrated Total Internal Reflection
GUI	Graphical User Interface
HCI	Human Computer Interaction
HID	Human Interface Device
OS	Operating System
PC	Personal Computer
SDK	Software Development Kit
SPE	Software Performance Engineering
TCP	Transmission Control Protocol
TUI	Tangible User Interface
TUIO	Tangible User Interface Protocol
UDP	User Datagram Protocol
UI	User Interface
USB	Universal Serial Bus
WDK	Windows Driver Kit
WISPTIS	Windows Ink Services Platform Tablet Input Subsystem
WPF	Windows Presentation Foundation

Epigraph

You only live twice or so it seems,

One life for yourself and one for your dreams

:

This dream is for you, so pay the price

Make one dream come true, you only live twice.

John Barry, Leslie Bricusse, and Nancy Sinatra. *You Only Live Twice*.

Chapter One: Introduction

The release of the Windows 7 operating system marks a significant change for multi-touch devices. Now a fully integrated component in Windows 7, touch input has earned an equal standing among other input devices such as mouse and keyboard. [1] As a result, Microsoft undertook some of the responsibilities for multi-touch data processing and handling that are playing a vital role in multi-touch software development.

Multi-touch hardware vendors see this as an opportunity to alleviate some of their responsibilities to software developers as they can now inject touch data from their tracking software directly into the Windows OS. The OS processes incoming touch data and then sends it through the appropriate channels so that developers can access this data in a mainstream Application Programming Interface (API) of their choosing. As a result, vendors no longer need to provide custom-built touch-sensitive widgets in their Software Development Kits (SDKs). Hardware vendors, such as Dell, Hewlett Packard, and SMART Technologies, have acted quickly to supply Windows 7 drivers for their devices allowing software developers to easily port their multi-touch applications from one device to another without any additional development effort. [2] [3] [4]

This change has also had an impact on the multi-touch research community, which is composed of two primary groups: Device Hardware and Human Computer Interaction (HCI) researchers. The hardware group is mainly concerned with the underlying technologies that capture and process touch input and is less concerned about what the data will be used for after it is sent to client applications. This group focuses on all of the materials that are used in the device's construction and how to make the capture process as effective and efficient as possible. They test different sensing techniques to

determine the best way to capture touch interactions and investigate different display screen technologies to determine which will provide the best way to provide visual feedback. This group will also investigate the use of different materials for display screens to determine which will provide the appropriate amount of friction for touch interactions. Finally, this group investigates software for these devices but are concerned with finding the most efficient algorithms that are applied to process raw sensor data for multi-touch applications. [5] [6] [7]

The HCI group, on the other hand, is mainly concerned with the usability of multi-touch applications. They focus their efforts on what to do with the incoming touch data and how to use it. They investigate different interaction techniques for multi-touch software. They apply incoming touch data so that end-users can use their fingers and hands to interact directly with information displayed on the screen. They will also take incoming data and use it to create custom groupings which form the basis for gesture interactions. [8] [9] [10] Through this research, new multi-touch interaction techniques are conceived. Since this group is focused on what to do with touch data, they are less concerned with where the data came from or how it was sent to them. Situations do exist, however, when this group turns its focus to the data's source, because end-users have reported unexpected behaviour which affected their user experience on the device. This system, or functional, behaviour can range from hardware malfunctions to functional problems or slow responsiveness in the device's software.

Through their work both research groups intend to demonstrate how multi-touch devices can be commercialized by producing efficient, low-cost devices, with a wide range of software and share a common goal of providing a rich user experience.

However, a large gap still exists between both sides of this community in spite of their joint efforts. Another group, however, is emerging. Its members are concerned with logistics, system performance, and tuning. They focus on how to send multi-touch data efficiently to applications by creating middleware. Evidence of this can be found in the Windows Touch system for Windows 7. [1] This system provides a common interface so that all multi-touch devices can send data to the OS where it is processed and shipped to multi-touch applications.

The common goal that all three groups share is to provide the best user experience for multi-touch devices. However, they go about achieving this in different ways and one of these ways is in the realm of performance, which is a term used to describe efficiency. Each group has its own definition of this term and it is important to make a clear distinction between measuring performance in usability evaluations, hardware devices, and software systems.

Hardware performance evaluates different types of capture technologies along with the building materials that are used in the construction of multi-touch devices to determine which materials yield the best results for capturing touch interactions. [7] Performance measurements in usability studies evaluates how efficiently participants complete a series of tasks on multi-touch devices to determine which techniques yield the lowest completion time with the fewest amount of errors per. [11] System performance evaluates the amount of time a system takes to process all of the data generated by end-users, or some other source, to determine which software components require further tuning in the system so that they will yield the lowest delay when responding to touch interactions. [12] [13]

The common focus is on the user when designing the overall product and all three groups share this when evaluating performance to provide the best user experience for devices they are interacting with. This thesis is aimed at continuing the work that has been conducted by the middleware group to help bridge the gap between the hardware and HCI groups in terms of performance.

This chapter defines concepts that are used throughout the thesis. First an overview of Touch Input is provided followed by an overview of a Windows 7 multi-touch software system. The concept of multi-touch system performance is presented along with a common problem shared by these systems. Finally, a list of research questions that motivated this research is presented.

1.1 Touch Input Overview

People interact with computers using various types of input devices, which have included: mouse, keyboard, joystick and stylus. These devices have been in use for over 30 years and the way people interact with computers has changed very little during this time. [14] Touch input allows people to interact with the User Interface (UI) of an application with their fingers and provides an interesting new alternative to other pointing devices such as pens, mice, and joysticks. Touch input also opens up a new realm for interaction techniques and can change the way people interact with computers. [15] Even though it is often referred to as the “new” input device, touch input has actually been around since the 1980s and possibly earlier. [16] Rather, touch input should be regarded as the “forgotten” input device because the amount of research in this area has only increased considerably within the last 10 years with the introduction of applications that demonstrate various multi-touch interaction techniques. [8] [9]

With so many of these techniques re-occurring in applications, developers have started creating reusable software libraries, commonly known as Application Programming Interfaces (APIs), to ease development effort. [17] [18] [19] These APIs, often created by hardware vendors, have been developed for individual devices because earlier versions of the Windows operating system (and other OSs) have lacked built-in multi-touch support. Therefore, touch input has been treated as a special form of input and has been kept separate from mouse and stylus input.

In Windows 7, touch input was fully integrated into the input device hierarchy. This integration has slowly progressed from a separate pen and ink add-on services for Windows XP to single touch support through pen and ink support in Windows Vista to full multi-touch support in Windows 7. [1] In order to understand how the features can be utilized by applications, it is important to identify and define the individual components of the entire system that will be deployed on Windows 7. Breaking a system down into its individual components is a commonly used practice that helps developers locate and fix bottlenecks in large-scale software, which can also be applied to multi-touch software systems. [12]

1.2 Multi-Touch Software Systems

A multi-touch software system is a collection of software components that, when combined together, acts as a communication and processing pipeline for multi-touch data. These systems are driven by data generated by touch interaction. Each system component represents a stage, or a filter, along the pipeline where raw data from a capture sensor is refined on its way to client applications. Over time these systems have evolved into modularized client/server systems that utilize a layered architecture. [20] [21] Figure 1.1

illustrates the layers of the communication pipeline and how each component of a Windows 7 multi-touch software system fits together.

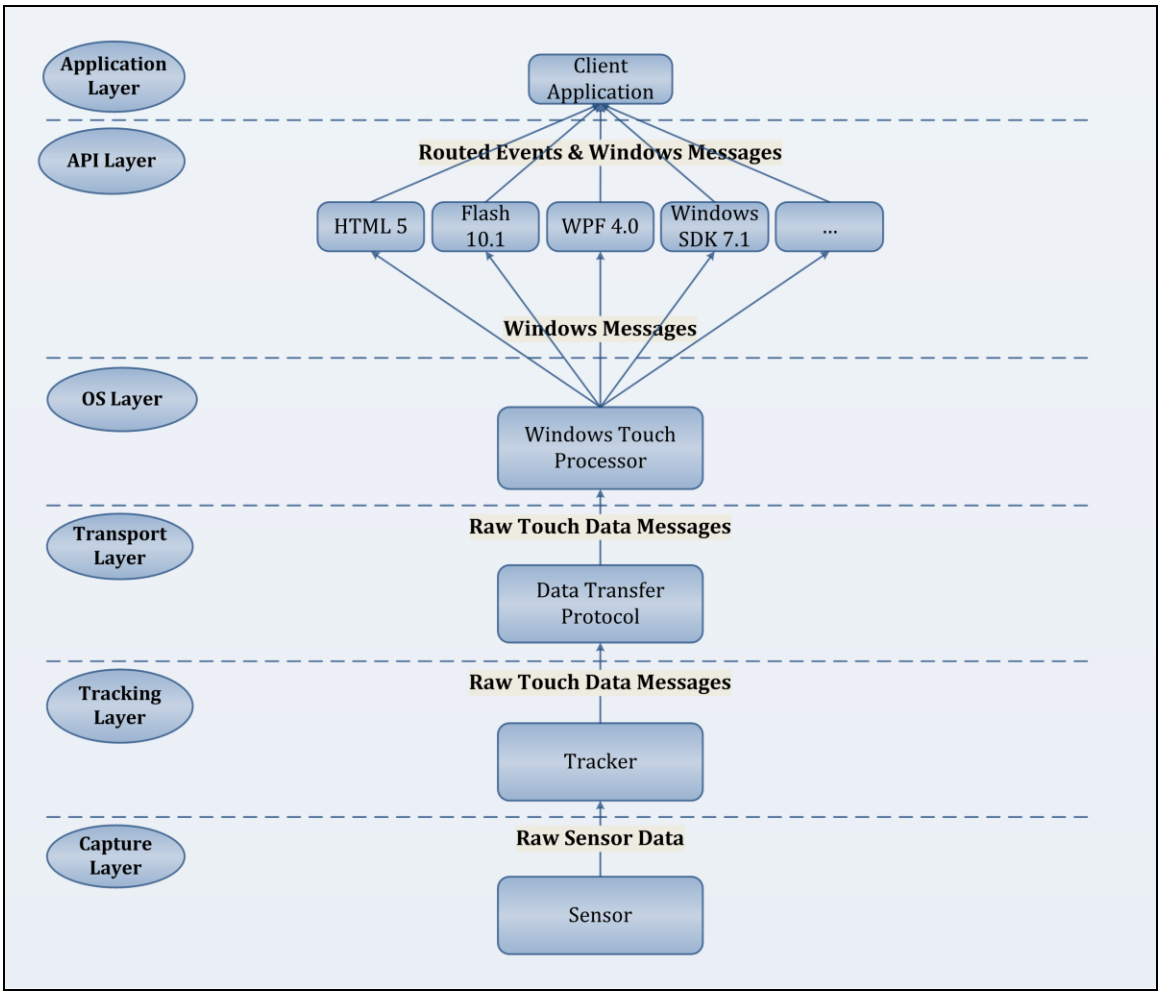


Figure 1.1 The Components of a Multi-Touch Software System for Windows 7

The system is composed of five layers: Application, API, OS, Transport, Tracking, and Capture. The capture layer represents the start of the pipeline where individual touches are captured by sensing devices, such as infrared cameras. [22] These devices will transmit raw data to a multi-touch tracker where it begins an arduous process of refinement before it is received by a client application.

1.2.1 Tracking Layer

The Tracking layer, often relying on Computer Vision, represents the first filter in the pipeline of the system. It is responsible for tracking all touches currently making contact with the screen of a device and it acts as the server for a multi-touch system. Multi-touch trackers are often regarded as the most complex part of the system because they are responsible for image processing and data packaging for transport to client applications. Figure 1.2 illustrates the individual components of a multi-touch tracker.

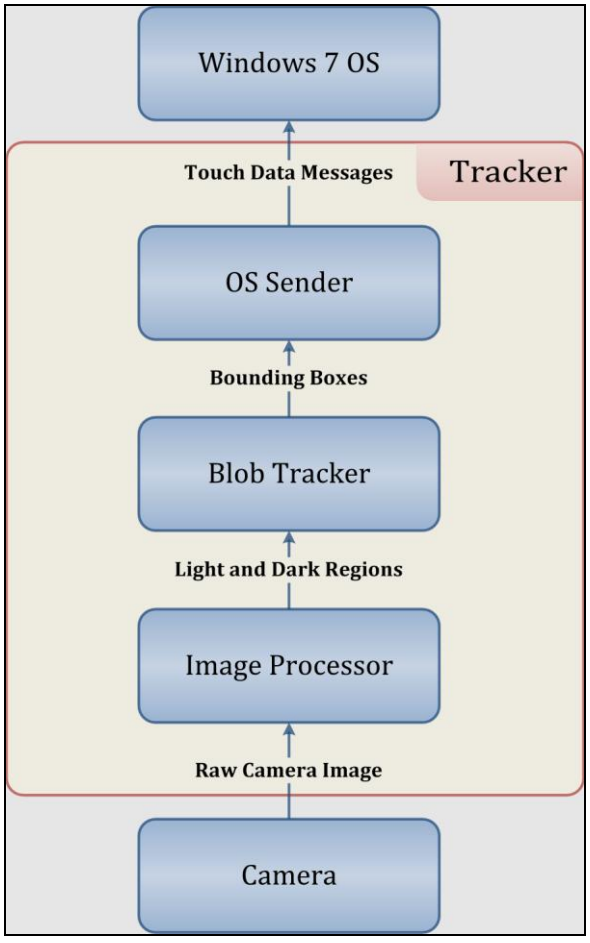


Figure 1.2 The Components of a Multi-Touch Tracker

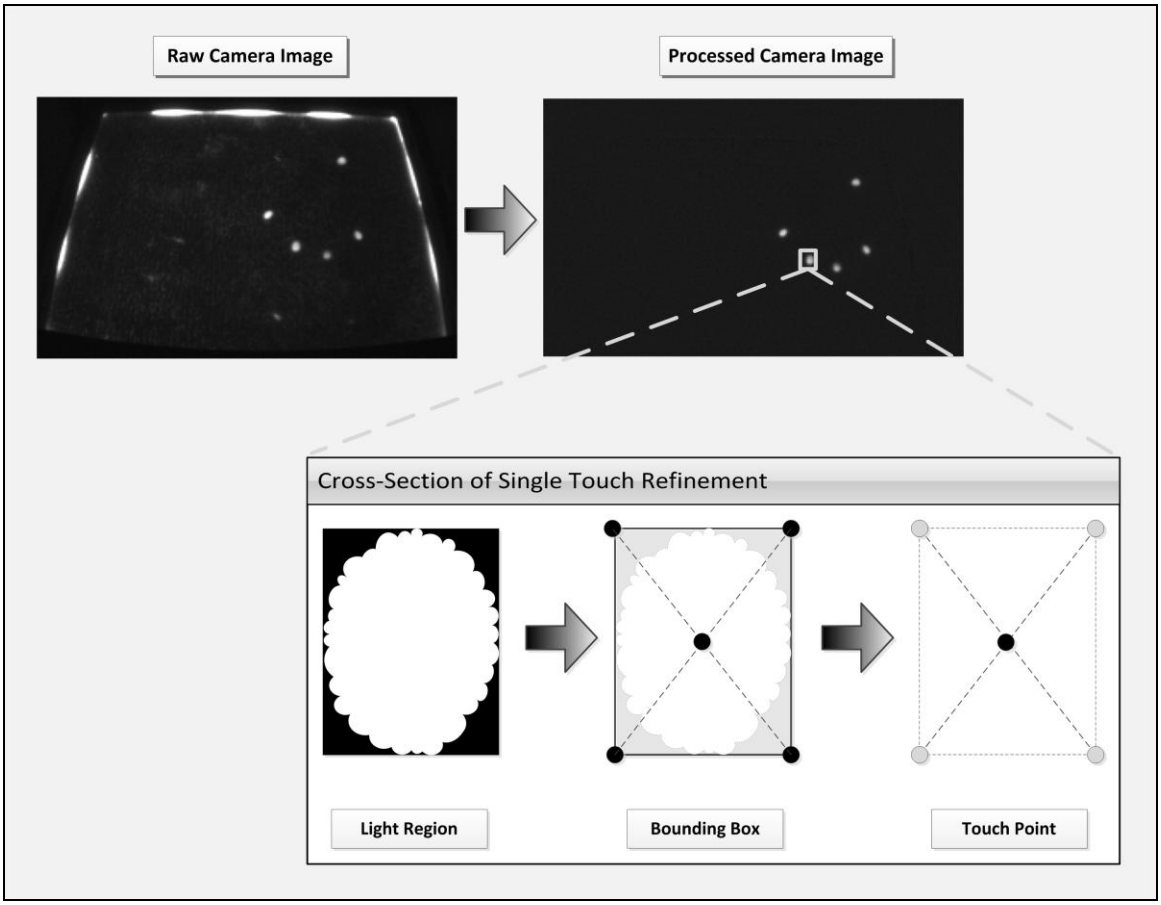


Figure 1.3 The Refinement Process of a Multi-Touch Tracker

Figure 1.3 illustrates the refinement process that each individual touch goes through before it is sent to the OS. Raw images are sent from a camera at a pre-defined frame rate and it processes this image. The raw image is cleaned up so that only regions of light and dark remain in the image. This process often requires multiple image processing algorithms to completely refine the image. [5] [7] Regions of light are extracted from the image and officially become locations where touches have occurred. Various calculations such as location, area, orientation and pressure, of the light region are calculated. The regions are then stripped of their individual pixels and a skeletal

version is created, which is commonly known as a bounding box. At this point the light regions become “touches.”

Before the data is sent to the OS it must be packaged up in a format that the data transfer component will accept. During this process the touch still has an area but the coordinates of the bounding box are funnelled down into a single point, the center point of the box, becoming the official location of the touch. The touch is also given an identifying number along with a state, which typically include: down, update or up. [5]

1.2.2 Transport Layer

Since multi-touch software systems have evolved into client/servers systems, it has become important to define a separate layer responsible for transport services. This layer is often referred to as middleware and is driven by protocols. A protocol is the pre-defined format in which data is sent from one software component to another. [12] [20] For multi-touch systems this layer is responsible for transferring processed camera data to the Windows OS. Currently, there are two main protocols for multi-touch data transfer: Human Interface Device (HID) for Universal Serial Bus (USB) and the Tangible User Interface Protocol (TUIO).

HID for USB, is a low level software interface for USB-supported input devices such as mouse and keyboard. Device drivers are written so that chunks of device data, also known as packets or messages, are injected into the OS where they are processed and handled by the OS and then automatically passed along to client applications. Therefore, multi-touch data messages are transferred using the same type of input buffer that other device messages travel along. [23]

TUIO is a protocol for multi-touch data transfer and was first introduced in 2005. It was originally designed to handle touch input for Tangible User Interfaces (TUI) and is actually built on top of two other protocols: Open Sound Control (OSC) and User Datagram Protocol (UDP). [24] The OSC protocol was created for transferring data from music devices to a central computer and is built on top of the network protocol UDP. [25] [26] A TUIO server opens a UDP socket to transmit data.

At a conceptual level, the HID and TUIO protocols represent further efforts to break up the transfer stage of a multi-touch system into a separate layer in the system. This can have many benefits as it contributes to system modularity. Protocols reduce the coupling between client applications and specific trackers. It also means that developers are afforded the flexibility of replacing problematic components with better functioning ones. [20] However, at a technical level, both protocols have similarities and differences. HID APIs are typically lower-level implementations that only expose a raw injection buffer. TUIO APIs are usually higher-level implementations where a low-level OSC buffer is encapsulated in higher-level method calls. Both protocols send multi-touch data to clients, but do it indifferent ways.

1.2.3 Operating System Layer

Processes running in the Windows 7 OS are responsible for dealing with incoming touch messages and translating the data from the protocol's message format into a uniform callback format used by an API. At this stage the data encapsulated inside the messages undergoes additional processing for the environment that the OS is running on. The system responsible for processing and handling is called Windows 7 Touch. [1]

1.2.4 API Layer

After processing has been completed, the data is then packaged up and sent to an API where it is channelled to client applications. These APIs are part of the mainstream of software development and are used by a vast numbers of developers. APIs such as Windows Presentation Foundation (WPF) and Flash are a few mainstream APIs that directly support multi-touch input. [27] [28] They receive touch data in the same input layer that they receive mouse or keyboard data. From here they encapsulate the data inside their own callback structure. These callbacks are the communication channels for different forms of data within the API and they carry touch data to the presentation layer where developers have access to it. [27]

1.2.5 Application Layer

This layer represents the final stage of the data transfer pipeline where the processed touch data is used by software developers. This layer is typically responsible for providing feedback to end-users and is where they interact with a device. Applications developed using mainstream APIs, such as WPF 4.0, can be ported from one multi-touch device to another provided the device has a multi-touch driver for Windows 7.

1.3 Multi-Touch System Performance Overview

Multi-touch systems, as demonstrated in the previous section, are comprised of many different software components. They exhibit similar behaviour shown in other client/server software systems with layered architectures. Multi-touch systems are susceptible to the same problems that other client/server systems face over time. These problems can include how to effectively test individual components to ensure functionality, how to evaluate usability to ensure a smooth user experience, how to

address known performance issues that have crept into the system over time, and how to effectively maintain these systems over time to ensure a steady stream of releases with the least amount of effort for the development team. [12]

Multi-touch systems for Windows 7 Touch are quite complex and can contain multiple individual processes that drive the data pipeline. The tracker has its own process. A TUIO socket and a multi-touch driver can also have processes running in the background of the OS. The APIs and client applications could also have multiple processes that handle different tasks. With individual components spread across multiple processes, tracking functionality and performance issues can be a daunting task.

Multi-touch systems are also driven by data, since a single touch interaction is capable of generating hundreds of touch messages for the system to process. Each component in the system is responsible for not only processing large amounts of data correctly, but also as quickly as possible. This thesis focuses on the performance aspect of these issues from the system's perspective at the transport layer, which means the system will be evaluated on how fast it transfers data to a client application.

Previous work in this area has been limited to other areas of performance that focus on the end-user's perspective. Indeed, it is at the application layer where end-users and developers notice usability, functionality, and performance issues because this layer is responsible for providing various forms of feedback to end-users. Very little work has been published in the past that focuses specifically on system performance. This is a problem because research in this area can help developers make clearer distinctions between performance issues in their applications versus performance issues bubbling up from the underlying system.

A common problem with multi-touch devices is that applications appear to be slow in responding to user input. Specifically, there is a noticeable time gap between touch interactions and the system responding with the appropriate feedback. [29] [30] Figure 1.4 illustrates this issue in more detail.

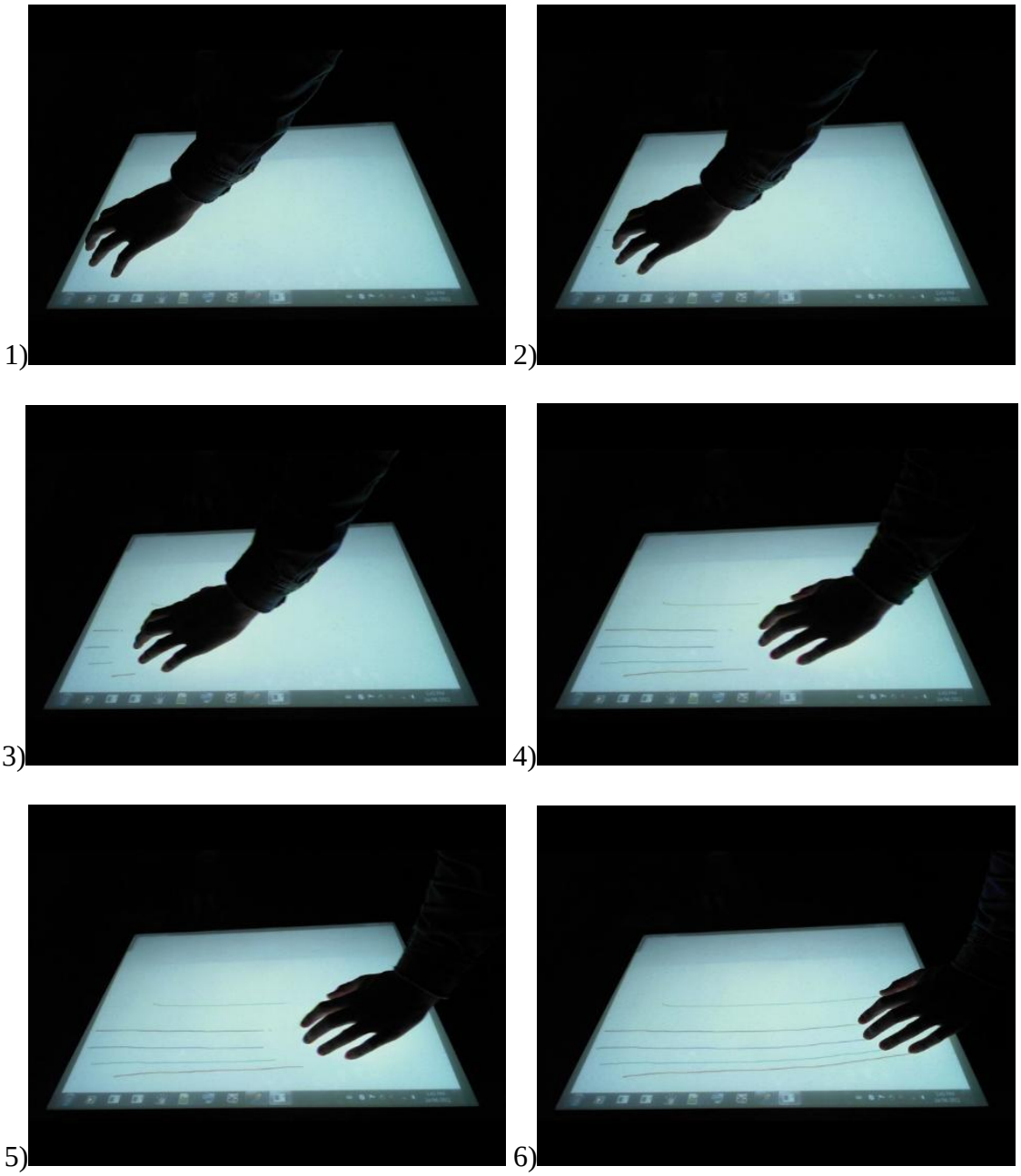


Figure 1.4 A Noticeable Visual Delay at Different Stages along an Interaction

If a user is moving a picture from one side of the screen to the other or drawing ink strokes across the screen there is a noticeable time gap between the touch interaction and the system responding to the interaction by providing the appropriate visual feedback. If a device is being used a musical instrument, users notice the same gap between touch interactions and the system responding with audio feedback. [31] This time gap is called latency, which is a term used to describe delay and is measured in milliseconds (ms). [12] [13]

The release of Windows 7 Touch has brought the HID protocol back into the foreground now that it has been expanded to include multi-touch input. However, it is still unknown whether or not it is a suitable protocol for multi-touch data transfer. Furthermore, software developers have made claims in the past that using TUIO slows their applications down. [29] [32] [33] [34] With different developers reporting this common problem, it is less likely a problem in their own applications and it is likely a problem originating from somewhere in the underlying system. These problems are typically reported from the user's perspective since it is most noticeable in the application layer of the system. However, the problem must be diagnosed from the system's perspective by breaking the system down into its individual components so that they can be isolated for analysis. [12]

Referring to Figure 1.4, it is not yet clear which component, or components, in the underlying system are causing this latency gap and it generates many questions that can be difficult to answer with complete certainty. Is the tracker taking long to process the images from the camera? Is the transfer component taking too long to send the data? Is the problem occurring in the APIs or client application? If the problem is truly in the

transfer component, then at what thresholds do both protocols begin to show stress and what impact would this have on client applications using it? [35] [36]

With these problems still plaguing multi-touch software development, and no solid answers, it was time to address these issues. This thesis applies the same methodologies used to measure client/server system performance to measure multi-touch system performance. This thesis describes the challenges faced when applying traditional performance methodologies to multi-touch systems and how they were overcome.

1.4 Research Questions

Given the problems presented in the previous section, the central question that this thesis will attempt to answer is in the transport layer of a multi-touch software system. HID and TUIO are compared in an attempt to answer the following questions:

1. Is HID faster than TUIO and under which conditions do they exhibit stress and start to slow down?
2. Can traditional performance methodologies help researchers answer this question?

To make the measurement process controlled and repeatable, this thesis will replace actual user input with simulating various touch interactions so that the system could be driven without a camera. This opens up the possibility of conducting software performance evaluations off of the deployment environment where it could be evaluated manually. This way automated performance testing can be conducted off of the device and specifically focus on software while manual performance testing can be conducted on the device to include hardware components. The goal in addressing this problem is to have both methods complement each other during the development lifecycle.

Previous systems, such as the MS Surface Simulator and the Multi-Touch Vista project, have attempted to simulate touch input through mouse devices. [37] [38] This method helps developers test client applications off of the deployment environment. However, it still requires mouse devices to carry out the actual execution of a touch interaction. Multiple mouse devices are unsuitable for performance testing and make it impossible to simulate input on a device that can support 50-100 concurrent touches. Furthermore, repeatedly gathering multiple people around a larger multi-touch device, such as a tabletop, to produce this many touches becomes even more troublesome. This problem is addressed in the following question:

3. Is a camera the only way to capture touch input or can it be simulated directly inside of a multi-touch tracker before touches are sent to the OS?

Another problem is in the area of tool support for evaluating multi-touch system performance. While performance evaluation methodologies for other areas of computing may be compatible with multi-touch systems, it is still unknown whether or not current tool support for these methods can also be applied. For example network protocol analysis tools, such as Ethereal and Wireshark, are real-time monitoring tools that conduct various performance calculations on incoming messages from different protocols. [39] [40] However, it is unlikely that these tools can be applied to multi-touch protocols because they were created to evaluate network protocols. This problem generates the following research question:

4. What is an efficient way of analyzing multi-touch protocol data?

1.5 Research Goals

The primary goal of this work is to determine which protocol is faster for multi-touch data transfer. The purpose of this goal is to provide the multi-touch research community with set of conclusions, based on the results of a formal and repeatable evaluation. These results can also help software developers make better informed decisions when choosing between HID and TUIO in their software.

The secondary goal of this thesis is to demonstrate how traditional performance methodologies can be applied to multi-touch system performance and to report on the process that was followed to answer to the first research question. The purpose of this goal is to report on the technical challenges surrounding multi-touch protocol performance evaluations and to encourage further work in this area.

1.6 Thesis Overview

This chapter described performance concepts and problems surrounding multi-touch software systems. The remainder of this thesis is divided up into two categories: background material and contributions to the area of multi-touch system performance research. The background material begins with a review of the work that has already been conducted in multi-touch performance. Chapters 3 and 4 provide background information on the HID and TUIO transfer protocols to explain their similarities and differences. Chapter 3 also provides an in-depth look at the Windows 7 Touch system and its software components to help put multi-touch driver development into perspective. Chapter 5 provides an overview of Software Performance Engineering methodologies that were applied to the design of a performance evaluation comparing HID and TUIO.

Chapters 6, 7, 8, and 9 all discuss the main contributions of this thesis. Chapter 6 presents the details of how a multi-touch protocol performance evaluation was designed and implemented to determine which protocol was faster. Chapter 7 presents the final results, and methods used to calculate these results, which indicate that HID is faster than TUIO in situations involving multiple concurrent touches being generated by multiple concurrent users interacting with devices.

Chapter 8 provides an in-depth discussion of the final results to help developers understand the contributing factors responsible for performance degradation and what impact this can have on their multi-touch applications. Chapter 8 also outlines the limitations and technical challenges faced when attempting to make a fair comparison of both protocols to provide developers with a solid list of issues they will likely face when attempting to design their own performance evaluations. Finally, chapter 9 presents the main contributions and provides answers to all of the research questions. Finally, possible avenues for future work are outlined in the area of performance evaluations for multi-touch software systems.

Chapter Two: Related Work

Much research has already been conducted in the areas of multi-touch hardware technologies and interaction techniques for multi-touch software. Performance is typically evaluated solely from these two perspectives. Unfortunately, very little work has been published in the general area of performance evaluations that focus on the responsiveness of multi-touch software systems to end-user interactions and no work has been published in the specific area of performance evaluations that focuses on HID and TUIO to determine which protocol yield lower latencies these systems. Furthermore, the area of multi-touch software system performance still needs to be better defined for the research community. The concept of system performance needs to be separated from hardware performance and performance measurements in usability evaluations. This chapter outlines previous work conducted by other researchers in all three areas multi-touch performance and closes with some concluding remarks.

2.1 The Roots of Multi-Touch Performance

Multi-touch systems can be traced as far back as 1985 when Buxton *et al.* reported on the possible benefits and limitations of including touch input alongside other types of pointing devices for touch tablets. This paper is an early attempt to map end-user tasks from mouse and joystick input devices to touch. A touch tablet device was constructed along with a tracking system and set of different applications were developed to demonstrate interaction techniques for the device. [16]

The term performance was used in the paper but it was not evaluated. However, problems regarding surface friction, jitter, and the need to provide suitable visual feedback for touch interactions were reported. It also mentioned that touch input was

suitable for some types of interactions and not for others. Touch input therefore was not a replacement for other input devices and several issues were reported with touch tablets when end-users interacted with the device. [16]

An interesting aspect of this paper was the construction of cardboard templates to control what areas of the tablet screen were touched. Square and rectangular holes were cut in the template so that two different types of interactions could be tested. When the template was placed over the tablet screen touches were restricted to the same locations on the screen and each region in the template represented a separate device.

Figure 2.1 Sample Templates Placed on the Screen of a Tablet Device

The template's purpose was to provide tactile feedback around the touch to give the feeling of actually touching a menu item on the screen. As illustrated in Figure 2.1, square holes cut into the template were used for typing interactions as there were no physical buttons on the tablet's screen. Even though this idea was used in a different context in Buxton's paper, it helped inspire the idea of controlled touch interactions for the experiment in this thesis. [16]

Even though Buxton's paper did not formally evaluate the performance of the hardware and software for the tablet device, it did provide a good place to start when

attempting to put performance in a hardware and end-user perspective. Since then, performance has followed these two streams. In the context of hardware performance evaluations have focused on comparing different capture devices, display technologies and low friction materials for touch screens. [5] [7] In the context of usability evaluations, performance measurements have focused to comparing how end-users perform tasks using touch input versus how they perform the same tasks using other input devices. These measurements have also focused on how end-users perform tasks using different interaction techniques, such as gestures. [8] [9] [10] [15] [41] [42] [43] [44]

2.2 Previous Multi-Touch System Performance Work

In his 2008 thesis, Laurence Muller conducted a usability evaluation of multi-touch software and a performance evaluation of multi-touch hardware with the purpose of demonstrating how touch input can be utilized on digital tabletops. His work was divided into two separate areas: device construction and multi-touch software systems. [6]

Muller's thesis was completed at a time when tabletops had not yet been introduced to the marketplace. Therefore, the first step was to construct devices using popular multi-touch capture technologies used at the time. Two prototype tabletops were constructed, one using Frustrated Total Internal Reflection (FTIR) technology and one using Diffused Illumination (DI) technology. [5] [7] Building the tables from scratch allowed Muller to report on the process that was used during construction. Furthermore, he was able to report on the benefits and drawbacks of choosing different camera products for touch input capture, different projector products for the highest quality visual display, and the best suited materials for other parts of the tables. [6]

A multi-touch software system was constructed for both tabletops. Figure 2.2 illustrates each of the components in this system. The system was comprised of all the components introduced in Chapter 1, with the exception of the Windows Touch system since it pre-dated the release of Windows 7.

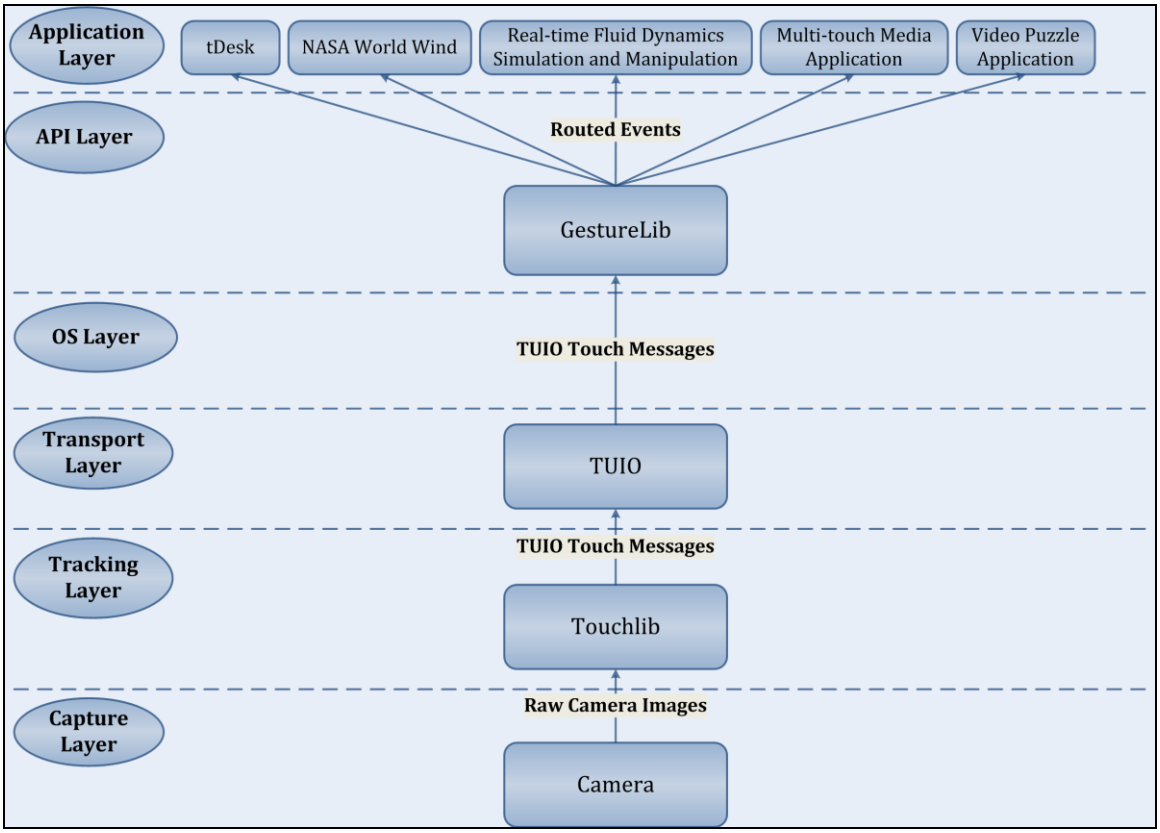


Figure 2.2 Lawrence Muller's Multi-Touch Software System

Touchlib, an open source computer vision library, was used to handle the image processing. [45] Touchlib sends touch data to client applications through TUIO. A set of applications were created to showcase a variety of multi-touch interactions such as panning, zooming and tapping interactions. These interactions were built in a separate gesture API, called Gesturelib. [6]

The evaluation is broken up into two separate stages. The first stage is a usability study to determine how effective touch input is compared to mouse input. This stage closely resembles other work comparing the two input devices. [15] [16] Participants were given a series of tasks to complete, which included object manipulation, collaborative sorting, and a collaborative point and select task. [6]

The second stage was a performance evaluation of the multi-touch system as a whole to determine how fast the device's hardware and software system processed and handled touch data generated by end-user interactions. The software system was broken up into individual components where touches were logged by the system by creating benchmarking tools. Each component's individual response times were logged and calculated. Different hardware components were compared during the study. This included an FTIR tabletop and DI tabletop along with 8 different projectors. [6]

The system latency was measured at different points inside of the Touchlib tracker. Specifically, the individual stages of the image processing were measured. Also, the latencies for the blob tracker and event dispatcher were calculated in the tracker. Two interaction scenarios were observed in Touchlib: 0 touches and 5 touches. Scenarios that had a higher number of touches were not attempted because the algorithms inside Touchlib relied on the CPU for image processing. [6]

The TUIO transfer component and Gesturelib were not individually evaluated. Instead the total system latency, the time when the raw images arrived to when the processed data was displayed on the projector, was calculated by subtracting transmission times from the arrival times using custom-built benchmarking tools. Unfortunately, specific details were not provided as to how these tools functioned in the system.

Two projectors, the slowest and the fastest, were compared to determine what effect a higher system latency would have on participants using the system. The total latency for the slowest projector turned out to be 195 ms, while the total latency for the fastest projector was 65 ms. The latency for the slowest projector was reduced to 163 ms when a bug, that stalled touch data for 32 ms using a sleep statement, in Touchlib was fixed. A similar situation was faced during the course of this research and will be explained in Chapter 6. [6]

Muller's evaluations have many interesting aspects to them. He went beyond hardware performance and software usability and moved into the realm of software system performance. He provided, in great detail, all of the components of a multi-touch software system and how each of them worked. This is very important when evaluating the performance of any software system. However, one limitation to his approach was that tabletops had to be constructed before the study was conducted. This required a substantial amount of effort in order to achieve his research goals. This issue was beyond his control at the time due to lack of commercial products. However, it remains a risky endeavour for most of the researchers in the multi-touch community especially if they are primarily focused on writing applications for these devices. [6]

Another possible limitation was that an entire multi-touch software system, with the exception of Touchlib, had to be constructed thus increasing the amount of implementation time. However, the benefit in doing so gave the researchers complete access to all of the code in the system. In the four years since Muller completed his thesis, tabletop devices have entered the marketplace so researchers do not need to construct these devices nor do they need to build basic software systems for them.

However, the downside to this is that the software systems are closed off to developers hoping to attempt new ideas in the code.

Each application used specific gestures from Gesturelib and thus limiting the number of touches captured by the system to 1-5 simultaneous touches. It is important to note that at this time the image processing algorithms that converted raw camera images into processed touch points utilized the CPU. From a performance perspective, this means that only 5 concurrent touches could be efficiently tracked by the system because exceeding this threshold would slow down the system considerably due to CPU overload. Since then, and as Muller originally suggested, the image processing was been moved over to the GPU allowing upwards of 100 touches to be tracked at the same time. Furthermore, the image processing is currently making another transition and this time it is from the GPU to over to the camera's firmware in the hopes of further bringing down the overall system latency. [46]

The term performance does have a split meaning in Muller's thesis. It is applied to hardware and software latency and it is also applied to the context of usability. For example, limitations in building materials led to a great deal of friction on the tabletop surfaces and the overall accuracy of touch input was lower when compared to mouse input. These issues, regardless, affected the way users were able to interact with both devices thus degrading their user experiences. It is important to separate these perspectives when attempting to address and fix these problems.

One comment that re-occurred throughout Muller's thesis is the lack of built-in multi-touch support in the Operating Systems of the day. Converting incoming multi-touch data from TUIO's callbacks to each of the API's callbacks became a reoccurring

challenge. [6] This challenge was partially solved with the release of Windows 7 Touch and the Multi Touch Vista bridge for TUIO. [1] [38] However, the problem of conversion also occurred in the evaluation for this thesis and is explained in chapter 5.

Muller's thesis stands as an excellent early attempt at evaluating the performance of individual hardware and software components of a multi-touch device. It demonstrates the role that latency plays in user interactions and serves as the main source of inspiration for this work.

More recently Montag *et al.* reported on the issue of high latency in multi-touch devices through a performance evaluation. [31] Similar to Muller's evaluation, an FTIR tabletop was constructed and a multi-touch system was constructed out of existing software. [5] [6] The purpose of the evaluation was to address the issue of high latency in multi-touch devices that were being used as musical instruments. This work is significant because it defined latency as the time between touch interactions and audio feedback.

Montag *et. al.'s* evaluation mainly focused on system performance. However, it also included device and software system construction. Community Core Vision (CCV), an open source computer vision library, was used as a tracker. [47] TUIO was used to transfer data to client applications. The resulting system produced an overall latency of 30 ms which turned out to be lower than average of 75 ms previously reported for binaural audio displays. [48] Three different cameras were compared and the one contributing the lowest latency was used in the final results. The term performance was also used in different contexts in this paper, as it was used to mean latency and efficiency. However, performance also refers to how end-users play musical instruments. This adds further resolve to ensure there is a clear separation between perspectives. [31]

This thesis continues the work of defining performance for multi-touch software systems, but only for software and ignores hardware components. This thesis also reports on the process followed to help developers understand all of the issues surrounding multi-touch system performance.

2.3 Summary

This chapter introduced previous work conducted in the area of multi-touch system performance. Montag *et al.'s* work demonstrated that high system latency is not only a problem for applications that provide visual feedback, but is also an issue for applications that provide audio feedback. [31] System performance, specifically latency is an important aspect to study because end-users notice high response times when they interacting with multi-touch devices. Devices, and the software running on them, can appear to be slow when users interact with them and touch interactions lose their natural smoothness. Unfortunately, high latency issues must be addressed because they do not always disappear by purchasing faster hardware as some of these issues must be addressed in how developers implement software for multi-touch devices.

Muller's work provided inspiration when conceiving the conceptual design for this evaluation. For example, calculating the system latency for 0 and 5 touches provided inspiration using the number of simultaneous touches as an independent variable. [6] Buxton's idea of a paper template resting on a tablet's screen provided inspiration for different types of touch interactions. [16] However, before the contributions of this work are presented, it is important to provide background material surrounding the HID and TUIO protocols. This is started with an in-depth look at the Windows 7 Touch system.

Chapter Three: The HID Protocol

Starting with Windows 7, Microsoft fully integrated multi-touch input into its operating system. As result the OS provides a common interface for touch so that any multi-touch device can inject its data directly in the OS. The OS reads incoming data traveling through an input buffer and it does this in the same way that it reads incoming data from a mouse device or keyboard device. This data is read and processed before it is sent to a client application. Kiriaty et al. used the analogy that touch input has finally earned an equal standing, or its "citizenship", among input devices. [1] It is important to understand what exactly built-in touch support for Windows 7 actually means. Multi-touch integration can be understood by defining the Windows 7 Touch system.

This chapter provides an in-depth look at Windows 7 Touch and the underlying Human Interface Device for USB protocol. By writing low-level device drivers, developers can pass the final stages of multi-touch data processing over to the Windows 7 OS. This chapter is aimed at providing background information surrounding the Windows 7 Touch system and the HID protocol with the goal of putting OS multi-touch integration into perspective. This chapter begins with a description of Windows 7 Touch and its basic components. This includes a description of the Windows 7 Touch processor and how software developers can obtain access to the data after it is processed. This leads to a review of the HID for USB protocol specifications and how developers can go about writing a multi-touch driver for their device.

3.1 Windows 7 Touch: A Look under the Hood

The Windows 7 Touch system is a collection of software that acts as a common interface for touch data. The goal of Windows 7 Touch is to provide a common input

stream for all multi-touch devices. This means that all devices are treated equally and there is no difference between a digital tabletop device, a vertical display or even a mobile device. Windows 7 Touch support is achieved by writing low-level multi-touch device drivers, which are typically provided by the hardware vendor. [1] Application developers can access incoming touch data from any device using the same callback format that they are accustomed to using in a mainstream API. The Windows 7 Touch system can be divided into two categories: Processing and Handling.

3.1.1 The WISPTIS Processor

The multi-touch processing component for Windows 7 Touch is called WISPTIS, which stands for the Windows Ink Services Platform Tablet Input Subsystem. WISPTIS is responsible for refining raw data from the device's HID input buffer for the hardware environment. [1] It will convert the position coordinates of each message from a spatial pixel format used by the image processor into a screen pixel format based on the device's display resolution.

WISPTIS also conducts error handling, such as data loss, so that any irregularities that occur from the underlying protocol can be corrected so that it will have a minimal impact on the overall user experience in a client application. For example, if WISPTIS receives a series of messages that indicate a touch is moving but it did not receive an initial down message that forms the start of the interaction's message sequence then it will automatically create this message. It will also deal with climbing latencies. For example, if the latencies of the incoming messages begin to climb and gets too high WISPTIS will decide to drop a chunk of messages and not process them. Finally,

WISPTIS also contains a gesture processor and will group touch messages together. These groupings are mapped to the pre-defined gesture set in Windows 7 Touch. [49]

The handling side deals with packaging and shipping of multi-touch data through various inter-process communication channels, also known as callbacks or delegates, so that software developers can access the data in their applications. [50] In Windows 7 Touch, there are three different levels of multi-touch data access: high level access through the Windows Presentation Foundation (WPF), lower level access through the Windows SDK, and at the driver level through the Windows Driver Kit (WDK).

3.1.2 WPF 4.0: Higher-Level Touch Access

Starting with WPF 4.0, touch input was directly inserted into the input device hierarchy, which is illustrated in Figure 3.1. Multi-touch data is automatically channeled from the input layer of the API to widgets in the presentation layer. The widgets in the Framework Element hierarchy have been upgraded to listen for incoming touch data. Software developers can obtain access by registering widgets, such as canvases or buttons, to data through routed touch events. [27]

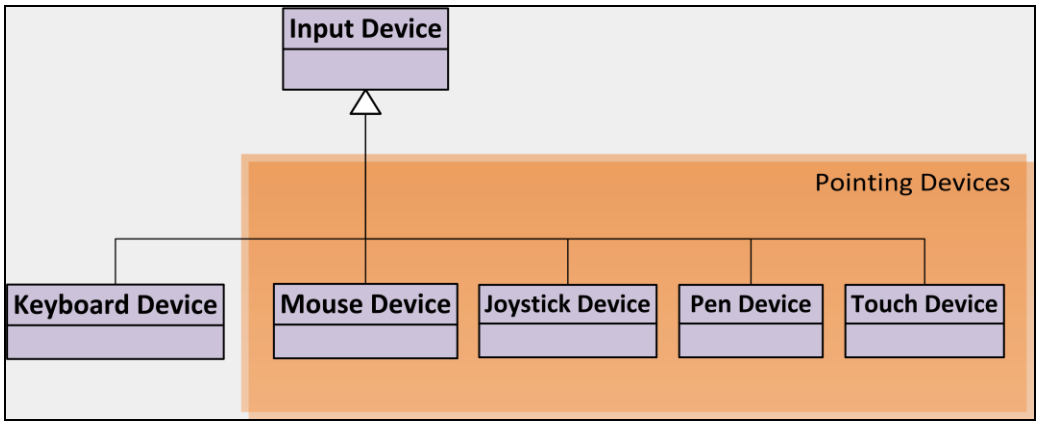


Figure 3.1 The Input Device Hierarchy in WPF 4.0

3.1.2.1 Callback Structure: Routed Events

WPF uses routed events and delegates to handle inter-layer communication within the API where each event is a line of communication. Incoming data is unpackaged and channelled through these lines to the widgets. Starting in WPF 4.0, a new TouchDevice class was extended from the InputDevice and inserted alongside a MouseDevice and a KeyboardDevice in the hierarchy. [27] Touch input data is accessed through two new sets of events: Touch Events and Manipulation Events.

3.1.2.2 Touch Events: Touch Message Data

Touch events represent individual touch messages that were injected into the OS. However, at this level the processed message data has been encapsulated inside the classes surrounding the events. In WPF 4 there are three Touch Events: TouchDown, TouchMove and TouchUp. The TouchEventArgs class exposes touch data to developers, which is further broken down into two separate classes: a TouchDevice class and a TouchPoint class. A TouchDevice contains an identification number, or ID, and the current state of the touch. A TouchPoint contains the position and the size of the touch. The touch's timestamp along with a reference to its input device are also provided. [27]

3.1.2.3 Manipulation Events: Gesture Data

Manipulation events represent gestures, or a grouping of individual touch messages. Windows 7 has built in support for multi-touch gestures, such as Tapping, Dragging, and Rotating screen objects. [49] This data can be accessed through one of six different Manipulation Events: ManipulationBoundaryFeedback, ManipulationCompleted, ManipulationDelta, ManipulationInertiaStarting, ManipulationStarted, and ManipulationStarting. Each event has its own set of event

arguments in which settings for rotations and scaling can be toggled on and off. These arguments also include settings for basic 2-D physics such as inertia and bounds detection. For example, a picture can be rotated or scaled and then tossed to another person on the other side of the table. After a short period of time the picture will begin to slow down as it reaches the other side. If the picture does not stop before it collides with an edge of the screen then it will bounce off of the edge and continue to slow down. [27]

The widgets in WPF can subscribe to one or different combinations of these events so that end-users can perform various gestures on these widgets. Furthermore, some of widgets such as a ListView or a ListBox automatically support panning so that this feature does not need to be activated by a developer. This functionality was once provided in the SMART Table SDK and Microsoft Surface SDK, but due to increasing popularity these gestures have been integrated into a mainstream APIs like WPF. [27]

3.1.3 Windows SDK 7.1: Lower-Level Touch Access

Developers can also access touch data by using a lower level Windows API such as the Windows SDK. The Windows SDK mainly supports C++, but it can be accessed through C# using the Interoperability services. [27] [51] Starting with version 7.1, developers can access touch data at the lower levels of the OS and this means that developers have access to the data immediately after it has been processed by WISPTIS. This also means that they do not have to wait for it to be channelled up to an API like WPF. Instead they can access multi-touch data using the lower level callbacks called Windows Messages.

3.1.3.1 Callback Structure: Windows Messages

A Windows message contains bundled up data and is the format used to handle communication between multiple low-level processes in Windows. A Windows Message can also be used to transfer data from an input device to an application. Multi-touch data can be accessed by registering the application's main window to receive a particular message from the OS. The main window listens for these messages using the Window Procedure, or WndProc callback. Once the window has been registered, this function is called repeatedly as touch messages are injected. [1] Figure 3.2, illustrates the process.

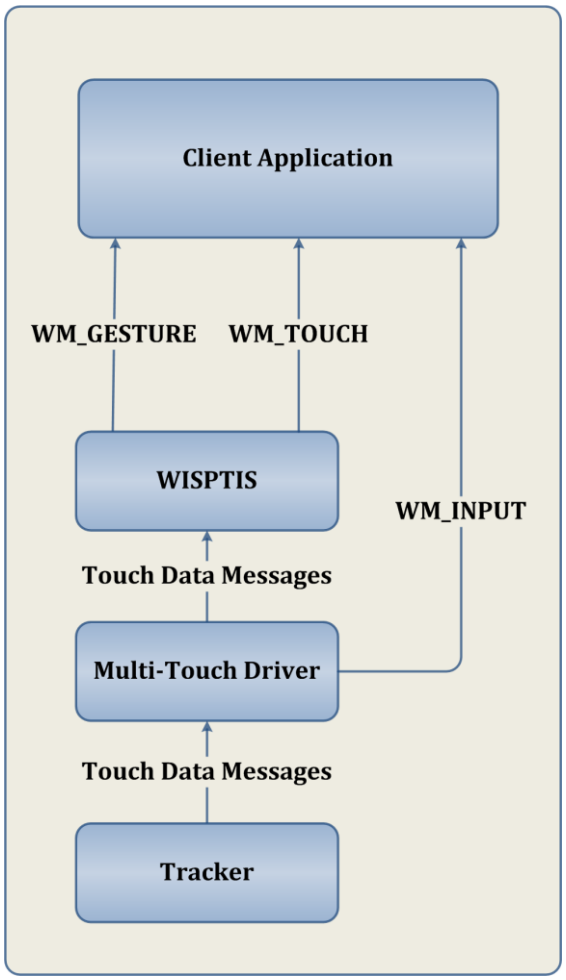


Figure 3.2 WISPTIS and Windows Messages

In WndProc, developers have access to three messages that relate to multi-touch: WM_GESTURE, WM_TOUCH, and WM_INPUT. After processing individual touch messages, WISPTIS will package the refined data into either a WM_TOUCH message or a WM_GESTURE message. This depends on what type of message the main window has subscribed to receive. [1]

3.1.3.2 WM_GESTURE: Gesture Data

A WM_GESTURE message represents a combination of touch messages and gives developers direct access to gesture data. These messages can be unpackaged using the GetGestureInfo function. Gestures are identified by an ID value and can be mapped to a pre-defined macro in the WinUser header file. For example, GID_ZOOM is the macro for pinch and zoom while GID_ROTATE is the macro for a rotation gesture. A WM_GESTURE message is a grouping of individual messages that has been refined into an interaction they do not contain much data. If developers wish two access individual message data, such as position or state, they have two options: WM_TOUCH or WM_INPUT. [1]

3.1.3.3 WM_TOUCH: Processed Touch Message Data

The default message for Windows Touch is WM_TOUCH, which is a direct representation of an injected message that has been processed by WISPTIS. A WM_TOUCH message can be unpackaged using the GetTouchInputInfo function. A set of TOUCHINPUT data structures is returned to the caller through an output parameter. Each TOUCHINPUT structure contains the individual pieces data from an injected message. Developers have access to the following data: x and y coordinates in screen

coordinates, the state, the ID number, the handle to the input device itself, the timestamp for the touch, an extra information field, along with the width and height of the touch. [1]

The x and y coordinates have been adjusted by WISPTIS to fit the device's screen. The touch message's ID and State have also been adjusted. Windows Touch has seven states: Move, Down, Up, In Range (i.e., Hover), Primary (i.e., the first touch that was placed on the screen), No Coalesce, and Palm. [52] If developers attempt to do a direct comparison on the data that was originally injected with the processed data in a WM_TOUCH message, they will notice that the values do not match because of the work that WISPTIS has done to it. If developers require access to the original data then they have option of using a WM_INPUT message.

3.1.3.4 WM_INPUT: Raw Touch Message Data

A WM_INPUT message contains a raw data from the message that was originally injected. WISPTIS is by-passed and developers have access to the original message. These messages can be unpackaged by using either an un-buffered read or a buffered read. An un-buffered read is a standard read that grabs a single message from the device's input buffer. A buffered read retrieves a group of messages from the input buffer. [53]

3.1.4 Windows Driver Kit: Driver-Level Touch Access

The final level of touch access is at the driver level. Windows Messages and Routed events represent indirect access to the contents of the input buffer. However, by using the Windows Driver Kit (WDK), developers have direct access to the input buffer on the server side. Application developers on the client side do not usually have, or need, direct access to the buffer. Instead, this level mainly applies to hardware vendors that write drivers for their multi-touch devices. [54]

3.1.4.1 The Input Buffer

The input buffer is a secure region of memory that has been reserved for input device data. Developers can store device data in this secure region and only the OS will be able to access it. The buffer itself is an array of bytes that contains each value. However, to properly inject data into the OS, developers must declare the format of the data being sent. The WDK supports the HID for USB protocol and developers must load the input buffer according to these specifications. [23] [54]

The input buffer acts as the "Front Door" into the Operating System and it is important that data is sent in the manner that it is declared. Otherwise, the OS will completely reject it to protect itself from potentially harmful data. [23] The next section explains the specifications that developers must follow to inject touch data successfully.

3.2 HID Specifications and Driver Development

HID stands for Human Interface Device and is a common software interface through which all input device data travels after it comes in from a device connected to a USB port. After USB was introduced in 1998, vendors began migrating their input devices over to this format. Devices ranging from mice, keyboards, joysticks and gamepads to barcode readers and printers could be connected to a common hardware port while at the same time sending their data through the same software interface. [23] To send data through the input buffer developers must write two software components: a device driver and an injector. For a multi-touch software system, the tracker contains the injector component and communicates with a driver component.

3.2.1 The Role of an Injector

The injector is responsible for translating the touch point data, after it has been pulled off of the GPU, into the HID format that the OS will recognize. Since the input buffer is an array of bytes, it is the injector's responsibility to break up larger values such as integers into 8-bit chunks. Each chunk is shifted into the buffer using bit-wise operations. Once the buffer is loaded, it is sent off to the Operating System typically using the SetFeature function. [54]

3.2.2 The Role of a Device Driver

The driver component is responsible to handling the actual transfer to the Operating System. A device driver has two primary files: a System (.SYS) executable and an Information File (.INF). The executable contains the actual input buffer while the INF file contains a copy of the HID Report Descriptors for all of the input devices defined inside the driver. [54]

3.2.3 Report Descriptors

Report descriptors are templates that tell the OS what data to expect from a device that is trying to transmit data to it. The OS scans the input buffer and compares its contents against the pre-defined format of the report descriptor. A descriptor itself contains a series of hexadecimal values. Each value has a specific meaning and allows developers to determine the exact order and size of a touch message's payload. [23]

Figure 3.3 provides an example of HID Descriptor for a multi-touch device. The complete version can be found in WDK code sample: EloTouch Driver. [54] A Usage page defines the function of the device. In Figure 3.3, the device is a digitizer which is a device that measures its position in 2D space.

```
0x05, 0x0d,                    // USAGE_PAGE (Digitizers)
0x09, 0x04,                    // USAGE (Touch Screen)
0xa1, 0x01,                    // COLLECTION (Application)
0x85, REPORTID_MTOUCH,         //   REPORT_ID (Touch)
0x09, 0x22,                    //   USAGE (Finger)
0xa1, 0x02,                    //     COLLECTION (Logical)

0x09, 0x42,                    //       USAGE (Tip Switch)
0x15, 0x00,                    //       LOGICAL_MINIMUM (0)
0x25, 0x01,                    //       LOGICAL_MAXIMUM (1)
0x75, 0x01,                    //       REPORT_SIZE (1)
0x95, 0x01,                    //       REPORT_COUNT (1)
0x81, 0x02,                    //       INPUT (Data,Var,Abs)

0x09, 0x32,                    //       USAGE (In Range)
0x81, 0x02,                    //       INPUT (Data,Var,Abs)
0x95, 0x06,                    //       REPORT_COUNT (6)
0x81, 0x03,                    //       INPUT (Cnst,Ary,Abs)
0x75, 0x08,                    //       REPORT_SIZE (8)

0x09, 0x51,                    //       USAGE ( Temp Identifier)
0x95, 0x01,                    //       REPORT_COUNT (1)
0x81, 0x02,                    //       INPUT (Data,Var,Abs)

0x05, 0x01,                    //       USAGE_PAGE (Generic Desk..
0x26, 0xff, 0x0f,              //     LOGICAL_MAXIMUM (4095)
0x75, 0x10,                    //       REPORT_SIZE (16)
0x55, 0x0e,                    //       UNIT_EXPONENT (-2)
0x65, 0x33,                    //       UNIT (Inch,EngLinear)

0x09, 0x30,                    //       USAGE (X)
0x35, 0x00,                    //       PHYSICAL_MINIMUM (0)
0x46, 0xb5, 0x04,              //       PHYSICAL_MAXIMUM (1205)
0x81, 0x02,                    //       INPUT (Data,Var,Abs)
0x46, 0x8a, 0x03,              //       PHYSICAL_MAXIMUM (906)

0x09, 0x31,                    //       USAGE (Y)
0x81, 0x02,                    //       INPUT (Data,Var,Abs)
0xc0,                          //    END_COLLECTION
```

Figure 3.3 An HID Descriptor Sample

A Usage item defines a specific function for the data. In Figure 3.3, Touch Screen, Finger, X and Y give meaning to the sections of data. Usage items also exist for mouse, keyboard and pen so that different descriptors can be defined in the same report. A Collection marks a common grouping of data that performs one function. [23] Developers also have access to all of the usage values for various input devices. [56] Furthermore, Microsoft has provided developers with a specialized driver development environment, which is part of the WDK, to aid them in creating, compiling and verifying drivers. [54] The WDK also ships with a series of code samples and documentation to

help developers successfully structure and format of their HID Reports. Microsoft has also provided a series of online documents to walk developers through the tedious process of creating multi-touch drivers for the Windows OS. [57] [58] [59] [60] [61]

3.3 Summary

This chapter explained the Windows 7 Touch system in more depth and elaborated on how touch data is processed and how it can be accessed through the callbacks of the Windows Touch APIs. This chapter also introduced the underlying HID protocol for this system. Driver development is quite challenging and Chapter 5 will discuss some of the technical challenges that were faced when trying to modify an existing driver so that it would be able to carry a larger payload.

Chapter Four: The TUIO Protocol

The Tangible User Interface (TUIO) protocol is a protocol for multi-touch data transfer and was first introduced in 2005. [24] It was originally designed to handle input from tangible objects being tracked on a tabletop. Identification tags, also known as markers, were placed on the bottom of each object. A specialized tracker, reacTIVision, was created to track these objects and then send the object's data to client applications using a network socket. [62] The transfer component was extracted into its own component and the result was TUIO.

TUIO was built on top of another protocol called Open Sound Control (OSC). OSC was specifically designed to transfer data generated from multiple musical devices. This protocol was built on top of the UDP network protocol. UDP was chosen at the time because it provided faster data transfer and was less complex than TCP. The TUIO_CPP API contains an OSC buffer which handles the actual loading and transfer of touch data. The OSC API ensures that each data field is shifted into the buffer and the data is transmitted through the socket when a call is made in the TUIO_CPP API. [24] [25]

If the raw input buffer for HID is considered to be the front door into the OS, then sending touch data through a network socket would be considered sending data through an open window. This chapter discusses TUIO and how it differs from the HID protocol introduced in the previous chapter. Since its inception in 2005, TUIO has changed. The first two sections of this chapter discuss versions 1.0 and 1.1 of the protocol. This is followed by an overview of the supporting software that is available for TUIO along with some concluding remarks.

4.1 TUIO 1.0: The First Version of the Protocol

While HID was mainly upgraded to include finger tracking, TUIO was originally designed for object tracking. However, it was also expanded to include finger tracking. The TUIO website contains support documentation to aid developers in using TUIO. [63] TUIO supports a wide range of multi-touch devices from tabletops to vertical displays. The first version of the protocol defined three types of profiles for devices: 2D, 2.5D and 3D interactive surfaces. Each type contains two profiles: Object and Cursor.

4.1.1 Obj: Tangible Object Profile

The object profile is meant for object and tag tracking. Using this profile allows developers to send the following data: the object's ID, the ID of the tag marker, the object's position, and the angle. This profile also supports additional fields such as the speed and direction the touch is moving in, and finally the rate at which the object is accelerating. These fields can be applied to an object that is either being moved around or is being rotated on the display's surface. [64]

4.1.2 Cur: Touch Point Profile

The cursor profile is for finger tracking and simulating touch with a mouse device. However, this profile assumes that the tracker has already refined the touch down to a single point. This refinement was illustrated in Figure 1.3 of Chapter 1 and represents the final refinement stage before the touch is sent to the OS. The cursor profile contains the same information with the exception of the angle values since the touch has been refined past the bounding box to a point. [64]

4.2 TUIO 1.1: The Next Version of the Protocol

The next version of the TUIO protocol was expanded to provide developers with more data for touch interactions. The protocol was expanded in version 1.1 to include a new profile that contained additional touch data coming from parts of a hand. Specifically, the new profile includes additional fields for the area surrounding a touch point in Figure 1.3. [65]

4.2.1 Blb: The New Profile

The Blob profile was created for blob tracking. It provides developers with the same amount of data in the Object profile, but also provides additional data for the dimensions of the touch's bounding box. These values are: width, height and area of a touch. [65] Implementation for this new version of the protocol has recently emerged on TUIO's website complete with the Blob profile.

4.3 TUIO Software: Protocol Implementations

In addition to providing implementation for their multi-touch transfer protocol, the TUIO website also provides a wide range of software to support it. TUIO software components are very similar to the model of a multi-touch software system in Figure 1.1. Implementations of the TUIO protocol are broken up into individual software components. These components include: Trackers for image processing, clients for sending and retrieving touch data, APIs for retrieving touch data and exposing it to developers in custom-built callbacks, and finally a series of conversion bridges so that the TUIO callbacks can support conversion to other formats and so that other formats can be converted to the TUIO format. [66]

The TUIO clients are very flexible and support a wide range of development languages such as C++, C#, Java, and Flash. Support for these languages is provided in individual client APIs such as TUIO_CPP for C++ and TUIO_JAVA for Java. Each client provides a TUIO Server for sending touch data and a TUIO Listener for receiving data. Similar to WPF 4.0, data is received through callbacks in the form of routed events. However, these routed events are structurally very different than the WndProc callback in the Windows SDK or the routed touch events in WPF 4.0.

TUIO's main strength lies in cross-platform use. It does not restrict developers to the Windows OS and allows them port applications to devices running other operating systems. It also represents an early attempt at breaking up a multi-touch software system into individual components. If one component is not working properly, or is performing poorly, it can be swapped with another component from the site. Furthermore, the TUIO website is constantly updated with newly available software giving developers more selections to choose from. Multi-touch software systems that support TUIO are a lot like Lego bricks. [67] Once developers have chosen a device and the OS platform they can construct a system by connecting the interlocking components together after they have downloaded them from the website. [66]

4.4 Summary

This chapter provided details surrounding the TUIO protocol and discussed its various states of change. This was followed by a description of the supporting software available on TUIO's website so that developers are able to construct their own multi-touch software systems. This chapter brings a close to the background material on both protocols and moves towards traditional research methods to evaluate their performance.

Chapter Five: SPE Methodologies

As mentioned in Chapter 1, there is very little work previously published in the area of multi-touch system performance and no work has been published describing which of the two multi-touch protocols, introduced in Chapters 3 and 4, is faster. To answer this question, a performance evaluation was conducted in order to compare the average latencies yielded by both protocols.

In order to set up a multi-touch system performance evaluation, it was necessary to consult the methodologies used in evaluating other client/server systems. This chapter explains the general methods that have been used to evaluate the system performance of client/server software systems. It begins with an overview of Software Quality and Software Performance Engineering. This is followed by an overview of Benchmarking, Performance Management Tools, and some concluding remarks.

5.1 Software Quality

It was mentioned in Chapter One that multi-touch software systems for Windows 7 have evolved into modular systems that employ a layered architecture. Much research has already been conducted to determine efficient ways of evaluating how well these systems perform in the real world, and how they can be improved upon as time goes on. Software quality has several different elements which include: Function, Performance, Presentation, and Maintenance. [12]

Software Testing, which ensures functionality, is concerned with data being processed *correctly* by the given system. Software Performance, which ensures efficiency, is concerned with data being processed *quickly* by the given system. Usability Testing, which ensures ease of use, is concerned with producing a system directly for the

end-users of the given system. Software Maintenance, which ensures ease of transition, is concerned with the given system's ability to survive in diverse and often highly competitive markets as it undergoes changes based on fluid requirements. These elements all contribute to the overall quality and are crucial in achieving a successful system. [12]

5.2 Software Performance Engineering

Software Performance Engineering (SPE) is the process of creating a system that meets specific performance goals throughout the life cycle of a software project. SPE uses quantitative analysis methods to identify and eliminate performance problems as early as possible in the software's life cycle. These methods aid developers in adopting stronger system designs throughout the life cycle as the system begins to take shape. The primary goal of SPE is to achieve better performing software systems, through continuous testing, analysis, and fine-tuning based on specific targets. [12] [13]

SPE contains a variety of different concepts such as workload, response time, throughput, resource utilization, and resource service time. Resource utilization and resource service time are concepts that monitor system components externally and report on how the system behaves based on its usage of the available resources. Workload, response time and throughput are all concepts that report on a system's speed, by monitoring it internally or externally, to report on its ability to process data. A workload is a pre-defined set of data that the intended system is given to process. Workloads can be based on anything from device input from user interactions to formal requests for data in large-scale enterprise systems. Response time, also known as latency, is the time it takes to process a given workload. Latency is mainly used to find delays, or bottlenecks, in the

system and they typically depend on the size of the workload. Throughput is the amount of data that can be processed by the system within a given timeframe. [12]

5.3 Benchmarking

Benchmarking is a term used to describe work and is the process of setting up a software system for stress testing. The goal of a benchmark is to determine what components in the system require further tuning. Benchmarks are controlled tests that contain a pre-defined set of data, or workload, that system will process. Benchmarks monitor the system as it processes the data in the workload. They take measurements at key points along the system's processing pipeline to determine stress-points. The observed behavior is either displayed at run-time, while the system is processing, or stored for later analysis to minimize side effects on the measurements. Benchmarks can also include functionality that simulates user input. [12]

A benchmark harness, or benchmark suite, is a grouping of benchmarks that, when combined, can be used to help automate the benchmarking process so that multiple workloads can be lined up for the system to process in succession. A benchmark harness is useful in that developers have the flexibility to organize the order of the benchmarks to see how well the system stands up against different sized workloads. Benchmarks can be ordered based on a slow and steady increase in the amount of stress the system will be under or they can be ordered based on rapid changes in stress levels. This way the system can be stress-tested in a variety of different scenarios which can prove necessary for large-scale systems that must be flexible to changing conditions. [12] [13]

5.4 Performance Management Tools

If a benchmark harness is well-designed, and if there is a sufficient need for its features in other software systems, they can pave the way for performance management tools. These tools are responsible for observing software systems and reporting the results back to developers. They can be broken down into four categories: Cataloging, Monitoring, Profiling, and Historical tools. Cataloging tools record information based on configurations and parameters in the system, such as memory management and network settings. Profiling tools analyze and display the internal code structure of a component and provides a window into the inner workings of a component. [12]

Historical tools analyze data that were previously logged either by a benchmark harness or another tool. Data is stored and indexed by date and time. Calculations are performed after it has been stored. Monitoring tools can track the performance of the system as it executes in real-time or store the observed behaviour for future analysis. There are two ways of monitoring a component: internal monitoring by taking measurements inside and external monitoring by taking measurements before and after the component processes the workload. Unfortunately, monitoring tools themselves are known to slow down performance and can result in inaccurate measurements. Due to this reality, many developers restrict monitors to a benchmark harness where each observation is logged by the system for future analysis in a historical tool. [12]

The performance concepts presented in this section have been applied to numerous types of computing, such as network computing and enterprise information processing. These concepts serve as inspiration for the design and implementation of a benchmark harness for a multi-touch software system.

5.5 Summary

This chapter introduced the basic concept of Software Quality and how the methodologies in SPE can contribute to producing high-performance software systems. Concepts such as Benchmarking are useful techniques when attempting to evaluate the efficiency of a given software system. Performance Management Tools can be used to analyze and present results produced by these evaluations.

The concepts in Chapters 3 and 4 were presented to explain how both protocols are similar used to evaluate the performance of the HID and TUIO protocols used in a multi-touch software system. The past three chapters outlined background material relating to the most commonly used multi-touch protocols. The following chapters outline the process of applying SPE methods to answer the central research question.

Chapter Six: Performance Evaluation

This chapter provides an in-depth analysis of the process followed to set up and run a multi-touch protocol performance evaluation. The evaluation's conceptual design is first presented. This is followed by the technical details surrounding the implementation of a multi-touch benchmark harness, followed by some concluding remarks.

6.1 Design

In order to set up a multi-touch benchmark harness that simulates touch input, the conditions for each benchmark must be defined before it is implemented and activated. This section explains the conceptual design of the evaluation and how SPE methods influenced its design. Figure 6.1 illustrates the proposed design for the evaluation.

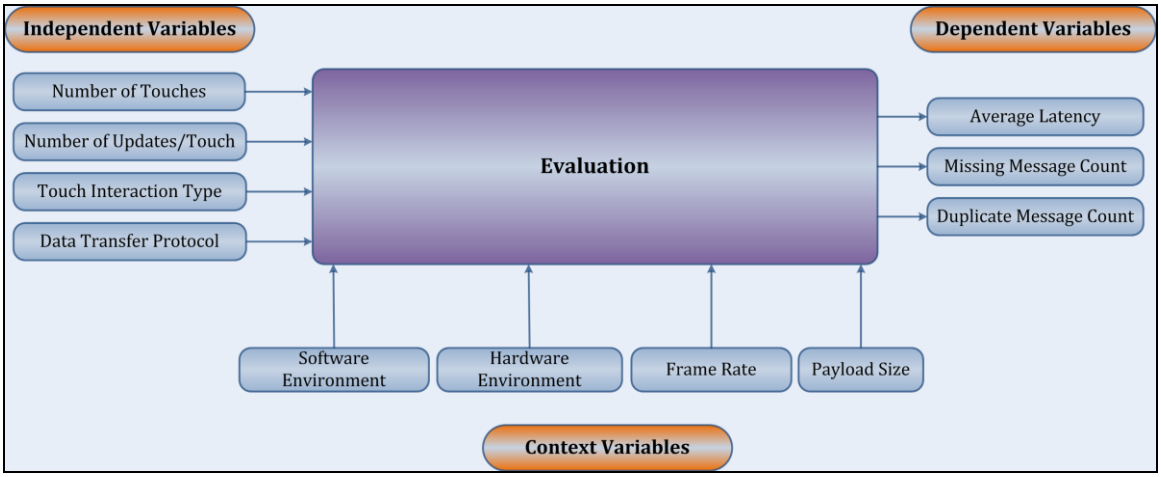


Figure 6.1 Designing the Performance Evaluation

The following sub-sections explain the evaluation in detail by breaking the surrounding variables down into three sets: Context, Dependent, and Independent. Each variable is related to the performance concepts presented in the previous chapter and motivation for why they were selected for their particular categories.

6.1.1 Context Variables

These variables have an impact on the results but remain unchanged throughout the course of the evaluation and four were identified: Software Environment, Hardware Environment, Camera Frame Rate, and Payload Size. The first two variables are concerned with the overall environment that the system will be deployed on. From a performance perspective it is important to know the resources that are available to the software system before it is activated. [12]

6.1.1.1 Software Environment

It is important to know the capabilities of the software, such as the Operating System version and the total number of processes executing in the background, before conducting a performance evaluation. [12] Windows 7 Enterprise Edition was selected as the Operating System because of its built-in multi-touch capabilities. Referring to Figure 1.1, the individual components of a multi-touch software system, the tracker, the Windows 7 Touch API, and the client application all remained unchanged during the experiment. The only exception in this system was the transfer protocol because it was the layer in the system that was to be evaluated.

6.1.1.2 Hardware Environment

In some evaluations it is normal to test various sized workloads on different brands of hardware such as hard disk drives determine which brand stores and retrieves data faster. [12] This situation existed in Muller's evaluation when different projector products were compared and in Montag *et. al.'s* evaluation when different camera products were compared. [6] [31]

However, since the focus of this evaluation is on software, the conditions in the hardware environment were not changed. The computer selected was a Mac Mini with a 2.53 GHz Intel Core 2 Duo processor, and 4.00 GB RAM. The multi-touch device and intended deployment platform for the proposed multi-touch software system is the SMART Table. [69] This computer was selected for the SMART Table because of its compact size making it easily portable, and also because of its ability to run both Windows and Mac OS platforms. It was removed from the device for the evaluation because touch input was simulated, thus eliminating the need for participants to generate the touches. A widescreen monitor capable of achieving a maximum resolution of 1920x1200 was selected as the main display. This resolution was chosen because newer digital tabletops, such as the EvoluceOne, are supporting higher screen resolutions. [70] Older tables, such as the SMART Table, only support up to 1024x768. [69]

6.1.1.3 Camera Frame Rate

In the tracker, the camera frame rate was set at 60 frames per second (fps). From the system's perspective, at the transport layer, the frame rate is merely the rate at which data is sent to the OS. It was chosen as a context variable to minimize the total amount of data that needed to be analyzed after the benchmark harness was deactivated. The amount of logged data that needed to be analyzed became a significant problem, and will be explained further in the results section of this chapter. In Muller's evaluation, infrared cameras at that time were only capable of sending 30 fps, but newer products are capable of achieving higher frame rates. [6]

6.1.1.4 HID Payload Size

The final context variable for this experiment is payload size. The payload sizes for HID messages remained unchanged due to technical challenges and limitations in the Windows 7 Touch system. The issues surrounding this decision are explained in the implementation section of this chapter.

6.1.2 Dependent Variables

The dependent variables represent the outcome measures, or performance calculations, of the evaluation and represent the calculations a performance management tool would conduct on logged data. [12] Three variables were chosen: The Average Latency, The Number of Missing Messages, and The Number of Duplicate Messages.

6.1.2.1 Average Latency

Latency was chosen as the primary performance calculation. It represents the amount of delay incurred when different sized workloads are sent through the transport layer of a multi-touch software system. The average latency variable represents the mean of all the recorded latencies observed by the benchmark harness. Average latency was selected as a dependent variable to help break down the data since a single benchmark could contain hundreds or even thousands of touch messages.

6.1.2.2 Missing Message Count

The next dependent variable that was chosen was the total number of messages that might go missing during the course of an interaction. This variable was chosen to see what impact larger workloads have on data integrity. If too many touch messages go missing between transmission and arrival it could mean that there are severe limitations in the transfer protocol and it can also lead to poor user experiences on multi-touch

devices. End-users are unlikely to notice small numbers of messages go missing, but they are highly likely to notice larger numbers go missing as visual feedback appears choppy and their interaction loses its natural smoothness. Furthermore, users will notice if edge messages at the beginning (down) or the end (up) of the interaction fail to appear.

6.1.2.3 Duplicate Message Count

The total number of duplicated messages was selected as a dependent variable to determine possible external causes for high latencies. These messages are usually sent to cover up irregularities, such as data loss, in transfer protocols. The number of duplicates is important to calculate because, depending on the original workload and on the protocol's limitations, it can dramatically increase the size of the original workload thus creating an unintentional bottleneck without the developer's knowledge. [12]

6.1.3 Independent Variables

An effective benchmark harness is one that is made up of well-chosen benchmarks that have a variety of different workloads. The purpose of this is to ensure that no benchmark does the same thing repeatedly and does only one thing. [12] To avoid duplicate benchmarks, the conditions in each one changes throughout the course of the evaluation and four were selected: the Transfer Protocol, the Number of Concurrent Touches, the Number of Update Messages per Touch, and Types of Touch Interactions.

6.1.3.1 Data Transfer Protocol

The transfer protocol variable represents the main point of comparison for this evaluation. The HID for USB and TUIO multi-touch protocols were selected for the comparison. From the system's perspective, at the transport layer, there are only messages of multi-touch data and at this stage of the pipeline this is the only type of data

the system deals with. Touch messages are divided up into groupings and when combined together they form the basis for how multi-touch interactions are interpreted at this layer.

Down messages are used to notify the system when a touch has made contact with the screen. Update messages are used to notify the system that a touch is still making contact. Up messages are used to notify the system when a touch is no longer making contact. In order to construct multi-touch benchmarks that simulate touch input, it is necessary to set the data inside each message before it is sent to the OS.

6.1.3.2 Touch Interaction

The interaction variable represents different types of touch interactions. They are divided into two primitive categories: Stationary and Moving. Four interactions were chosen: Hold, Down, Right, and Diagonal. These interactions are illustrated in Figure 6.2.

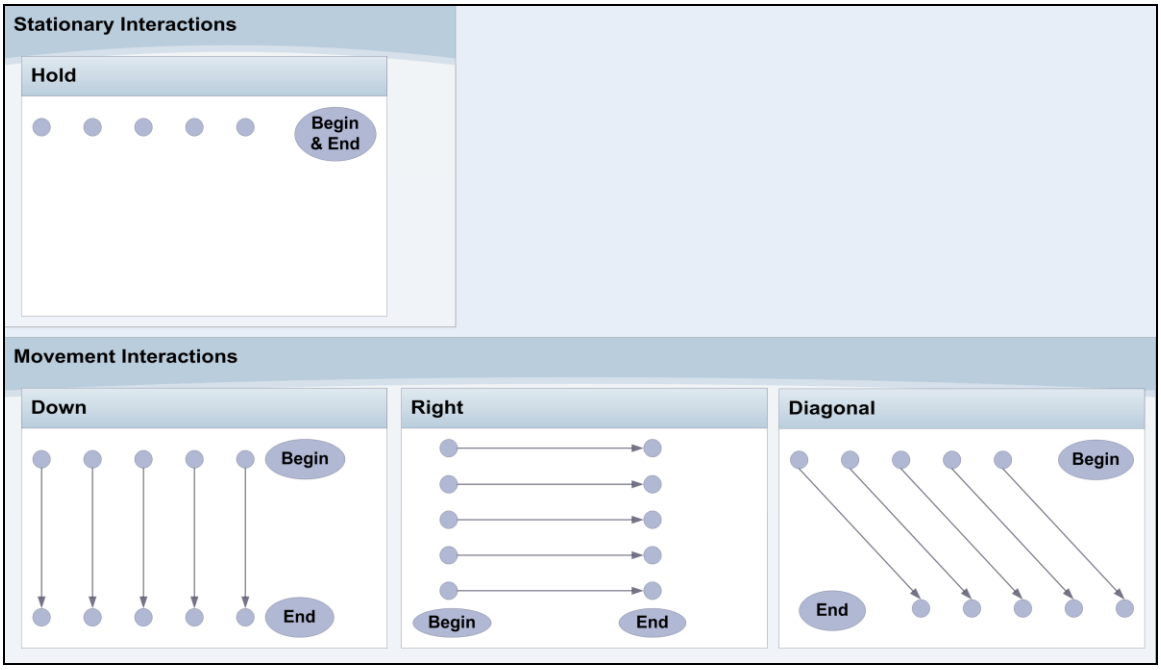


Figure 6.2 Types of Touch Interactions and their Categories

Hold represents stationary interactions. Touches are placed at the top of the screen and remain where they are until they are removed. Down represents interactions that are moving vertically on a touch screen. Touches are placed at the top of a screen and move downward towards the bottom where they are released. Right represents interactions that are moving horizontally on a touch screen. Touches are placed at the left-hand edge of the screen and move towards the right-hand edge where they are released. Diagonal represents the interactions that are moving in a skewed pattern. Touches are placed at the top of the screen and move downwards in a skewed pattern until they are released at the bottom edge. The basic idea for these interactions was inspired by the paper templates used on Buxton *et al.'s* tablet device. [16]

6.1.3.3 Number of Update Messages per Touch Interaction

The number of updates per touch represents the duration that each touch would be making contact with a touch screen. Touches making contact for longer periods of time contain larger numbers of update messages than touches that are making contact for shorter periods. The values chosen for this variable were: 10, 50, 100, and 200 updates per touch. Since the frame rate is 60 fps, frames are sent through the system every $1/60^{th}$ of a second. Therefore, touches are making contact with the screen for approximately 1 second and during that period of time 60 frames are sent to the system for processing. Approximate durations can be calculated using the following equation: $1/60 = n/m$, where n is the approximate duration (in seconds) touches are making contact with a device's screen and m is the number of updates per touch. When the update values are used in place of m, n can be calculated ($n = m \times (1/60)$) yielding the following durations: ~0.17s (10 updates), ~0.83s (50 updates), ~1.67s (100 updates), and ~3.33s (200 updates).

The selection of these values was also influenced by the screen resolution of the display when deciding the length of each movement interaction. Since the test hardware is capable of achieving a maximum resolution of 1920x1200 pixels, higher numbers of updates can be chosen. For example, a value of 200 for Down means each touch will use approximately 80% of screen's height. For Right, approximately 60% of the screen's width is utilized. A value of 10 updates per touch for Down uses approximately 4% of the screen's height. For Right, approximately 3% of the screen's width is used. For lower resolutions, these update values would need to change because touches from 100 to 200 updates per touch would not fall within the screen's boundaries.

The spacing between updated coordinates also influenced the selection of these values. Since the screen resolution is set at 1920x1200, the changing coordinates for each update message in a benchmark can be evenly spaced out by 5 screen pixels. For example, touches are placed at the top of the screen for Down. Each y coordinate can be increased by a value of 5 pixels for each update message. For Right, each x coordinate can be increased by a value of 5 pixels for each update message.

6.1.3.4 Number of Touches

The number of touches represents number of concurrent touches that the system has to contend with. A total of eight values were selected for this variable: 1, 2, 5, 10, 20, 40, 80, and 100 concurrent touches. The number of touches and the number of updates per touch, when combined, form the basis for constructing a workload.

Choosing the workloads for a benchmark is never an easy task. They must be realistic and also capable of locating a system's stress points. [12] The values chosen for number of touches are broken down into two scenarios: ones that mirror reality and ones

that are exploratory. 1 touch represents single user single touch interactions. 2 to 10 touches can represent single-user multi-touch interactions. 20 to 40 touches can represent multi-user multi-touch interactions. These values represent *realistic* scenarios, where one to four people could be situated at an edge of a SMART Table. 80 to 100 touches were chosen to represent *stressful* scenarios that may or may not mirror reality, but are necessary to explore the system's capabilities by putting it under duress. In practice, such workloads could reflect scenarios involving physically large displays such as those that might be found in collaborative group work settings in the future.

6.1.4 Constructing a Multi-Touch Performance Model

With the variables and their chosen values defined, they can be combined together to form a benchmark. A benchmark harness can be used by developers to put the benchmarks in a sequential order for execution. [12] The following sections describe the process used to create multi-touch benchmarks and how to combine them together to form a benchmark harness.

6.1.4.1 Constructing a Multi-Touch Benchmark

Three elements are required in order to construct a multi-touch benchmark: A Workload, a Touch Interaction, and a Protocol. The first step in this process is to calculate the workload sizes using the independent variables number of touches and number of updates per touch. The touch values (1, 2, 5, 10, 20, 40, 80, and 100) are combined with the update values (10, 50, 100, and 200).

For example, 1 touch and 10 updates for this interaction results in 12 messages for the system to process: 1 down message, 10 update messages, and 1 up message. For 2 touches and 10 updates per touch, there are a total of 24 messages: 2 down messages (1

per touch), 20 update messages (10 per touch), and 2 up messages (1 per touch). Workloads can be calculated using this formula: *(# of Touches × 2) + (# of Updates per Touch × # of Touches)*, where 2 is the number of notify messages sets (1 down message + 1 up message per touch). Appendix A contains a table with all 32 workload calculations.

6.1.4.2 Constructing a Multi-Touch Benchmark Harness

Once the workloads have been calculated the touch interaction and protocol variables are added to the benchmark. However, it is important that developers make clear distinctions between benchmarks so that there are no redundant ones. For example, a benchmark with a 12 message workload for a downward moving interaction intended for HID is not the same as a benchmark with a 12 message workload for downward moving interaction intended for TUIO because the protocol has changed from HID to TUIO. Similarly, a 12 message HID benchmark for a stationary interaction is not the same as a 12 message HID benchmark for a downward moving interaction because the interaction has changed from Hold to Down.

After all of the benchmarks have been defined, they can be combined to form a benchmark harness, which is also referred to as a benchmark suite during the design phase when the benchmark parameters are defined. A common practise for achieving the conceptual design of a benchmark harness is to construct a performance model. A performance model is an effective method that developers can use to visualize and order their performance tests. [12] Table 6.1 presents the multi-touch performance model for the proposed evaluation.

Combining all of the benchmarks together results in 256 in total: 128 for HID and 128 for TUIO. Both protocols contain 4 interactions each, with 32 benchmarks in each

interaction: 8 concurrent touch combinations · 4 sets of updates per touch. Workloads are continually increased, which allows for systematic load-testing on each interaction. [12] Each set is repeated throughout the evaluation, but they are not identical benchmarks because the interaction and protocol variables change. Workloads only repeat themselves for other interactions and interactions only repeat themselves for other protocols.

Table 6.1 A Performance Model for the Transport Layer of a Multi-Touch System

Purpose		**Transport Layer: Multi-Touch Protocol Comparison**							
Env.		SMART Table Software System, Windows 7 Enterprise Edition SP1, Mac Mini 2.53 GHz, 4GB RAM, 60fps Refresh, Tabletop Absent							
Workload Variables		HID Protocol				TUIO Protocol			
		Hold	Down	Right	Diag.	Hold	Down	Right	Diag.
10	1	12	12	12	12	12	12	12	12
	2	24	24	24	24	24	24	24	24
	5	60	60	60	60	60	60	60	60
	10	120	120	120	120	120	120	120	120
	20	240	240	240	240	240	240	240	240
	40	480	480	480	480	480	480	480	480
	80	960	960	960	960	960	960	960	960
	100	1,200	1,200	1,200	1,200	1,200	1,200	1,200	1,200
50	1	52	52	52	52	52	52	52	52
	2	104	104	104	104	104	104	104	104
	5	260	260	260	260	260	260	260	260
	10	520	520	520	520	520	520	520	520
	20	1,040	1,040	1,040	1,040	1,040	1,040	1,040	1,040
	40	2,080	2,080	2,080	2,080	2,080	2,080	2,080	2,080

	80	4,160	4,160	4,160	4,160	4,160	4,160	4,160	4,160
	100	5,200	5,200	5,200	5,200	5,200	5,200	5,200	5,200
100	1	102	102	102	102	102	102	102	102
	2	204	204	204	204	204	204	204	204
	5	510	510	510	510	510	510	510	510
	10	1,020	1,020	1,020	1,020	1,020	1,020	1,020	1,020
	20	2,040	2,040	2,040	2,040	2,040	2,040	2,040	2,040
	40	4,080	4,080	4,080	4,080	4,080	4,080	4,080	4,080
	80	8,160	8,160	8,160	8,160	8,160	8,160	8,160	8,160
	100	10,200	10,200	10,200	10,200	10,200	10,200	10,200	10,200
200	1	202	202	202	202	202	202	202	202
	2	404	404	404	404	404	404	404	404
	5	1,010	1,010	1,010	1,010	1,010	1,010	1,010	1,010
	10	2,020	2,020	2,020	2,020	2,020	2,020	2,020	2,020
	20	4,040	4,040	4,040	4,040	4,040	4,040	4,040	4,040
	40	8,080	8,080	8,080	8,080	8,080	8,080	8,080	8,080
	80	16,160	16,160	16,160	16,160	16,160	16,160	16,160	16,160
	100	20,200	20,200	20,200	20,200	20,200	20,200	20,200	20,200
Measures		Average Latency, Missing Message Count, Duplicate Message Count							

6.2 Implementation

To achieve the proposed design, a benchmark harness was implemented for a multi-touch system. However, as noted in the Muller and Montag *et al.* performance evaluations, a multi-touch software system had to be constructed out of existing software components. [6] [31] Figure 6.3 illustrates the components of the system, and the points along the pipeline where the benchmark harness will take measurements.

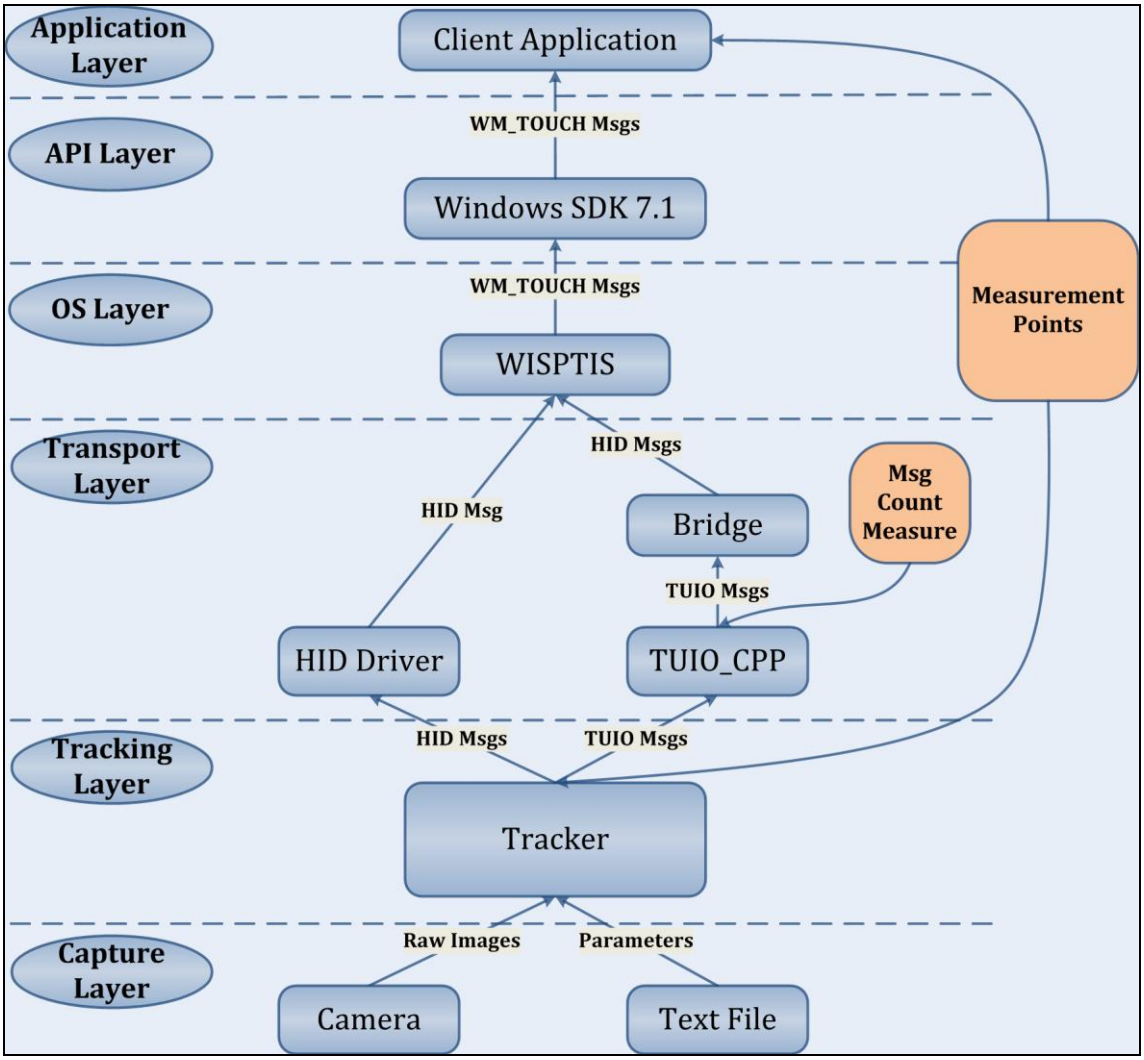


Figure 6.3 The Modified Multi-Touch Software System

The system contained five primary components. A multi-touch tracker begins the pipeline. In the transport layer, a Windows 7 multi-touch driver was selected to represent HID. [4] TUIO_CPP was selected to represent TUIO 1.1 for two reasons. First, looking at the TUIO repository it was noted that TUIO_CPP had the highest number of downloads indicating that it was the most popular TUIO API. Second, during the implementation stage of this research, TUIO_CPP was the only API that was updated to include the features of TUIO 1.1. The TUIO_AS3 API was updated months later, but APIs such as

TUIO_CSHARP remained unchanged. [66] A Windows SDK code sample, MTScratchpad, acted as the client application. [51] A prototype multi-touch benchmark harness was constructed for this system. The following sections provide an in-depth look at the process followed when attempting to implement this prototype.

The transfer protocols are the center of this investigation and form the only difference in the processing pipeline where data could travel in one of two separate avenues. Therefore, benchmark harness functionality had to be placed in the surrounding components of the system. This functionality is divided up into four areas: Touch Input Simulation and Automation, Monitoring and Logging, Protocol Bridging, and Payload Modification. This functionality is spread across three components of the system in Figure 6.3: the Tracker, the Client Application, and the HID Driver.

6.2.1 The Tracker

The multi-touch tracker was modified to send data through an HID driver or through a TUIO socket. The transfer mode could be toggled between HID and TUIO in the tracker's UI. The tracker was then modified to simulate multi-touch input.

6.2.1.1 Touch Input Simulation

The image processor inside of the tracker's main event loop was by-passed as both pipelines were able to utilize the same image processor. If a camera was connected to the computer, the system assumed it was in the actual deployment environment. Otherwise, it assumed it was in a testing environment. Since the remaining components were not receiving data from the image processor, they had to receive data from another input source. This process involved two steps: creating a list of input parameters from the independent variables and then constructing individual benchmarks by loading them into

a buffer in memory to ensure that file access did not impact the benchmark's results. After the loading process was complete, the data could be sent to the OS at the pre-defined camera frame rate. This separation between load-time and execution-time was done to avoid latencies resulting from repeated disk access.

A text file contained all of the input parameters and each parameter was taken from the following independent variables: number of touches, number of updates per touch, and touch interaction. For example, 1 touch with 10 updates per touch with that touch being stationary, the parameters would be 1 (touch), 10 (updates), and 0 (where 0 = Hold, 1 = Down, 2 = Right, and 3 = Diagonal). For 2 touches with 10 updates per touch where both touches are moving downward, the parameters would be 2, 10, and 1.

In order to achieve movement, the position coordinates of each payload was modified before they were loaded into memory. The path can be controlled via the x and y coordinates. For example, to achieve a vertically moving interaction, only the y coordinates need to be changed and the x coordinates remain the same. To achieve a horizontally moving interaction, only the x coordinates need to change and the y coordinates remain the same. To achieve a diagonally moving interaction, both the x and y coordinates need to change.

Based on the configuration settings, the tracker reads each parameter and constructs the corresponding workload in memory. Since the transport layer is being evaluated this means constructing each message. However, there are significant differences in the format of each protocol and this leads to workload distortion. [12]

The TUIO specifications allow three main sets of messages: Down, Update, and Up. [64] [65] However, the HID specifications allow up to five sets of messages: Hover

In-Range, Down, Update, Up, and Hover Out-Of-Range. [38] [46] Developers are given the option of using all five or as little as three (Down, Update, Up) of them. For example, if four sets are used to make up an interaction for HID, then an intended workload of 12 messages (1 down, 10 updates, 1 up) becomes 13 messages (1 down, 10 updates, 1 removing, 1 removed). The workload remains at 12 messages for TUIO but turns into 13 for HID. For an intended workload of 24 messages (2 down, 20 updates-10 per touch, and 2 up) becomes 26 messages (2 down, 20 updates-10 per touch, 2 removing, and 2 removed) for HID while it remains at 24 for TUIO.

These differences in format can distort the intended workloads outlined in a developer's performance model. However, this situation is inevitable and unavoidable when comparing two protocols because developers lose control of their workloads as they are at the mercy of someone else's code in the protocol's API. This code could contain error handling and other safety mechanisms to smooth out irregularities in the protocol. [12] While this functionality is necessary, it can further distort a developer's intended workload. This problem is discussed in further detail throughout this chapter. However, it is still important to make intended workloads of equal size before they are sent through either protocol so that workload distortion can be minimized on the developer's part [12]. Therefore, a different data structure common to both senders was required. Resolving this issue also proved necessary for other reasons.

Simulating touch input inside both senders led to large amounts of unnecessary code duplication. This meant for each interaction, the path manipulation and buffer storage code had to be written for HID and then re-written for TUIO because TUIO_CPP

and the WDK are completely different APIs. This is a cumbersome approach because it makes the addition of new interactions in the future very difficult.

The second reason was that sleep() statements were required to slow HID down because messages were being passed to the injector far too fast leading to massive data loss in the client. The reason for this was due to poor timing and messages were colliding in memory. Messages were stored for the injector to send, but before it could do so they were being replaced by the next ones in the sequence. Furthermore, the presence of sleep statements at any stage in the pipeline poses a problem for system performance because they can increase the overall latency for the entire system. The presence of sleep statements prior to transfer can also introduce a bias against both protocols.

This problem was addressed by Muller when attempting to take measurements inside the Touchlib tracker. [6] Muller's solution was to treat them as a *bug* and remove them from Touchlib. Sleep statements were used in this evaluation's tracker, but they were used at the end of each touch interaction and not between individual message transmissions. This was done for two reasons. First, it helped ensure that the pipeline was clear before the start of each interaction so that messages from the previous interaction would not interfere with messages from the next interaction. Second, it opened up the possibility of simulating the time it would typically take users to release touches from a device's display to the time it would take to re-initiate contact. The sleep value was defined as a variable so that developers could specify their own sleep durations. This value was specified in the input parameters text file.

To address these timing problems, the tracker's main event loop was reused to handle the delicate timing of the transfer. The code that handled transfer through HID and

TUIO was also reused. This was beneficial because it allowed for new types of interactions to be defined without the creation of so much redundant code. Furthermore, the image processor could be by-passed without being disabled at run-time. A camera could still be hooked up at any time because the image processor was merely dormant. The bounding box structures that emerge from the image processor were used in place of messages. This allowed for same-sized workloads to be passed to either sender.

6.2.1.2 Touch Input Automation

The buffer loading process involved lining the workloads up for sequential transfer. This allowed for a certain degree of automation to occur in the system so that the benchmark harness could monitor gradual changes in the system's environment, which is important when trying to detect bottlenecks in the system. [12]

When the tracker was ready to send data to the OS, which is the next frame in the event loop, the buffer could be indexed to accurately time the transfer. This proved tricky because a process that carefully indexed the buffer had to be defined to mimic the way the tracker would normally send processed data from the image processor. This indexing process involved breaking up the entire buffer into touch interactions and then further breaking each interaction up so that touch messages would be sent based on the original parameters outlined in the text file.

If a raw image was processed and 1 touch was extracted, then that touch would be sent. If an image was processed and 5 touches were extracted then all 5 touches are sent. The input parameters were kept in memory after the benchmarks were constructed so that the indexing algorithm would have a sequence of instructions to follow. The indexing process involved creating counters to compare against the number of touches and number

of updates per touch in the parameter. The counters are constantly updated before the next frame is fired by the system. For example, for a 1 touch and 10 updates parameter, the indexer knows that there will be 1 message for down, 10 updates, and 1 up. When the frame is fired the down message is sent. When the next frame is fired the first update message is sent. Once all of the updates are sent, then the up message is sent. Counters are reset based on the next set of input parameters after all data in the workload is sent. This process is illustrated in Figure 6.4 and the algorithm is provided in Appendix B.

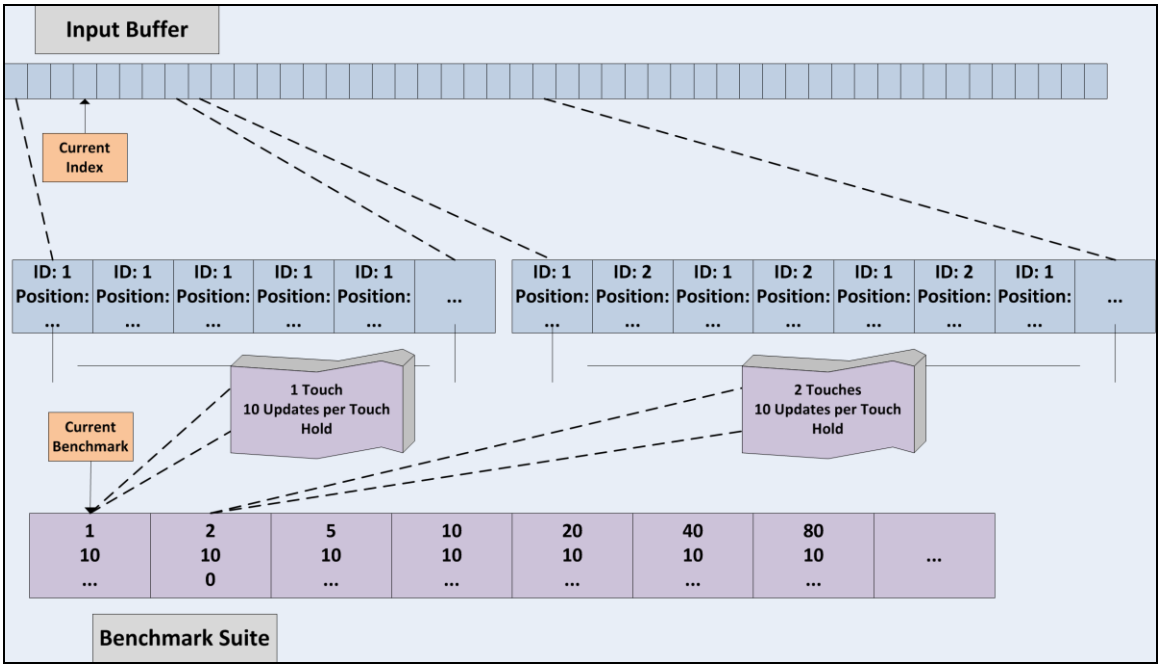


Figure 6.4 The Indexing Process for Timing Simulated Multi-Touch Data Transfer

6.2.1.3 Server-Side Monitoring and Logging

For monitoring, the benchmark harness takes measurements from two points along the system's pipeline: one in the tracker just before data is sent and one in the client application after messages are processed by WISPTIS and the corresponding callbacks are received by the client. Two strategies were used to minimize resource usage: logging

monitored behavior for later analysis and keeping non-essential processes, such as the image processor and the network packet processors, in Windows idle. The computer was disconnected from the network and removed from the SMART Table before the benchmark harness was activated. All of the logged data was kept in memory until the benchmark harness completed its run, and then was written to file for future analysis.

The benchmark logging method, outlined in SPE for historical analysis, was chosen for several reasons. First, it helped minimize the impact on resource usage at run-time so that measurements would not be affected. It also gave the transfer components as much access to system resources so they would not have to compete with other processes. Finally, it opened up the possibility of making informal observations, such as viewing CPU or Memory Usage in the Windows Task Manager, without taking measurements because the Task Manager and network processes were the primary resource users. [12]

When the benchmark harness was activated, the tracker and client application monitored the system as workloads were applied. At both of these points along the pipeline, observations were made. An observation was made by creating a series of timestamps. A timestamp represents a single snapshot of touch data that was recorded by the system during the observation. Each timestamp contains the touch's ID, State, Position, and Tick Count from the system clock. Other information, such as Width, Height, Angle, etc. was also included as additional fields. Timestamps were stored in memory and then written to separate text files after the benchmark harness completed its run. The tracker generated one set of text files from the first measurement point and the client application generated another set of text files from the second point.

6.2.2 The Client

MTScratchpad, an open-source code sample included in the Window SDK software bundle, was the client application for the system. This code sample demonstrated how developers could access touch data through WM_TOUCH and WM_GESTURE messages in the WndProc callback by drawing ink strokes on the screen. [51] The Windows SDK was chosen as the Windows 7 Touch API over WPF because of the speed and simplicity of Windows Messages.

WM_TOUCH messages were selected because they represent processed touch data that mainstream APIs accept in their input layers. They also closely resemble the original messages that were sent to the OS and, since the transport layer was being evaluated, it was decided to stay as close as possible to that format and to stay away from routed event callbacks. Furthermore, it is unknown how much latency is added the system as WPF unpacks the WM_TOUCH data and encapsulates it inside its own data structures. This could involve additional processing beyond what WISPTIS provides.

6.2.2.1 Client-Side Monitoring and Logging

For the evaluation, the MTScratchpad also remained as a primitive application that only had two modes: Drawing and Logging. The application was upgraded to include the same logging functionality as the tracker through a reusable module. Touch data was received through a WM_TOUCH message in WndProc. The rendering code was disabled when the benchmark harness was activated so that the application would not do any CPU intense work. However, the rendering code could be re-activated if informal visual observations were taken to report additional information regarding both protocols.

6.2.2.2 Protocol Bridging

In order to make a uniform comparison it was necessary to take measurements at the same point before transmission and at the same point after transmission. When making a comparison at the end point, the callbacks must be of the same format to avoid comparing data from two completely different sets of callbacks. [12] This holds true for multi-touch protocols since the callbacks in TUIO_CPP greatly differ in format from the callbacks in the Windows SDK.

To achieve a single uniform format, both formats must be reconciled with each other. This involves choosing a single callback format and converting one set of callbacks over to the chosen format, typically achieved through a bridge. [12] The TUIO Blob callbacks were converted to WM_TOUCH messages, which are widely supported in APIs such as WPF. Bridges on the TUIO website, such as Multi-Touch Vista, could not be used because they only supported TUIO 1.0 at the time and not the blob profile in TUIO 1.1. [38] [66] For this evaluation the same method used by Multi-Touch Vista, an injection, where TUIO data travels through the Windows 7 Touch pipeline and processed by WISPTIS.

The choice to bridge TUIO into Windows 7 Touch was made because the latter format seems to be more widely adopted by mainstream developers than the former. [1] [27] [28] While this choice adds some extra latency to the TUIO results, Chapter Eight explains that TUIO's performance degradation under higher workloads can mostly be attributed to the TUIO_CPP API producing too many duplicate messages and not the bridge. Chapter Eight also outlines alternate methods that were attempted to achieve protocol conversion in order to avoid a formal injection.

6.2.3 Data Transfer

This section outlines the work conducted in the transport layer of the system. In order to make the evaluation as fair as possible, both protocols should send roughly the same amount of data. [12] In fact, the difference in workload size is a direct result of the protocol implementations and represents the same conceptual workloads.

Work was conducted in the area of payload sizes of HID and TUIO messages to determine similarities and differences. Specifically, different HID payloads were compared to all of the touch profiles in TUIO 1.1. The idea behind this was to compare an HID *Touch Point* descriptor to TUIO Cursor, an HID *Touch Object* descriptor to TUIO Object, and an HID *Touch Blob* descriptor to TUIO Blob. Therefore, HID payload size could be an independent variable in the evaluation. However, due to mounting technical challenges faced in the Windows 7 Touch APIs, this was a difficult goal to achieve and a solution was not fully implemented. However, an attempt was made to modify the payload of an HID descriptor so that it would send the same amount of data.

6.2.3.1 Payload Size Analysis

The TUIO Blob profile was selected as the primary payload for the comparison because it bore the closest resemblance to the HID payload. TUIO Blob had fields for Width and Height, but TUIO Cursor did not. TUIO Object had these fields but also had a field for tags, which was not needed. Figure 6.5 illustrates the comparison conducted on both payloads. There were two comparisons made, one before the HID payload was modified (left) and one after (right). It can be seen that there are significant differences in both leading to an unbalanced comparison.

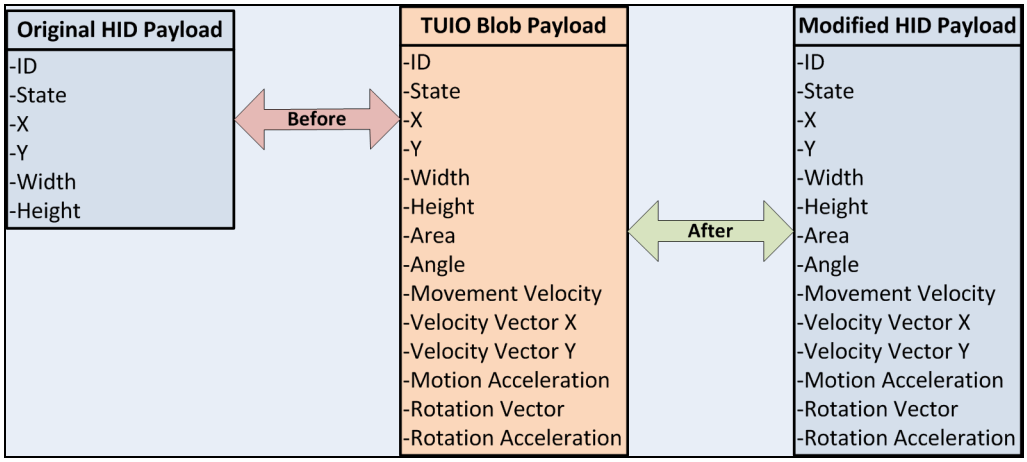


Figure 6.5 A Payload Comparison of an HID Descriptor and a TUIO Blob

A multi-touch HID descriptor typically carries six values: ID, State, X and Y Coordinates, and sometimes carries the optional values of Width and Height to support technologies that are capable of reading parts of hand. [5] [7] [60] [61] The TUIO Blob profile carries fourteen values: ID, State, X and Y Coordinates, Width, Height, Area, Angle, Movement Velocity, Velocity Vector X, Velocity Vector Y, Motion Acceleration, Rotation Velocity, and Rotation Acceleration. [65] There is clearly marked a difference, as the blob message carries over double the amount of data than HID does.

To even out both payloads, there were two options available: modify the HID payload or modify the TUIO payload. To help developers understand payload size, a payload can be divided into two sections: a Header and a Body. The headers for an HID payload are different than the headers in a TUIO payload. [54] [65] Changing one payload's header to suit the other proves to be counter-productive because they are defined by the original protocol specifications and are enforced in the APIs. Furthermore, modifying the body of TUIO payload would not solve this problem because the 1.1 protocol specifications firmly state that all fourteen values must be sent. [65]

Any modifications to the blob message code in TUIO_CPP, such as deleting values Angle through to Rotation Acceleration or attempting to change the header, would mean that TUIO would no longer being compared in this evaluation. Instead, a hybrid version of TUIO 1.1 that specifies a smaller payload for the original blob message would be compared against HID. However, this restriction does not apply to HID. As mentioned in Chapter Three, the HID protocol is the very flexible. It allows developers to construct their own payloads. They can specify the size and order of the usages in their descriptors. The original descriptor's usages and their sizes can be, and were, modified to closely match the values in TUIO. [23] [56]

6.2.3.2 Modifying an HID Descriptor to Carry a Larger Payload

To achieve this goal, the descriptor in the original HID driver was modified to carry a larger payload to the OS. Based on the conceptual idea outlined in Figure 6.5, eight 32-bit Vendor-Defined usage values were added to the original descriptor. These eight fields are the same additional fields used by the new TUIO blob profile: Angle, Area, Movement Velocity, Velocity Vector X, Velocity Vector Y, Motion Acceleration, Rotation Vector, and Rotation Acceleration. A vendor-defined usage page containing these fields, which is illustrated in Figure 6.6, was inserted into the original descriptor.

```
0x06, 0xA0, 0xFF,                   // USAGE_PAGE (Vendor Defined)
0x09, 0x01,                         // USAGE (Vendor Usage 0x80)
0x15, 0x00,                         // LOGICAL_MINIMUM (0)
0x27, 0xFF, 0xFF, 0xFF, 0x7F,       // LOGICAL_MAXIMUM (2147483647)
0x75, 0x20,                         // REPORT_SIZE (32)
0x95, 0x08,                         // REPORT_COUNT (8)
0x81, 0x02,                         // INPUT (Data, Var, Abs)
```

Figure 6.6 A Vendor-Defined Usage Page for a Multi-Touch HID Descriptor

The idea to use a vendor-defined usage page came from the WacomKMDF Code sample from the WDK. [54] In that descriptor, two additional usage values were defined at the end of the descriptor. HID already deals with the Width and Height values so it was not necessary to add them as vendor-defined fields. The tracker was upgraded to calculate this data and then, depending on the transfer mode, injected all of it into Windows 7 or transmitted all of it through a TUIO socket.

6.3 Summary

This chapter provided an in-depth look at how to set up and implement a multi-touch protocol evaluation. SPE methods were applied to the design and implementation of a multi-touch benchmark harness. The next chapter continues the process of applying these methods to the data analysis phase of this evaluation as this thesis edges closer to answering the central research question.

Chapter Seven: Results

Once the implementation was completed, the benchmark harness was activated for the evaluation. The system logged significantly large amounts of data (approximately 2 GB), and the need to automate the data analysis process became necessary. This chapter provides an in-depth look at the process used to break down large amounts of logged multi-touch data for analysis. This is followed by a presentation of the final results for each of the dependent variables measured: Average Latency, Missing Message Count, and Duplicate Message Count.

7.1 Data Analysis Methodology

Since no previous work has been published in the area multi-touch performance data analysis, or even multi-touch performance management tool support, this process was devised completely from the ground up. This process was conducted very carefully since benchmarking results can be misleading. [12] The data analysis was broken up into four stages: Partial Manual Analysis, Timestamp Matching, Performance Calculations, and Statistical Computations.

7.1.1 Manual Analysis

A partial manual analysis was first conducted on the collected data. This proved necessary to explore the data and determine possible trends and patterns within it, since no related work exists in the area of analyzing data from a multi-touch software system performance evaluation. [12] Timestamps were imported into a MS Excel spreadsheet, shown in Appendix C, and divided into two sides: Beginning and Ending Timestamps.

A key issue at this stage was to match a log entry from the Beginning Timestamp series to a log entry from the Ending timestamps series. This was required to compute the

latency resulting from sending the data through the pipeline. Excel's built-in features such as auto-fill, ability to evaluate conditions through IF-Statements, and perform calculations were used at this stage. It helped visualize behavior in the data and helped establish the matching criteria becoming an integral part of the analysis. However, a full manual analysis proved too time-consuming on the entire data set because of three common trends. First, it was found that timestamps from the tracker often did not appear in the client. Second, massive amounts of redundant messages in the form of duplicates had appeared in the client. Third, some messages that appeared in the client were not in the same order that they were originally sent from the tracker side. These three factors turned the client's data set into a large jigsaw puzzle. The overall amount of data also proved difficult to import and sort in Excel.

The benchmark harness logged almost 2 GB of data which contained over 6.5 million timestamps. The manual analysis was not completed but was conducted long enough to roughly determine how HID and TUIO responded to the workloads. The Tracker logged a total of 771,936 timestamps (388,032 from HID and 383,904 from TUIO). The client application logged a total of 5,735,297 timestamps (656,068 from HID and 5,079,229 from TUIO). The differences in the number of logged values represent the differences in each protocol's format when taking server-side measurements and the presence of duplicate messages in the client.

7.1.2 The Eva Analysis Tool

With over 6.5 million timestamps to organize and compare, tool support was required to help automate the matching and calculation stages of the analysis. The work done during the manual analysis actually turned Excel into a low-fidelity prototype for a

multi-touch performance management tool. [11] [76] Excel contained all of the general layout features of what such a tool might look like, minus functionality. A tool, called Eva, was created to achieve a level of automation. Eva is a prototype for a historical multi-touch performance management tool that analyzes timestamps after a benchmark harness has logged its observations. The process Eva follows involved five steps: Read Data, Format Data, Match Data, Perform Calculations, and Write Results.

The first step involved reading timestamps from both sets of files (Tracker and Client) and storing their raw contents into two separate lists in memory. From here both lists were analyzed to determine how the timestamps could be matched, and it was found that only a subset formed the basis for the actual matching criteria. Four values from both lists were used in the matching criteria: ID, State, X, and Y. It was quickly found that the data was in three different formats: Raw HID and Raw TUIO from the Tracker, and Windows 7 Touch from the Client. It was decided to convert the two protocol formats over to the Windows Touch format help make the matching stage easier.

HID had four separate state values in the data: 0 (Adding), 1 (Updated), 2 (Removing), and 3 (Removed). [38] Underneath its primary callbacks (Down, Update, and Up), TUIO actually had seven separate state values in the data: 0 (TUIO_IDLE), 1 (TUIO_ADDED), 2 (TUIO_ACCELERATING), 3 (TUIO_DECELERATING), 4 (TUIO_ROTATING), 5 (TUIO_STOPPED), and 6 (TUIO_REMOVED). [66] During the manual analysis it was noted that HID used all four of its states, but the TUIO Server actually used five of its seven states. It used Added for Down messages and Removed for Up messages. However, it used Rotating, Accelerating, and Decelerating for Update messages. This was strange since all of the coordinates in the update messages for

movement interactions were evenly spaced apart thus negating the possibility of touches speeding up or slowing down. Indeed, Rotating was mainly used for Update messages with Accelerating and Decelerating being used occasionally. Three states were consistently used in Windows 7 Touch set: 1 (TOUCHEVENTF_MOVE), 2 (TOUCHEVENTF_DOWN), and 4 (TOUCHEVENTF_UP). [51]

The position coordinates were different as the HID set had raw device coordinates and the TUIO set had the same screen pixel format as the Windows Touch set. However, some of the TUIO coordinates were off-by-one digit when compared to the Windows Touch coordinates and additional checks were required. The IDs were also different as the HID and TUIO sets contained the original IDs: 1 for the first touch, 2 for the second touch, etc. However, the states were different in the Windows Touch set where 1149 was consistently used for the first touch, 1150 was used for the second touch, etc.

A timestamp from the beginning list and a timestamp from the ending list are considered matches if their States are identical, their IDs are identical, and their X and Y Coordinates are identical. The matching criterion forms the basis for Eva's matching algorithm. This algorithm matches timestamps from the beginning list with its counterparts in the ending list. It does this by breaking up the timestamps in both lists. Individual benchmarks are first identified in both lists and then timestamps are matched one benchmark at a time. Eva's matching algorithm is provided in Appendix D.

At the end of each benchmark, the matching algorithm was also responsible for calculating all three dependent variables and producing a master list of matched timestamps and their corresponding performance calculations. As each timestamp was matched the individual latency was calculated, which represents the time it took the

message to travel through the pipeline and arrive in the client application. The individual latency = Message Arrival Time – Message Transmission Time. When all of the individual latencies were calculated, the mean latency for the benchmark was calculated: Sum of all Latencies / The Total Number of Latencies.

The number of missing and the number of dropped messages proved trickier to calculate. One way to do this was to compare the differences in the lists: the Total Number of timestamps in end list – Total Number of timestamps in the beginning list. If the ending list contains less timestamps then the beginning list, then data loss occurred and the number of missing can be calculated. However, if any number of duplicated messages showed up in the client this would result in a false calculation. A detailed comparison of the timestamps was required during the matching. The matching algorithm maintains a running count for number of missing. If a matched candidate cannot be found in the ending list, then this counter is incremented. After it is stored it is also used to calculate the number of duplicates: Size of Ending List Benchmark – (Size of Beginning List Benchmark + Number of Missing).

7.1.3 Statistical Computation Method

After performance calculations were completed, the results were written to a text file for the next stage in the data analysis where the final statistics were calculated. A nonparametric test was used to compare HID with TUIO to determine the differences between both protocols. The Wilcoxon Signed Ranks Test was selected to make this comparison for all three of the dependent variables. Based on the calculated data, this test was conducted to determine which protocol is better for multi-touch data transfer. [77]

A nonparametric Rank-Test was conducted, instead of a parametric T-Test, for two reasons. First, it was apparent that when comparing benchmark results from HID with benchmark results from TUIO there were outliers in the TUIO data set. Second, there was only one overall observation made per benchmark as individual benchmarks were only executed once. In these cases the overall form of data comparing both protocols could no longer be assumed to be normal with outliers indicating erratic behaviour in one of the protocols. With these issues present in the data, a nonparametric test proved more appropriate and is typically used in these situations. [77]

Since a nonparametric test was applied to the calculated data, the Spearman Rank Correlation Coefficient was applied to determine the strength of linear association between HID and TUIO for all three dependent variables. [78]. IBM's Business Analytics software, SPSS Statistics, was used calculate the final statistics for Average Latency, Number of Missing Messages, and Number of Duplicate Messages. [79] These results were generated in the form of ranks, correlation coefficients, and p-values to provide a compact summary of the result. [77] The final results are presented in the remaining sections of this chapter, which begins with the first variable Average Latency.

7.2 Average Latency Results

The first set of results is for the Average Latency variable. The results are grouped and presented by interaction. The following sections present the results from the Wilcoxon Test where each benchmark's average latency was assigned a rank. The Spearman Rank Correlation Coefficient is presented to determine the strength of linear association between HID and TUIO in terms of the average latencies they yielded. The p-value is presented to state the statistical significance of the result. Finally, a set of charts

are presented to illustrate the differences in average latency between both protocols. Each chart compares the impact workloads have on each protocol as they are increased.

To help visualize these results, the data is broken up into 2 benchmark sets (one chart for HID and one for TUIO). The x-axis represents the number of touches and the y-axis represents the average latency in milliseconds. The average latency values are provided in Appendix E. Since latency is a measure of time, the intended workloads are expressed in their original form of number of touches and number of updates to help visualize the scenarios when delay occurred. Each benchmark's average latency was assigned a rank based on the following equalities:

- If *avg_latency_tuio* < *avg_latency_hid*, then TUIO was faster than HID because it yielded a lower latency and a *negative* rank is applied.
- If *avg_latency_tuio* > *avg_latency_hid*, then HID was faster than TUIO because it yielded a lower latency and a *positive* rank is applied.
- If *avg_latency_tuio* = *avg_latency_hid*, then TUIO was observed to be just as fast as HID and a *tied* rank is applied.

7.2.1 The Hold Interaction

Of the 32 tests, there were 3 negative ranks, 29 positive ranks, and 0 ties. TUIO was faster on 3 of the tests, HID was faster on 29 of the tests, and they were never the same. The correlation coefficient was 0.596, indicating a moderate to good relationship. The p-value was 0.000 (< 0.05), which indicates the result is statistically significant.

Figure 7.1 illustrates the average latency comparison for the Hold interaction. 3 HID outliers were removed: 412 ms (50 Updates), 816 ms (100 Updates), and 1613 ms (200 Updates). 5 TUIO outliers were removed: 424 ms (1 Touch, 50 updates), 813 ms (1 Touch, 100 Updates), 2164 ms (1 Touch, 200 Updates), 636 ms (100 Touches, 100 Updates), and 842 ms (100 Touches, 200 Updates). Latencies in HID consistently

hovered between 0 and 16 ms, occasionally climbing to 32 ms and then back down to 0. TUIO behaved more erratically with latencies constantly shooting up to 32 to over 100 ms and then back down to 0 ms many times during the course of the interaction.

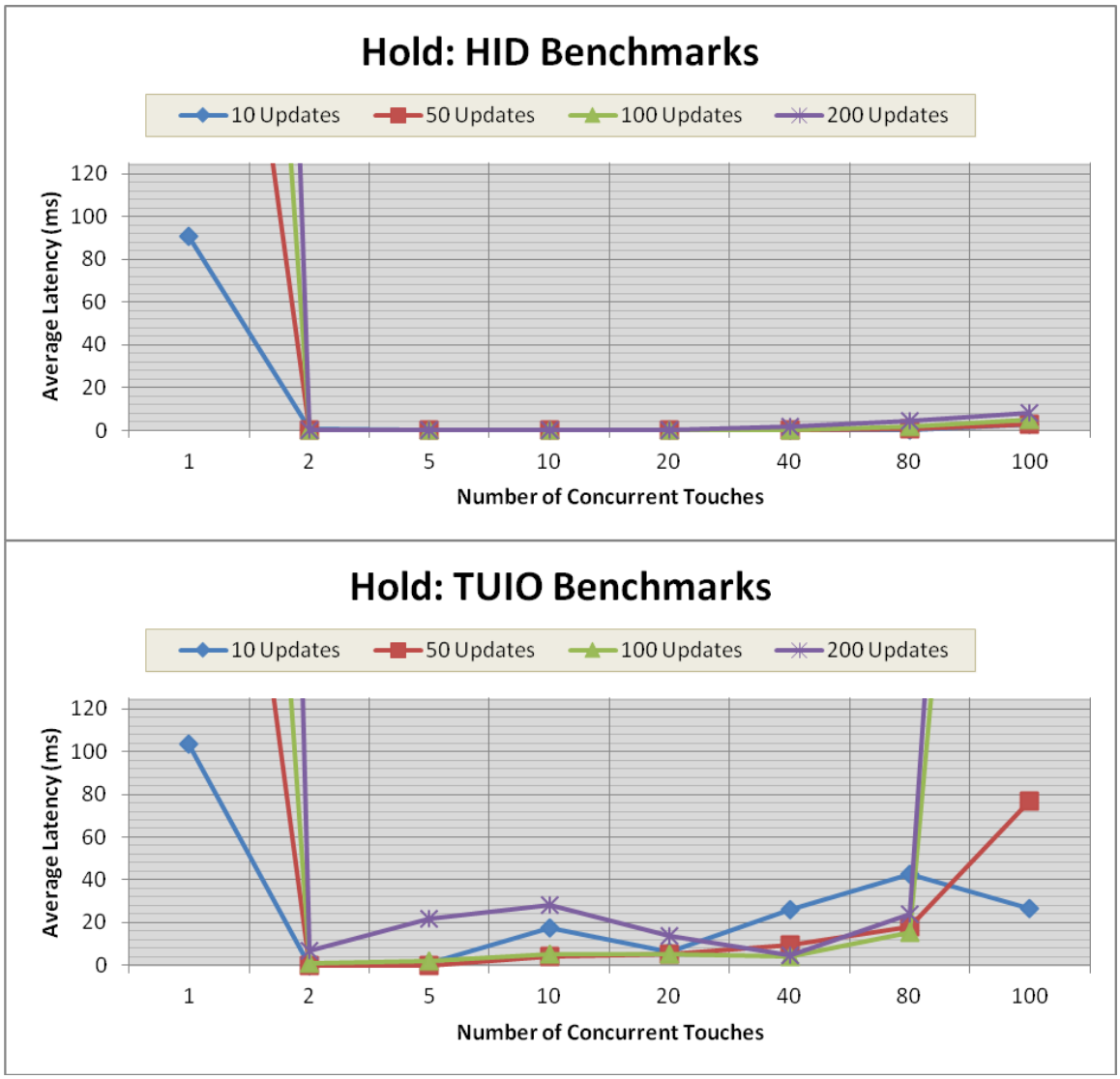


Figure 7.1 Comparing the Average Latency Between HID and TUIO for Hold

This behaviour was repeatedly noted as the number of updates per touch was increased and latencies started to climb as high as 515 ms (half a second) and for smaller workloads (1 to 5 touches) and would exceed 1000 ms (1 second) for larger workloads

(80 to 100 touches). The highest latency observed from TUIO was 15912 ms (15 seconds) and 234 ms from HID.

7.2.2 The Down Interaction

Of the 32 tests, there were 4 negative ranks, 28 positive ranks, and 0 ties. This means TUIO was faster on 4 of the tests, HID was faster on 28 of the tests, and they were never the same. Figure 7.2 illustrates the comparison for the Down interaction.

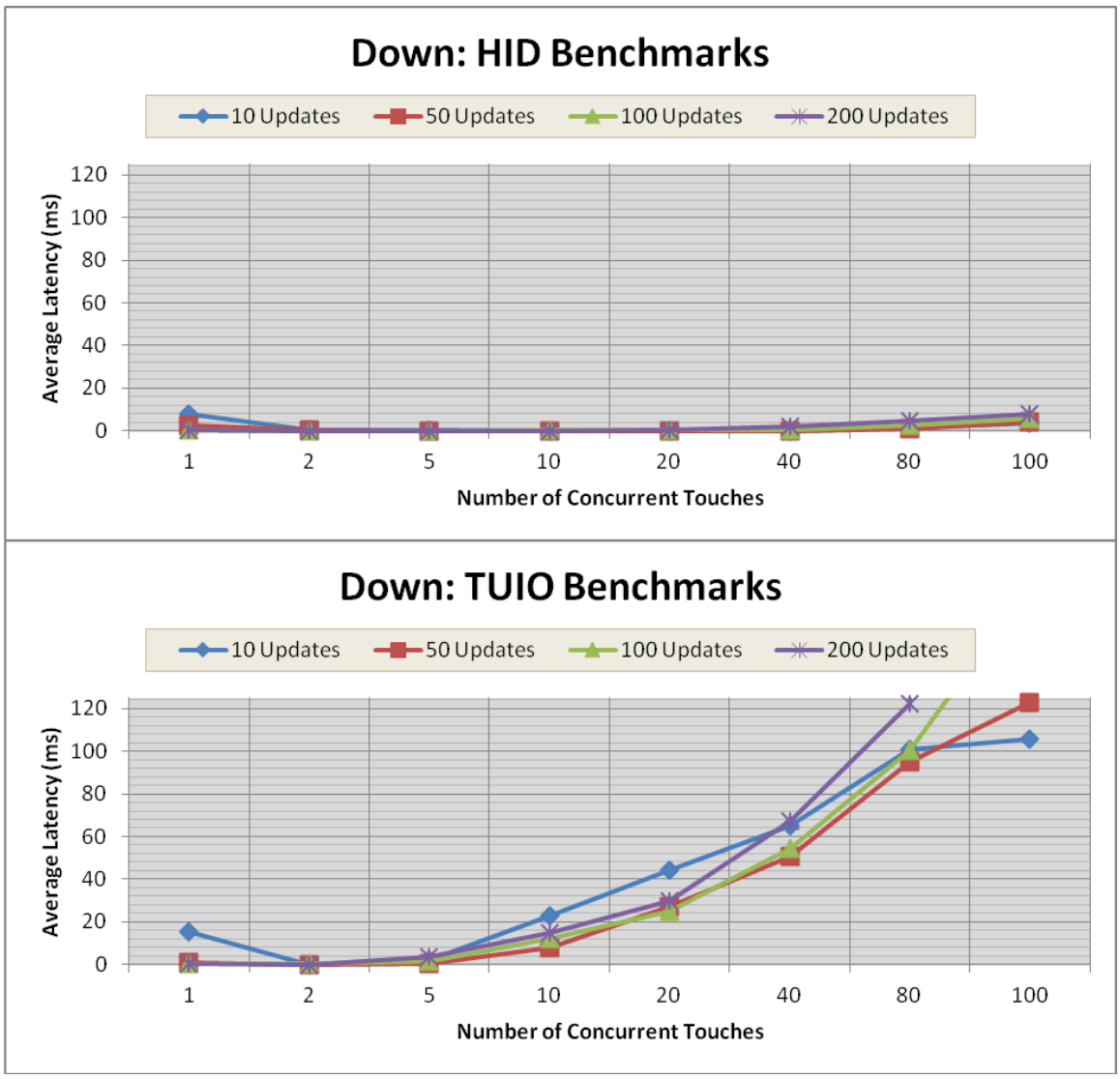


Figure 7.2 Comparing the Average Latency Between HID and TUIO for Down

The correlation coefficient was 0.509, indicating a moderate to good relationship. The p-value was 0.003 (< 0.05), which indicates the result is statistically significant.

2 outliers were removed from the TUIO set: 177 ms (100 Touches, 100 Updates) and 14586 ms (100 Touches, 200 Updates). Latencies in HID consistently hovered between 0 and 16 ms, occasionally climbing to 32 ms and then back down to 0. Both protocols were close together (0 to 16 ms) for workloads involving 1 and 2 concurrent touches. Afterwards, however, TUIO began to slow down with latencies constantly shooting up to 16 ms typically remaining there for longer periods before going back down to 0 ms. This behaviour was continually noted as workloads increased and latencies climbed as high as 515 ms often staying above 100 ms for longer periods before going back down. The highest latency observed from TUIO was 44132 ms (44 seconds) and 234 ms from HID.

7.2.3 The Right Interaction

Of the 32 tests, there was 1 negative rank, 31 positive ranks, and 0 ties. This means TUIO was faster on 1 of the tests, HID was faster on 31 of the tests, and they were never the same. The correlation coefficient was 0.579, indicating a moderate to good relationship. The p-value was 0.001 (< 0.05), which indicates the result is statistically significant. Figure 7.3 illustrates the average latency comparison for the Right interaction.

5 outliers were removed from the TUIO set: 129 ms (100 Touches, 50 Updates), 112 ms (80 Touches, 100 Updates), 270 ms (100 Touches, 100 Updates), 132 ms (80 Touches, 200 Updates), and 18022 ms (100 Touches, 200 Updates). Latencies in HID consistently hovered between 0 and 16 ms, occasionally climbing to 32 ms and then back down to 0. Both protocols were close together for workloads involving 1 and 2 touches.

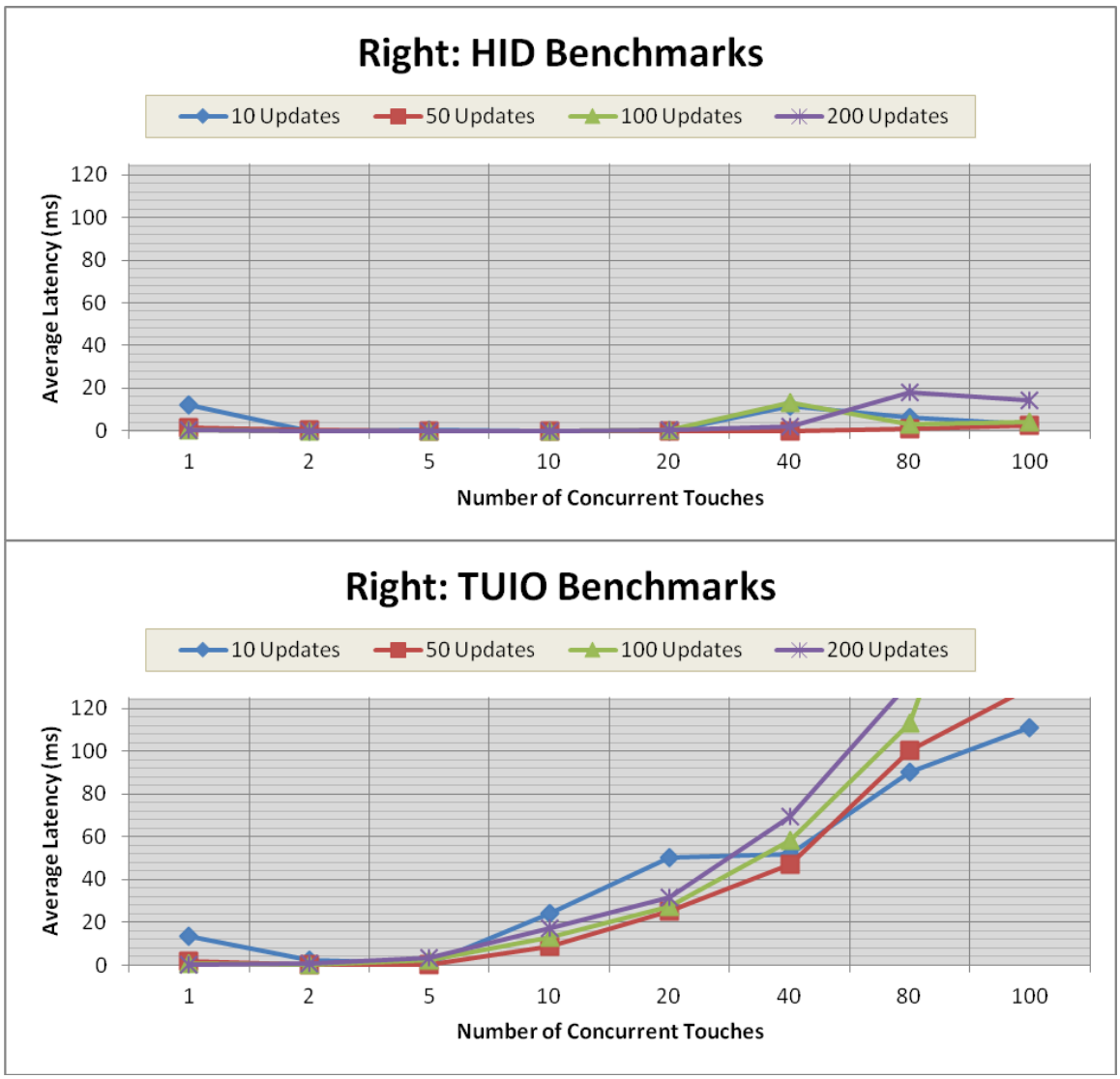


Figure 7.3 Comparing the Average Latency Between HID and TUIO for Right

Afterwards, however, TUIO began to slow down with latencies constantly shooting up to 16 ms, remaining there for longer periods before either going back down to 0 ms or climbing higher. This behaviour was continually noted as workloads increased and latencies climbed as high as 562 ms and staying well above 100 ms for longer periods. The highest latency observed from TUIO was 74225 ms (well over a minute) and 218 ms from HID.

7.2.4 The Diagonal Interaction

Of the 32 tests, there were 3 negative ranks, 29 positive ranks, and 0 ties. This means TUIO was faster on 3 of the tests, HID was faster on 29 of the tests, and they were never the same. The correlation coefficient was 0.462, which indicates a fair relationship. The p-value was 0.008 (< 0.05) which, indicates the result is statistically significant. Figure 7.4 illustrates the average latency comparison for the Diagonal Interaction.

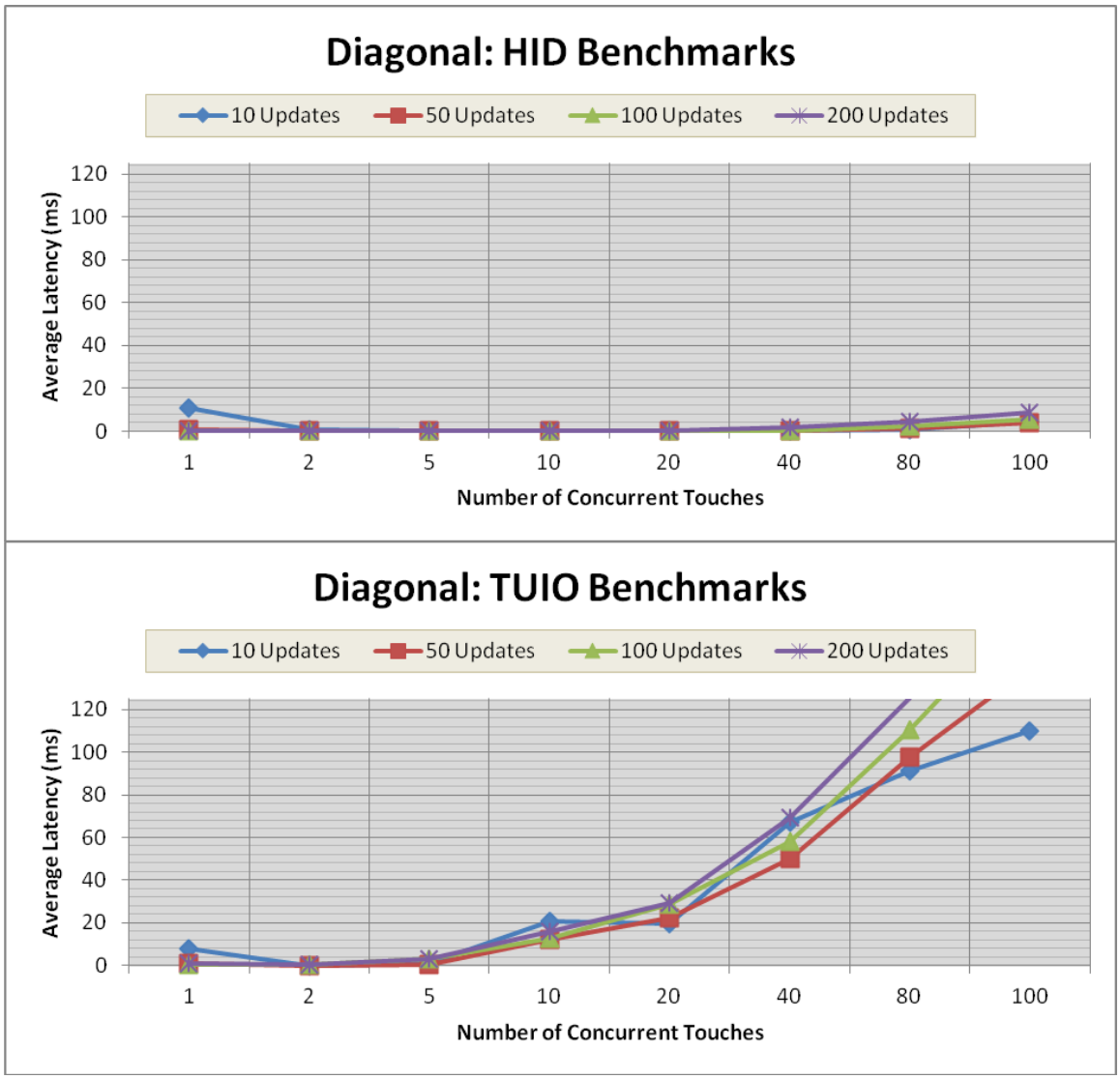


Figure 7.4 Comparing the Average Latency Between HID and TUIO for Diagonal

4 outliers were removed from the TUIO data: 139 ms (100 Touches, 50 Updates), 169 ms (100 Touches, 100 Updates), 125 ms (80 Touches, 200 Updates), and 1110 ms (100 Touches, 200 Updates). Latencies in HID consistently hovered between 0 and 16 ms, occasionally climbing to 32 ms and then back down to 0. Both protocols were close together for workloads involving 1 and 2 concurrent touches. Afterwards, however, TUIO began to slow down with latencies constantly shooting up to 16 ms and remaining there for longer periods before either going back down to 0 ms or climbing higher. This behaviour was continually noted as workloads increased and latencies climbed as high as, or higher than, 562 ms and staying well above 100 ms for longer periods before going back down. The highest latency observed from TUIO was 16723 ms (16 seconds) and 234 ms from HID.

7.3 Missing Message Results

The next set of results is for the second dependent variable Missing Messages. The results are grouped and presented by interaction. The following sections present the results from the Wilcoxon Test where each benchmark's missing count was assigned a rank, followed by the Spearman Rank Correlation Coefficient to determine the strength of linear association between HID and TUIO in terms of the percentage of messages that failed to appear in the client. The p-value is presented to state the statistical significance of the missing message results. This is followed by a chart to illustrate the comparison.

To help visualize the results, all update sets are included in the same chart. The x-axis represents the workload (number of touches and number of updates per touch) and the y-axis represents the percentage of messages that were that failed to appear in the

client. The percentages are provided in Appendix F. Each benchmark's missing message count was assigned a rank based on the following equalities:

- If *num_missing_tuio* < *num_missing_hid*, then HID lost more messages than TUIO because it yielded a higher value and a *negative* rank is applied.
- If *num_missing_tuio* > *num_missing_hid*, then TUIO lost more messages than HID because it yielded a higher value and a *positive* rank is applied.
- If *num_missing_tuio* = *num_missing_hid*, then TUIO lost as many messages as HID and a *tied* rank is applied.

7.3.1 The Hold Interaction

Of the 32 tests, there were 0 negative ranks, 3 positive ranks, and 29 ties. This means HID never lost more messages than TUIO, TUIO lost more messages than HID on 3 of the tests, and they both lost the same amount of messages on 28 of the tests. The correlation coefficient was 0.596, indicating a moderate to good relationship. The p-value was 0.000 (< 0.05), which indicates that the result is statistically significant. Figure 7.5 illustrates the comparison for the Hold interaction.

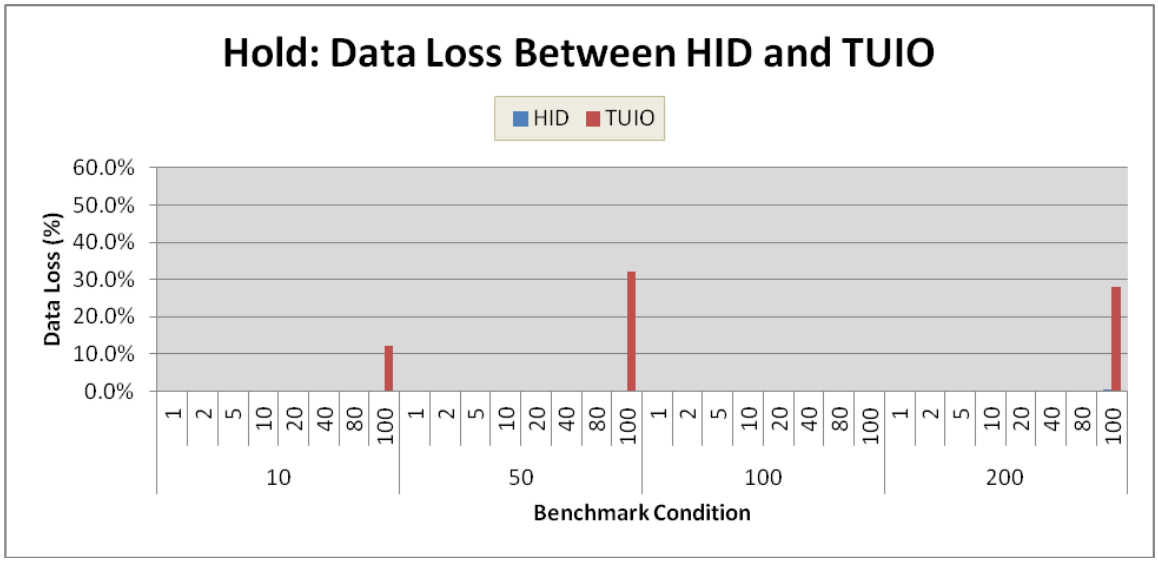


Figure 7.5 Comparing Percentage of Messages Lost for HID and TUIO for Hold

7.3.2 The Down Interaction

Of the 32 tests, there were 0 negative ranks, 27 positive ranks, and 5 ties. This means HID never lost more messages than TUIO, TUIO lost more messages than HID on 27 of the tests, and they both lost the same amount of messages on 5 of the tests. The correlation coefficient was 0.502, indicating a moderate to good relationship. The p-value was 0.003 (< 0.05), which indicates the result is statistically significant. Figure 7.6 illustrates the comparison for the Down interaction.

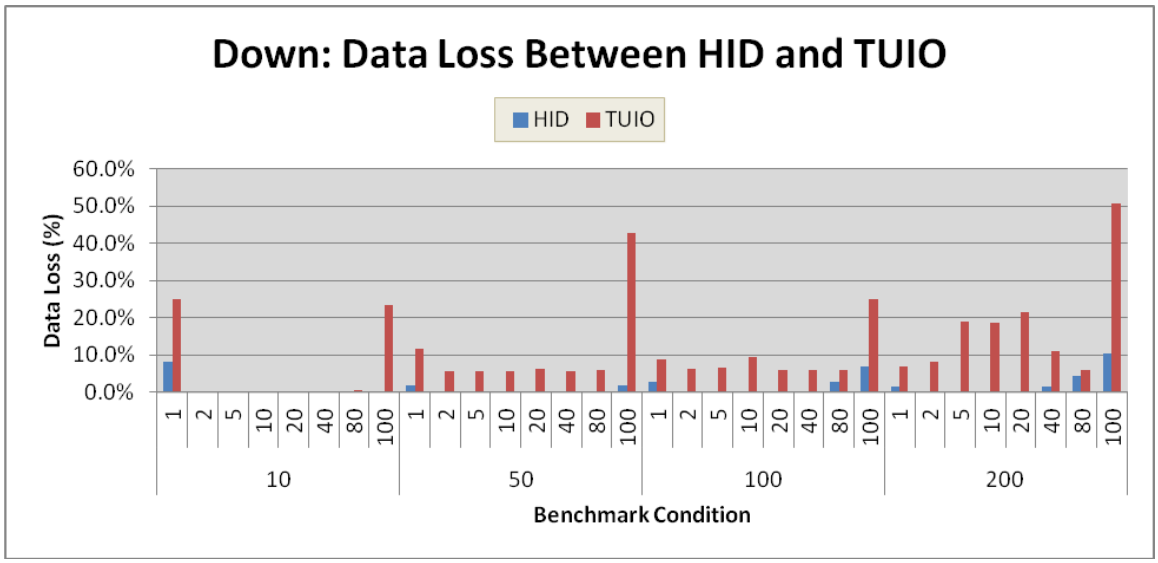


Figure 7.6 Comparing Percentage of Messages Lost for HID and TUIO for Down

7.3.3 The Right Interaction

Of the 32 tests, there were 2 negative ranks, 17 positive ranks, and 13 ties. This means HID lost more messages than TUIO on 2 of the tests, TUIO lost more messages than HID on 17 of the tests, and they both lost the same amount of messages on 13 of the tests. The correlation coefficient was 0.491, indicating a fair relationship. The p-value was 0.004 (< 0.05), which indicates that the result is statistically significant. Figure 7.7 illustrates the comparison for the Right interaction.

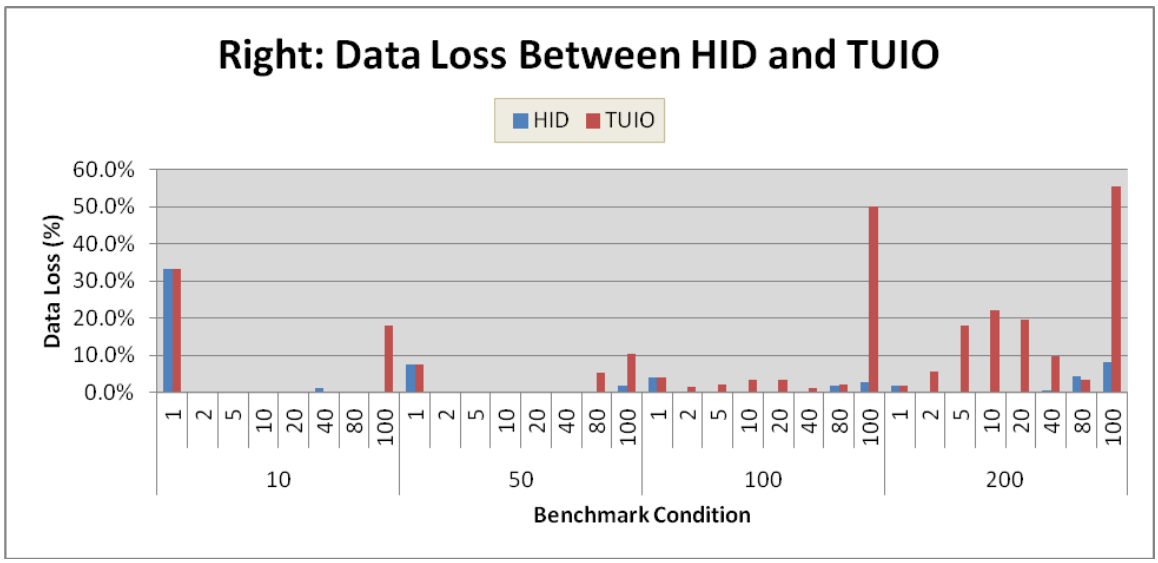


Figure 7.7 Comparing Percentage of Messages Lost for HID and TUIO for Right

7.3.4 The Diagonal Interaction

Of the 32 tests, there were 0 negative ranks, 26 positive ranks, and 6 ties. This means HID never lost more messages than TUIO, TUIO lost more messages than HID on 26 of the tests, and they both lost the same number of messages on 6 of the tests. Figure 7.8 illustrates the comparison for the Diagonal interaction.

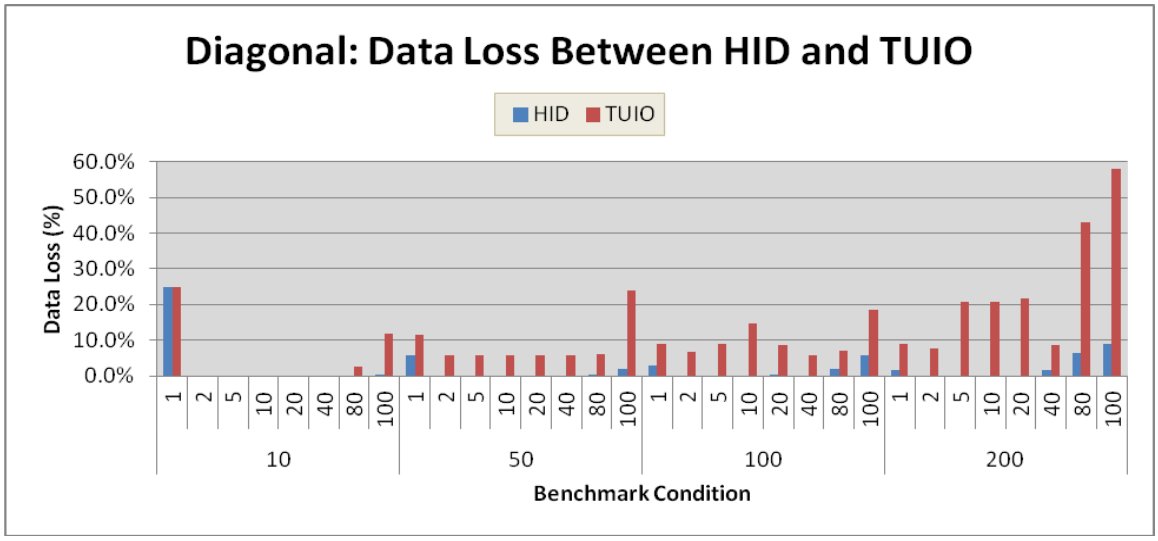


Figure 7.8 Comparing Percentage of Messages Lost for HID and TUIO for Diagonal

The correlation coefficient was 0.496, which indicates a fair relationship. The p-value was 0.004 (< 0.05), which indicates that the result is statistically significant.

7.4 Duplicate Message Count Results

The final set of results is for the third dependent variable Duplicate Message Count. The following sections present the results from the Wilcoxon Test where each benchmark's duplicate count was assigned a rank, followed by the Spearman Rank Correlation Coefficient to determine the strength of linear association between HID and TUIO in terms of the numbers of duplicates that arrived in the client. The p-value is presented to demonstrate the statistical significance of the duplicate results. This is followed by a set of charts to illustrate the comparison between both protocols variable.

The x-axis represents the workload (number of touches and number of updates per touch) and the y-axis represents the number of duplicate messages. A logarithmic scale was used on the y-axis so that all the collected data could be presented. Therefore, units increase 10-fold each time on the y-axis. The duplicate counts are provided in Appendix G. Since this variable measures message counts, the intended workloads are also expressed in message counts for easy comparison. The charts, however, contain the workloads in their original form. Each benchmark's duplicate message count was assigned a rank based on the following equalities:

- If *num_duplicate_tuio* < *num_duplicate_hid*, then HID generated more duplicate messages than TUIO because it had a higher count and a *negative* rank is applied.
- If *num_duplicate_tuio* > *num_duplicate_hid*, TUIO generated more duplicate messages than HID because it had a higher count and a *positive* rank is applied.
- If *num_duplicate_tuio* = *num_duplicate_hid*, then TUIO generated the same number of duplicate messages as HID and a *tied* rank is applied.

7.4.1 The Hold Interaction

Of the 32 tests, there were 0 negative ranks, 28 positive ranks, and 4 ties. This means HID never generated more duplicates than TUIO, TUIO generated more than HID on 28 of the tests, and they both generated the same number of duplicates on 4 tests. The correlation coefficient was 0.878, indicating a very good to excellent relationship. The p-value was 0.000 (< 0.05), which indicates the result is statistically significant. Figure 7.9 illustrates the comparison for the Hold interaction.

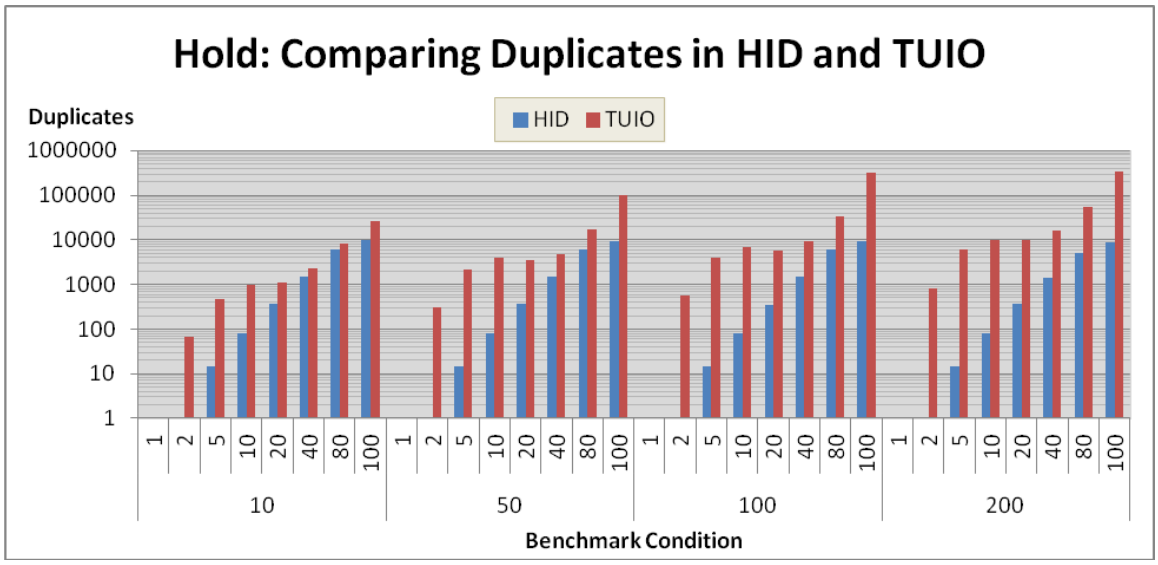


Figure 7.9 Comparing Duplicate Counts Between HID and TUIO for Hold

7.4.2 The Down Interaction

Of the 32 tests, there were 0 negative ranks, 32 positive ranks, and 0 ties. This means HID never generated more duplicates than TUIO, TUIO generated more than HID on all 32 tests, and they never generated the same number of duplicates. The correlation coefficient was 0.868, indicating a very good to excellent relationship. The p-value was 0.000 (< 0.05), which indicates the result is statistically significant. Figure 7.10 illustrates the comparison for the Down interaction.

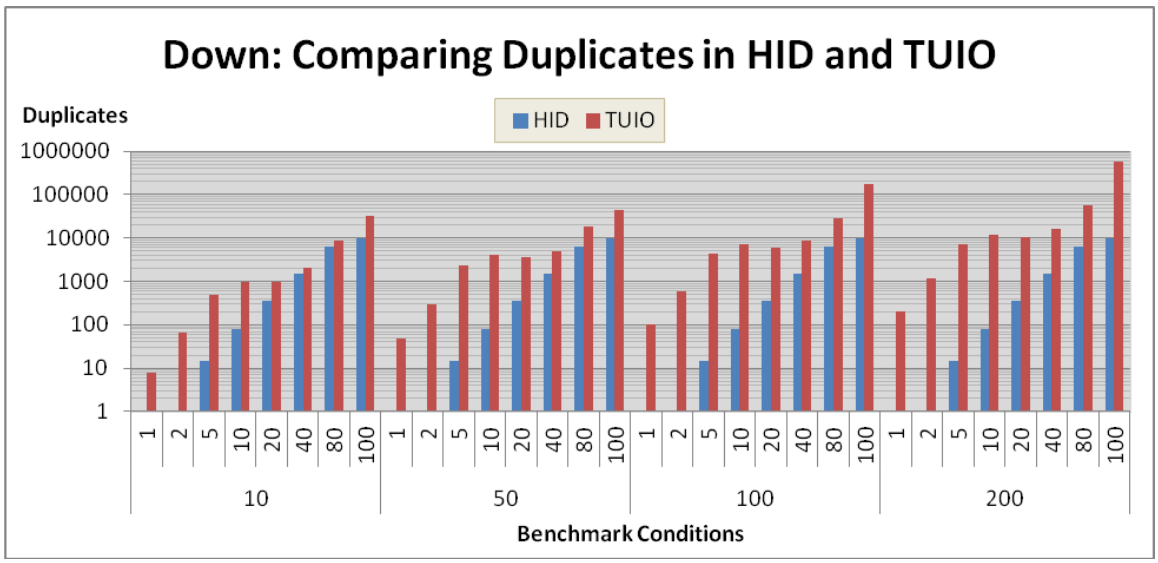


Figure 7.10 Comparing Duplicate Counts Between HID and TUIO for Down

7.4.3 The Right Interaction

Of the 32 tests, there were 0 negative ranks, 32 positive ranks, and 0 ties. This means HID never generated more duplicates than TUIO, TUIO generated more than HID on all 32 tests, and they never generated the same number of duplicates. Figure 7.11 illustrates the comparison for the Right interaction.

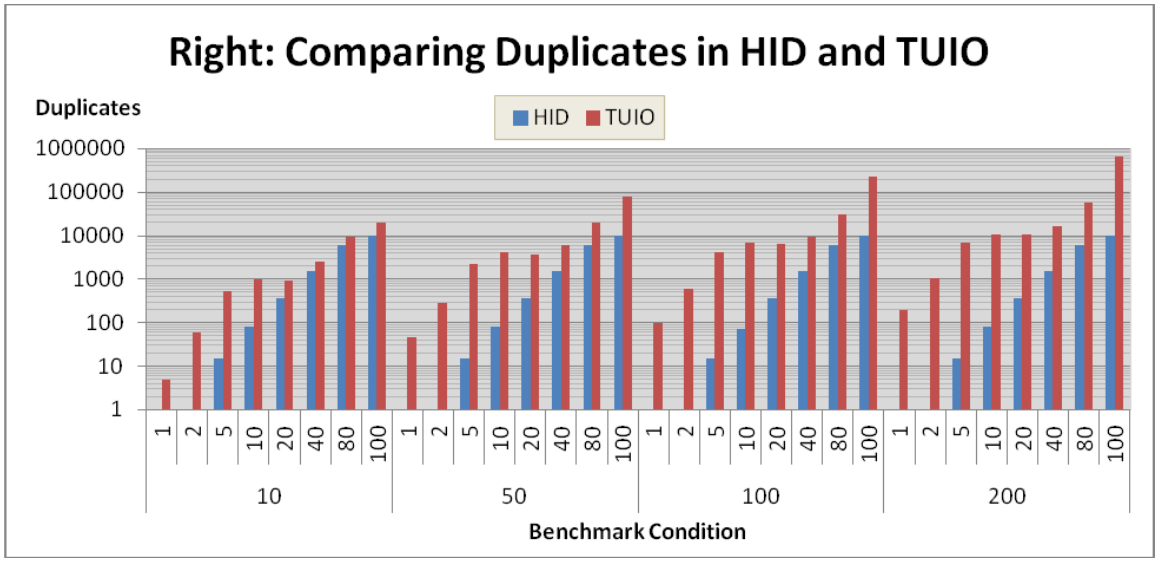


Figure 7.11 Comparing Duplicate Counts Between HID and TUIO for Right

The correlation coefficient was 0.867, indicating a very good to excellent relationship. The p-value was 0.000 (< 0.05), which indicates the result is statistically significant.

7.4.4 The Diagonal Interaction

Of the 32 tests, there were 0 negative ranks, 32 positive ranks, and 0 ties. This means HID never generated more duplicates than TUIO, TUIO generated more than HID on all 32 tests, and they never generated the same number of duplicates. The correlation coefficient was 0.864, indicating a very good to excellent relationship. The p-value was 0.000 (< 0.05), indicating the result is statistically significant. Figure 7.11 illustrates the comparison for the Diagonal interaction.

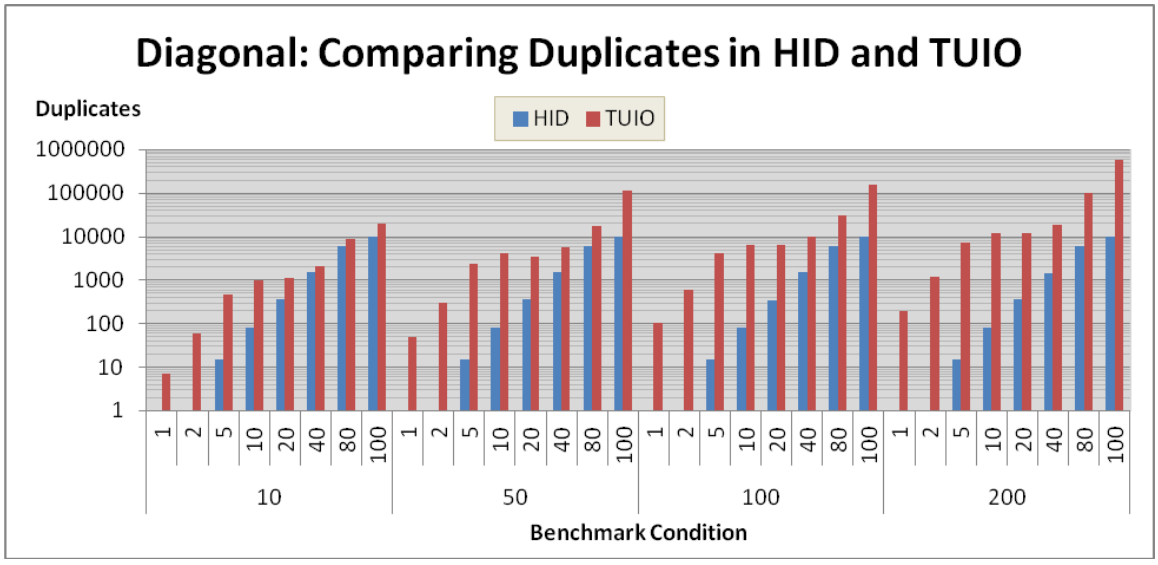


Figure 7.12 Comparing Duplicate Counts Between HID and TUIO for Diagonal

7.5 Summary

This chapter presented the results produced by the benchmark harness. SPE methods were further applied to the data analysis stage of this evaluation and the process followed was outlined. A discussion of these results now follows to interpret these results so that the research questions outlined in Chapter One can be answered.

Chapter Eight: Discussion and Limitations

This chapter provides a detailed discussion of the results produced by the benchmark harness. This is followed up by the technical challenges faced during the implementation and what developers can expect to encounter when attempting a similar evaluation in their own systems. Explanations as to how these challenges were addressed are also provided to help guide developers should they choose to recreate this evaluation.

8.1 Discussion of Results

The first section discusses the observed behaviour based on the formal measurements outlined in Chapter Six, which is followed by some informal observations made after the evaluation. Issues such as simulating jitter and observed message loss are discussed next. Finally, this section outlines possible reasons for TUIO's perceived performance degradation.

8.1.1 Observed Behavior: Formal Measurements

Figures 7.1, 7.2, 7.3, and 7.4 all demonstrate that both protocols are very similar until a certain workload is applied (typically around 5 concurrent touches) and then a noticeable latency gap begins to appear. HID gracefully handles even the highest workloads involving 100 concurrent touches and limits latencies to below 10 ms. TUIO tends to degrade under higher workloads (higher than 10 concurrent touches) and also demonstrates more erratic behavior with outliers showing very high latencies climbing as high as a 1000 ms (1 second) or 74000 ms (over a minute).

A high degree of fluctuations in latencies is an important issue to address because it introduces inconsistencies that can confuse end-users. For example, if a user is moving a picture with their finger and at one point along the interaction the system suddenly

stalls and the picture stops moving and, after a time, moves to catch up with the finger. At that point the fluidity of the interaction is broken leaving the user to wonder what had just happened, where they might ask themselves if they were at fault instead of the system.

This is issue is also important to address in the data analysis as it can affect the calculations for the average of the results. For example, the mean may not be the most suitable calculation for average if the data is showing outliers or if there is a high degree of fluctuation in the range of values. Instead, median or mode may be more suitable. However, developers must account for other factors, such as the overall sample size or if the data values are clustered together or spread out, when choosing an average to represent their results. In this case, HID latencies were clustered together indicating that the mean is a suitable calculation for the average. TUIO latencies, however, fluctuated more but did not fluctuate rapidly from extreme highs to extreme lows. Instead, latencies would rise to higher levels and remain there for a time before lowering. This indicates that the mean is still a good choice for average.

The outliers were removed in order to see this behavior in greater detail. The start of the upward slope represents the threshold at which one protocol begins to slow down. For some situations involving smaller workloads, HID is slightly slower than TUIO which remains unexplained. However, the majority of the results indicate that HID is faster and TUIO generally exhibits stress between 5 and 10 concurrent touches while HID exhibits stress at and above 80 touches. This difference is relevant because these situations mark a possible transition from single-user multi-touch interactions to multi-user multi-touch. While TUIO might be suitable for devices that assume a single user, or at least a small number of touches, it may not be a suitable for large tabletop devices that

have to support multiple concurrent users. At the very least, these results strongly indicate that further tuning is required for the TUIO_CPP API to address this problem.

The results for Hold show some peculiar behavior. First, at 1 touch they both show a very high average latency (~100 ms to ~2,000 ms) and this gets worse as the number of updates per touch is increased. This could be due to WISPTIS detecting a gesture in the given interaction. [49] Indeed, Windows Touch visualizations repeatedly appeared on the screen indicating that a gesture was recognized. For 10 and 50 updates per touch, the Tap visualization appeared. For 100 and 200 updates, the Press-and-Hold visualization appeared. WISPTIS might be withholding the WM_TOUCH messages and this could indicate the point when WM_GESTURE messages are fired off to clients.

The results for Hold also show that the latency gap begins to close between 10 and 20 touches, and then between 80 and 100 touches for 10 updates per touch. This was also observed between 10 and 40 touches for 200 updates per touch. The latency gap was expected to increase in the same manner as it did for Down. Instead, TUIO was closer to HID in terms of speed. This could be due to the fact that TUIO was originally created for tangible objects and TUIO_CPP was optimized for touches exhibiting little or no movement. However, it is equally possible that Eva made a mistake during its analysis of timestamps in stationary interactions.

The results for the movement interactions Right and Diagonal also produced some peculiar behavior. The results for Right show a strange spike at 40 touches for 10 and 100 updates per touch. The average latencies go back down after this. This spike also occurs at 80 concurrent touches for 200 updates per touch. Furthermore, the latency gap levels off for TUIO between 20 and 40 concurrent touches for 10 updates. The results for

Diagonal also show the average latency for level off in 10 updates per touch, but in this instance it happened between 10 and 20 concurrent touches. This behaviour remains unexplained, but it could be due to other side-effects and indicate that more observations per benchmark are required to see if this happens every time the workloads are applied. In this case, the each benchmark could be run at least five or more times and then the average of the results can be calculated to deal with side-effects that occurred. However, it was decided not to add this functionality to the benchmark harness because it would increase the total amount of data that needed to be analyzed afterwards.

It was important to present some of the individual latencies in Chapter Seven so that they can be compared with traditional response times. Unfortunately, no previous work exists on suitable response times for multi-touch software systems. This area still needs to be better defined so that developers will have solid targets to achieve when their systems are under construction. However, in 1968, Miller outlined different types of perceived response times and suitable targets for software system. Some of his guidelines included input devices such as keyboards and light-pens. Miller outlined that the delay should typically be no more than 100 ms (0.1 sec) when end-users conduct interactions with these devices. [80] It must be noted that, in light of Miller's definition, it is alarming to see the individual latencies of the TUIO transport pipeline often far exceeding 100 ms. A recent report shows that Microsoft Research is aiming for an overall 1 ms delay. They demonstrated the impact different response times had on touch interactions and showed that in situations involving a 1 ms delay, there was virtually no perceivable delay during the course of the interaction. [30]

8.1.2 Observed Behavior: Informal Insights

After the evaluation was conducted the benchmark harness was re-activated. The rendering code in the client application was turned on to *see* how fast both protocols handled the workload. These observations were not part of the formal measurements and were merely observations to gather additional information. HID was *perceived* to be faster than TUIO as ink strokes were drawn much quicker under HID than under TUIO.

Something strange occurred during the TUIO runs. It was observed that ink strokes became very choppy (multiple breaks in the stroke) under larger workloads. This could mean that data loss occurred during the TUIO interactions resulting in loss of focus for the interaction. The system misinterpreted this as a release and an up message was generated by WISPTIS. However, when the system was activated with the WM_TOUCH to TUIO bridge, this did not occur and could mean there are some memory clean up issues in the TUIO to WM_TOUCH bridge. However, it could also indicate that the factors slowing TUIO down also make it more unstable when put under duress and this has other unintended side effects to the proceeding stages in the pipeline.

The benchmark harness was re-activated a third time, but with the rendering code turned off in the client and the Task Manager in Windows 7 opened. It was *observed* that under TUIO, the benchmark harness was using large amounts of the CPU as larger workloads were applied. CPU usage was not a formally measured, but it was observed that TUIO was using 80 to 100% of the CPU to process all workloads involving 40 to 100 concurrent touches. It was less noticeable between 1 and 20 touches.

However, it must be noted that high CPU usage can be misleading because it does not necessarily mean that the system is performing poorly. Indeed, the CPU performs

better when it is in constant use executing instructions. However, attention must be paid when the entire CPU is being used for long periods of time and this is what was observed during the TUIO runs. [12] Under HID, CPU usage fluctuated between 30 and 60% and had a wider usage range. At times CPU usage would shoot up to around 80% but would go back down after the workload had been processed. However, under TUIO, it would rise to a very high level and remain there. Interestingly, CPU usage rarely went back down when the next workload was applied and usually fluctuated between 80 and 100%.

This could be a significant problem if TUIO, which is only responsible for transferring multi-touch data, conflicts with other processes over CPU access. Client applications are typically more elaborate than MTScratchpad. They can utilize a wide variety of APIs that might include physics engines for more elaborate rendering and these factors increase its dependency on system resources. [44] Multi-touch trackers with image processors also require access to system resources. Gesture processors are no exception, as the average latency results observed for Hold interaction indicates higher amounts of processing time. The responsibilities of an HID driver or a TUIO Server are less significant than other components in the pipeline, as surrounding processes typically run in the background unnoticed and should not hinder other processes.

It was observed that WISPTIS was a well-behaved process running in the background of the system. The Resource Monitor showed that it typically used between 2 and 4% of the CPU, but would never exceed this range even under larger workloads. WISPTIS may have limits when it comes to resource access in the OS. Under stressful situations it could have had difficulty interpreting messages (such as sudden interaction breaks) from TUIO and was unable to access additional resources to process the data. It

could see itself falling behind as latencies increased and chose to return an error in the system instead of dropping more messages.

A possible solution to the CPU usage issue would be to add faster hardware. However, adding faster hardware is not always the best solution to performance problems as they are often regarded as quick-and-easy solutions and do not always determine the root cause of the problem. [12] In TUIO_CPP's case, it was found that massive amounts of duplicate messages had appeared in the client. Faster hardware can give more resources to software system that is processing this many messages, but the problem remains that those duplicates are still being produced for developers running the same system on a variety of different hardware configurations. Often, fine-tuning algorithms to execute more efficiently or improving upon known design issues helps remove bottlenecks and reduce high latencies in software systems. [12]

A possible data ordering issue was noted for both protocols during the manual stage of the data analysis. It was found that significant numbers of timestamps in the EndingList were not being received in the same order that they were sent. This turned the client's data set into one large jigsaw puzzle and made the manual analysis a great deal harder to complete. Eva ignored this problem and only focused on finding possible candidates in the client regardless of where they were placed. However, this could indicate that there is another limitation but further investigation is required to formally measure and determine if protocols suffer from this. However, this could be a significant problem for gesture processors if they assume data arrives in the same order it was sent.

8.1.3 Simulating Jitter

The computer was removed from the SMART Table for the evaluation. This decision was made for two reasons: a dormant image processor minimized resource usage and it eliminated the need for participants to physically generate touch interactions. This proved beneficial since the benchmark harness was repeatedly activated. However, when conceiving workload values for touch simulation it was difficult to visualize, and differentiate between, realistic scenarios and stressful ones.

Furthermore, when ink strokes were viewed during the visual observations, the simulated interactions appeared to be *too perfect*. They either moved in a smooth line or remain perfectly stationary. Fingers are imprecise input devices making it impossible to have perfectly smooth interactions. This problem stems from the fact that the benchmark harness does not simulate jitter. [60] Also, the Eva tool was unable to effectively analyze data from manual interactions because it assumed smooth interactions without jitter. Jitter could be simulated by making slight modifications to the center point of a bounding box. The center point would still remain inside of the box, but could fluctuate by a couple of pixels around the original center point.

8.1.4 Data Loss

Figures 7.5, 7.6, 7.7, and 7.8 show data loss was present in both protocols, but it mainly occurred in the movement interactions. HID rarely dropped messages until large workloads involving 80 to 100 concurrent touches were applied. TUIO had a tendency to drop messages, but data loss in these cases generally remained under 10-15 % until workloads involving 80 to 100 touches were applied. Both protocols were stable for stationary interactions as HID rarely dropped messages and TUIO only dropped

messages at 100 concurrent touches. This further indicates that TUIO_CPP was originally optimized for stationary touches.

It was found that, through TUIO, the data for small to moderate numbers of concurrent touches remains intact with little loss only if they are not moving along a touch screen's surface for very long. While data loss is generally higher in TUIO, developers can expect to get most of their data when they use it. TUIO's algorithms for generating duplicates to account for data loss are very effective in minimizing it and only struggle in scenarios involving 80 and 100 concurrent touches. It must be noted that more messages may have been dropped in by both protocols, but WISPTIS could be generating messages (such as down and up messages) to ensure touch interactions remain smooth.

8.1.5 Duplicates and Workload Distortion

An explanation for TUIO slowing down can be attributed to very high numbers of duplicated messages arriving in the client, which was originally reported by the authors of TUIO. [24] Figures 7.9, 7.10, 7.11 and 7.12 show that message duplication occurred in HID and TUIO. This was interpreted in two ways. First, the duplicates generated by both protocols have different purposes. TUIO generates duplicates out of fear of message loss while duplicates inherent in HID are generated out of fear that there will be jitter in the interaction. Indeed, these duplicates are appearing at the beginning (when down messages come in) and at the end (when up messages come in) of an interaction. However, this could be a safety mechanism inside WISPTIS and it is generating duplicates instead of HID. This might also mean that WISPTIS is unintentionally adding duplicates onto the TUIO count, and this issue was investigated further. This issue can be expected in protocol bridging, as HID might appear to have a very high duplicate count if the system

were to be evaluated using a WM_TOUCH to TUIO bridge. [12] The TUIO Server might add duplicates during its stage of the bridge to ensure transfer. However, this issue was investigated further to determine if the high numbers observed were occurring in TUIO or in the bridge. Independent measurements were taken in the TUIO Listener where data is received in the TUIO Blob callbacks prior to bridging, and is illustrated in Figure 6.3.

Figures 8.1, 8.2, 8.3, and 8.4 illustrate the number of times the TUIO callbacks were fired in comparison with the workload sizes originally specified in the benchmarks. The x-axis contains the benchmark conditions. The y-axis contains the number of times a callback was fired in the TUIO Listener, which are presented using a logarithmic scale so all collected data could be presented. Units, therefore, increase 10-fold each time on the y-axis. The results show that the total number of messages arriving in the TUIO Listener far outweighs the number in the original workload. The workloads become severely distorted due to a high volume of messages that are then passed along to the bridge. For smaller workloads, such as 24 messages, 89 messages appeared in the client. This is more than triple its original amount to ensure all 24 made it through. This situation worsened as workloads became larger. For 60 messages, 563 appeared and is over nine times its original amount. For 20,200 messages, over 100,000 repeatedly appeared. Interestingly, the high numbers began to occur in workloads involving 5 concurrent touches and roughly where TUIO began to slow.

Large amounts of redundant messages can drive up the latency as they too must be dealt with in the pipeline. *Based on the data, it is believed that the TUIO socket in the TUIO_CPP API becomes clogged with duplicates and holds up the process of sending real messages in the workload.*

At very high workloads, WISPTIS is also adding duplicate messages. However, the number of duplicates added by WISPTIS is nearly identical for both pipelines. Hence, the measured differences in latencies between both protocols seem to be linked directly to the large amounts of duplicates coming from TUIO_CPP's Transport layer.

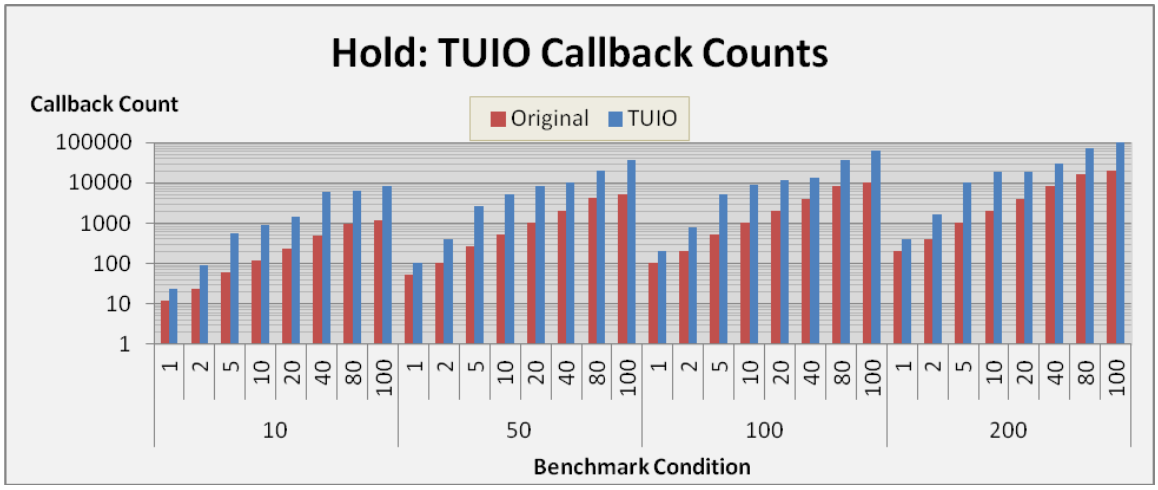


Figure 8.1 Comparing TUIO Callback Counts with Workload Counts for Hold

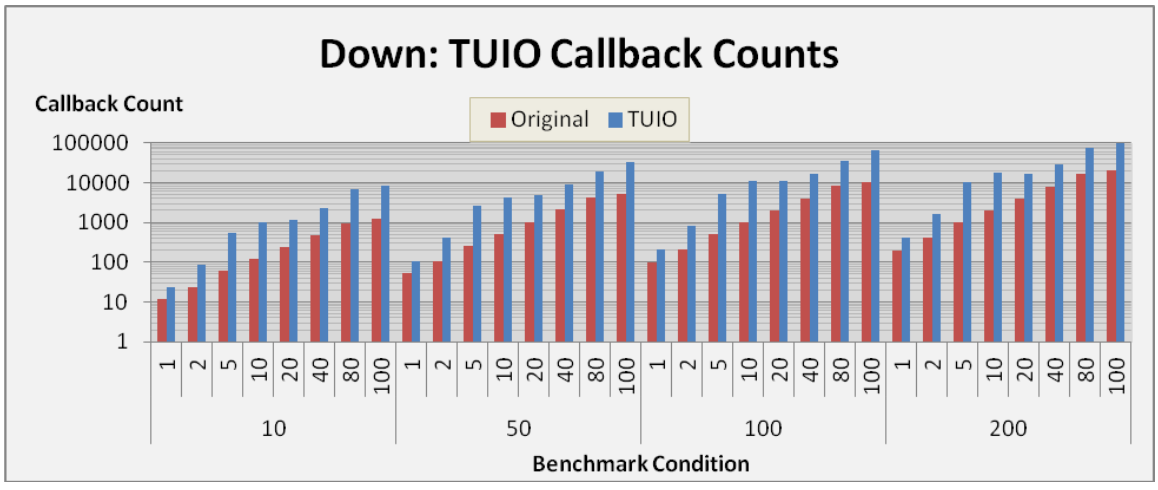


Figure 8.2 Comparing TUIO Callback Counts with Workload Counts for Down

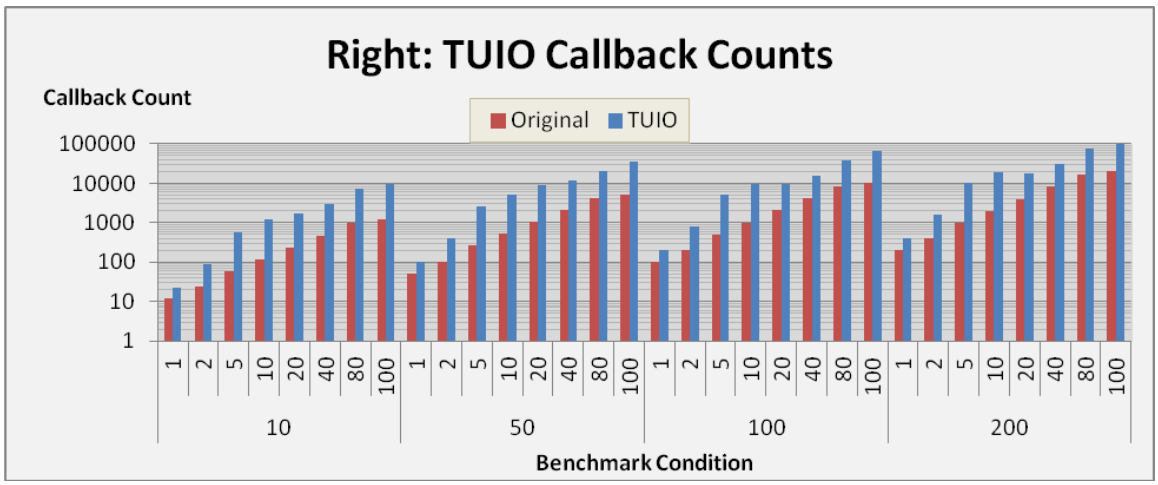


Figure 8.3 Comparing TUIO Callback Counts with Workload Counts for Right

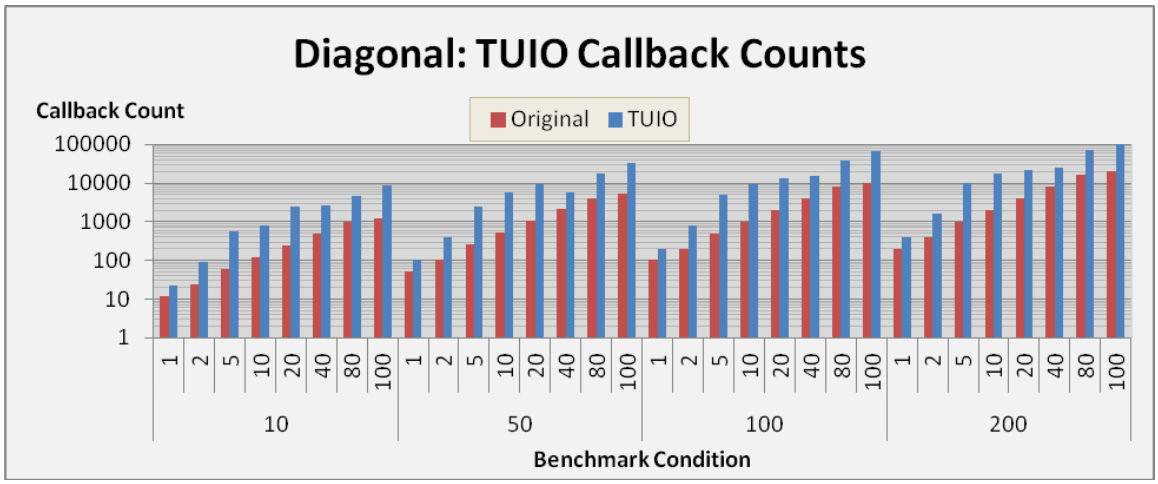


Figure 8.4 Comparing TUIO Callback Counts with Workload Counts for Diagonal

Workload distortion can be expected when comparing two protocols because of differing formats. When developers pass their workloads to a protocol they lose control of it until they appear in a client. They can therefore expect it to be distorted by the safety mechanisms in the APIs implementing the protocols. [12] This situation is reasonable since there are always format differences in the protocols that are being evaluated.

However, this situation becomes unreasonable when so many artificially created duplicates are required to smooth out irregularities in a protocol.

8.2 Technical Challenges and Limitations

All aspects of this evaluation's implementation suffered limitations brought on by technical challenges. This section outlines how solutions were devised to address and overcome them. The first two sections outline the issue of implementing measurement points and are followed by automation and payload size restrictions. Finally, issues surrounding timestamp matching and the overall development experience are presented.

8.2.1 Server-Side Measurement Point

The indexing approach did lead to one major limitation when it came to server-side measurements. Referring to Figure 1.3, touches do not have a state until they are packaged up into messages. Therefore, when touches emerge from the image processor in the form of bounding boxes they typically do not have a state at this stage of the processing pipeline. The tracker contains a list of all the touches currently making contact with the screen. At this point in the pipeline this list has not yet been updated. The tracker takes the locations of each bounding box and compares it with the locations of each touch it is currently tracking. At this point touches are given a state. Touches appearing for the first time are given a down state, touches appearing to have slightly moved are given an update state, and touches that have disappeared are given an up state. However, instead updating the bounding boxes, messages are created which do have states so they can be sent to the OS as quickly as possible. It was mentioned in the data analysis section of Chapter Seven that the presence of State field is an important field to have because it is part of the timestamp matching criteria.

The code that updates this list did contain some duplication when a TUIO Server was added to the tracker because of formatting differences between HID and TUIO. Due to this reality, code duplication could not be completely avoided and this meant that a measurement could not be taken at the exact same point in the tracker's code. However, the duplicated algorithms in this approach are completely identical with the exception of calls to the WDK and TUIO_CPP. Furthermore, an additional call to the WDK was required for a removed message in HID. This is an important issue because measurements should be taken as close to the actual call as possible otherwise they become less reliable due to inaccuracy. A measurement could be taken after the calls were made, but this would lead to an inaccurate measurement in the same way it would if the measurement is taken beforehand. [12]

8.2.2 Client-Side Measurement Point

When comparing two protocols, bridging always brings the possibility of adding delay to one protocol as additional time is lost in the conversion process. [12] Looking at the MultiTouch Vista code, it was noted that an injection operation was executed to achieve a TUIO to Windows Touch conversion. Multi Touch Vista currently supports TUIO 1.0 and not 1.1. Therefore, a custom-built bridge was required for the system. In order to avoid another injection, and give TUIO as much of an equal opportunity to be as efficient as HID, an attempt was made to use the SendInput and SendMessage functions to transmit messages to WISPTIS. [71] [72] [73]

Indeed, this method has been used in the past when developers have attempted to simulate Mouse and Keyboard data without using actual devices. [74] It was decided to try and use this same approach for bridging touch data to determine if this could help

alleviate the latency issue. Unfortunately, this proved to be unsuccessful as it was found that the OS was blocking the transmission of touch messages to WISPTIS. Microsoft was contacted to see if an injection was the only way to bridge. Unfortunately, this turned out to be the case and it was confirmed that in Windows 7 Touch an injection was the only way to achieve a TUIO 1.1 to WM_TOUCH bridge. However, it was recently announced that Windows 8 Touch will have a Touch Input Simulation API that allows developers to simulate touch input in client applications. [75] While this may not be a solution, it does warrant further investigation to see if it can be used to alleviate the problem of unintentionally slowing TUIO down during the bridging process.

8.2.3 Achieving Full Automation

While the data collection process is fully automated for each individual benchmark, some technical challenges prevented the full automation of benchmark harness. Several factors led to difficulties in achieving a fully automated benchmark harness. The first factor was that there was no master process controlling the entire system. Therefore, in order to activate the benchmark harness the client application and tracker had to be opened separately and then closed separately to deactivate it. The second factor was due to functionality in the tracker. The transfer mode had to be toggled in the UI and this feature was not automated inside of the tracker.

The third factor was more detrimental to the evaluation because it led to timestamp data being lost. This factor was due to a crash that repeatedly occurred when the benchmark harness was initially activated. In the early testing phases, the HID benchmarks would run but the as the TUIO benchmarks reached the end of the Diagonal interactions, the system would crash. This problem could not be fully explained.

However, the TUIO to WM_TOUCH bridge could be to blame. When the system was activated, and with the same workloads applied, using the WM_TOUCH to TUIO bridge all benchmarks completed their run without a crash. This could mean that the underlying protocol in the TUIO to WM_TOUCH bridge, the TUIO protocol, has a problem with data integrity and this led to the crash. When larger workloads were continually applied, TUIO might have had a problem keeping up with the incoming data. This problem could have gotten worse over time and WISPTIS found it difficult to process touches based on malformed data being injected. WISPTIS could have a safety mechanism that makes a decision to return an error rather than risk a crash on its end since this could disable Windows 7 Touch functionality for client applications running in the OS.

To deal with these problems, the benchmarks were broken up by protocol and by interaction type and then executed separately. All four interactions in the HID benchmarks were executed manually, one interaction at a time, followed by the TUIO benchmarks. The workload sequences in each interaction remained automated and this problem did not affect the system in any other way.

8.2.4 Payload Size Restrictions

Achieving payloads of equal size is impossible due to format differences in both protocols. However, efforts were made to equalize the payloads so they resemble each other as closely as possible. [12] As demonstrated in the previous section, it is possible to achieve this goal. However, a hybrid solution is required when using the Windows SDK in the retrieval process and can lead to problems later in the pipeline.

Windows 7 Touch only deals with a specific amount of data. Injecting larger payloads is possible because the HID protocol and the WDK API are flexible enough to

support it. However, the Windows SDK and WPF APIs are designed only to deal with the same amount of data that Windows 7 Touch specifies. Data that falls outside the realm of the TOUCHINPUT data structure of a WM_TOUCH message is completely ignored by WISPTIS and will not appear in the client. [52] [60] Thus injecting larger payloads can be achieved with no additional work but retrieving all of the data in the payload is not possible because they are ignored by the system. For the purpose of the evaluation, the workloads were equalized for both pipelines but extra data was not sent to the client applications as the descriptor was reverted to its original form.

Aside from WM_TOUCH and WM_GESTURE, developers can retrieve raw multi-touch data from the input buffer through a WM_INPUT message. These messages contain all of the raw data values before they are passed to WISPTIS for processing. Raw data can be intercepted and extracted in the client. These values can be divided into two categories: Primary Values and Additional Values. The primary values, supported by Windows 7 Touch (ID, State, X, and Y), can be re-injected for WISPTIS to process while the additional values can be stored for later use. However, the problem with this approach is that the data in the payload becomes fragmented and must be put back together again later in the pipeline. Reconstructing the payload requires matching the additional values with the primary ones after WISPTIS has processed them.

This solution can add unintentional latency to the pipeline and cause a bottleneck in the system. This could introduce a bias against HID to make both payloads of similar size. Another solution is to take a measurement in the code that receives WM_INPUT messages. However, since they deal with raw device data, it is unlikely that these messages are used by mainstream APIs.

8.2.5 Timestamp Matching

Matching timestamps for the Hold interaction was challenging for both protocols. The coordinates in Hold never change, but X and Y are an important part of the matching criteria. The performance calculations in Eva could be inaccurate due to difficulties in matching stationary touches. A known issue exists when matching timestamps for benchmarks involving larger sized workloads. It was found during the manual data analysis that large numbers of duplicate messages were arriving in the client at the beginning and at the end of the interaction. This was happening for both protocols. However, it was believed that neither protocol was responsible for this. Instead it was believed that WISPTIS has a safety mechanism to account for jitter at the start and end of a touch interaction. [60] It does not assume that a tracker has sent the first set of update messages at the same coordinates as the down messages and does not assume a tracker has sent the last set of updates at the same coordinates as the proceeding up messages.

Eva had a hard time differentiating between duplicate messages and update messages at these points in the interaction because the Coordinates, States and IDs were identical. It therefore chooses the first available message. This may not the best choice to make because the first available candidate may or may not be the exact match, but knowing precisely which candidate to choose becomes challenging because they are all correct matches. Therefore, the latency calculations for the first and last set of update messages are incorrect for larger workloads because there are so many candidates to choose from.

8.2.6 Development Experience

The final technical challenge has to do with the overall development experience in the transport layer of a multi-touch software system. HID's true strength lies in its flexibility to allow developers to create their own custom payloads for devices. TUIO's strength lies in its flexibility allowing it to be used on multiple OS platforms. However, when directly compared with TUIO, HID does have a significant limitation. Writing a driver takes awhile for developers to master. Producing a fully functional driver, that is approved by Microsoft within a relatively short be period of time, is difficult to achieve. [60] The injection buffer in the WDK is not wrapped up in usable function calls like the OSC buffer is inside TUIO_CPP. Developers are forced to approach the injection process from a low-level programming perspective as they must use bit-wise operations to shift data into the injection buffer. Therefore, marked differences comparing the learnability of the WDK versus TUIO_CPP remain in spite of available learning aids.

8.3 Summary

This chapter discussed the results of the evaluation in more detail and gave a reason for TUIO's perceived performance degradation. Technical issues surrounding the implementation of a multi-touch benchmark harness and the timestamp matching phase of the data analysis were also presented. With the evaluation's set up and results explained it is time to re-iterate and answer all of the research questions outlined in Chapter One.

Chapter Nine: Conclusions

This thesis presented results from a performance evaluation comparing HID with TUIO. This chapter outlines the contributions of this research and avenues for further work in the area of multi-touch software system performance.

9.1 Contributions

The primary goal of this work was to determine which protocol yields the lowest latency to multi-touch software systems through first research question: Is HID faster than TUIO? The results from this evaluation indicate that TUIO yields higher latencies under strenuous conditions, but is comparable when workloads stay below 10 concurrent touches. The results show that there is little difference between both for smaller sized workloads involving single user multi-touch interactions. However, as the number of concurrent touches and concurrent users rise, HID yields lower latencies than TUIO because of large amounts of redundant data messages holding up the pipeline in the TUIO Server. TUIO developers could focus their efforts on minimizing congestion by looking at previous work that has been done in the past with other systems that rely on UDP. Ultimately, TCP might prove to be a better choice since it contains built-in congestion control mechanisms. However, this would have to be examined more closely since the mechanisms that help ensure data transfer tend to slow systems down. [26]

The secondary goal of this work was to report on the process and the work involved in achieving an answer to this research question: Can traditional performance methodologies help in determining which protocol is faster? The answer is yes. It was shown that developers can apply Software Performance Engineering methodologies to multi-touch software system performance evaluations. Benchmarking can be used to help

developers locate bottlenecks in multi-touch system pipelines. This research also showed how to construct multi-touch workloads and where along the pipeline measurements can be taken to evaluate the transport layer.

For the third research question: Is a camera the only way to capture touch input? The answer is no, as it was found that a camera does not need to be connected to evaluate the performance of both protocols and touch input can be simulated directly inside of a multi-touch tracker. For the fourth research question: what is an efficient way of analyzing data from a multi-touch protocol evaluation? It was shown that, when the multi-touch benchmark harness prototype was activated, large amounts of data were logged for analysis. The process of how all of this data was analyzed was presented. It showed that the timestamp matching and performance calculation stages can be automated through the creation of a prototype performance management tool.

9.2 Future Work

More dependent variables should be added to future evaluations. Specifically, throughput should be added as it will help determine the messages making it through the TUIO socket per unit of time to further confirm the results. More observations per benchmark should be added in case unexplained side effects have contributed to misleading results. Furthermore, the peculiar behavior observed in Hold, Right and Diagonal warrants further investigation. It also makes sense to turn frame rate into an independent variable to see what impact different rates might have on the dependent variables.

The issue of jitter should be addressed in future evaluations involving simulated touch input. It could also make sense to turn jitter into an independent variable to see how

well both protocols handle both cases. Indeed, it was observed that TUIO_CPP did appear to be optimized for perfectly stationary (non-jittery) interactions such as tangible objects. Due to technical challenges payload will likely remain as a context variable even though comparing different sized payloads would remain a limitation future evaluations. Furthermore, the possibility of introducing a bottleneck that slows HID down is too grave for future evaluations as it would likely introduce a bias against HID. However, this solution can open interesting avenues for further work as it demonstrates that HID can still be used to carry customized device-specific payloads. The issues outlined in the data analysis section indicate a possible need for mainstream performance management tools if further work in this area is to proceed smoothly. Historical tools, such as Eva, can help developers evaluate their systems earlier in the development process and could play a vital role in producing higher quality multi-touch software systems.

High latency in multi-touch software systems is undesirable because it can impact the overall user experience on multi-touch devices. When transmission latencies go up, users notice slower responses to their actions as the device “feels” unresponsive and interactions lose their “natural smoothness”. Some of the latencies measured, which only cover a part of the overall pipeline, were well above the boundaries of perception. Software developers should expect negative user feedback on their TUIO applications as the number of concurrent users rises. Developers can use SPE methods to help locate performance bottlenecks. It is hoped that these methods are further applied to this area of research to help push it closer towards high-performance multi-touch computing.

References

1. Y. Kiriaty, L. Moroney, S. Goldshtein, and A. Fliess, *Introducing Windows 7 for Developers*. Microsoft Press, Redmond, WA, USA, 2010, pp. 101, 106-108, 129-131.
2. HP TouchSmart Blog. (November 2009) Touch Technology: Developed by you. [Online]. http://h20435.www2.hp.com/t5/HP-TouchSmart-Blog/Touch-Technology-Developed-by-You/ba-p/50110
3. USA Today. (November 2009) Windows 7 could hasten touch-screen computers [Online]. http://www.usatoday.com/tech/columnist/edwardbaig/2009-11-12-Baigtouchscreen12_CV_N.htm
4. SMART Product Drivers for Windows (2011) SMART Technologies Homepage [Online]. http://smarttech.com/us/Support/Browse+Support/Download+Software+Internal/SMART+Product+Drivers/SMART+Product+Drivers/SMART+Product+Drivers+for+Windows/SMART+Product+Drivers+10_7+SP1+for+Windows
5. J. Han, "Low-cost multi-touch sensing through frustrated total internal reflection," In *Proceedings of the 18th annual ACM symposium User Interface Software and Technology* (Seattle, WA, USA, October 23-27, 2005), ACM Press, New York, NY, 115-118.
6. L. Y. L. Muller, "Multi-touch displays: design, applications and performance evaluation," Universiteit van Amsterdam, Master's Thesis, 2008.
7. J. Schöning, P. Brandl, F. Daiber, F. Echtler, O. Hilliges, J. Hook, M. Löchtefeld, N. Motamedi, L. Muller, P. Olivier, T. Roth, and U. von Zadow. "Multi-Touch Surfaces: A Technical Guide," Technical Report TUMI0833: Technical Reports of the Technical University of Munich, (2008).
8. M. WU, and R. Balakrishnan, "Multi-Finger and Whole Hand Gestural Interaction Techniques for Multi-User Table Displays," In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Vancouver, BC, Canada, November 2-5, 2003), ACM Press, New York, NY, pp. 193-202.
9. C. North, T. Dwyer, B. Lee, D. Fisher, P. Isenberg, G. Robertson, and K. Inkpen, "Understanding Multi-touch Manipulation for Surface Computing," In *Lecture Notes in Computer Science, vol. 5727, Interact 2009*, Springer, pp. 236-249.
10. J. O. Wobbrock, M. R. Morris, and A. D. Wilson, "User-Defined Gestures for Surface Computing," In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, (Boston, MA, USA, April 4-9, 2009), ACM Press pp. 1083-1092.
11. J. Nielsen, *Usability Engineering*. Academic Press, San Diego, CA, USA, 1993, pp. 93-99, 192-194.
12. C. Loosley, F. Douglas, *High-Performance Client/Server: A Guide to Building and Managing Robust Distributed Systems*, Wiley & Son Inc., New York, USA, 1998, pp.

xxiv-xvii, 6-7, 37, 42-47, 58-59, 62-64, 66-67, 86-87, 110-111, 140-141, 147, 164-165, 173-174, 186-187, 192-195, 199, 349, 353, 468, 576-577, 579-580.

13. J. Shirazi, *Java Performance Tuning*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000, pp. 6-7, 10-17, 363, 399-402.
14. W. K. English, D. C. Engelbart, and M. L. Berman. "Display-Selection Techniques for Text Manipulation," In *IEEE Transactions on Human Factors in Electronics*, Vol. HFE-8, No. 1, March 1967, pp. 5-15.
15. C. Forlines, D. Wigdor, C. Shen, and R. Balakrishnan, "Direct-Touch vs. Mouse Input for Tabletop Displays," In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (San Jose, CA, April 28 – May 3, 2007), pp. 647-656.
16. W. Buxton, R. Hill, and P. Rowley, "Issues and Techniques in Touch-Sensitive Tablet Input," In *Proceedings of the Conference of Computer Graphics and Interactive Techniques* (San Francisco, CA, USA, July 22-26, 1985), Vol. 19, No. 3, pp. 215-223.
17. A. Esenther, C. Forlines, K. Ryall, and S. Shipman, "DiamondTouch SDK: Support for Multi-User, Multi-Touch Applications," *Proceedings of ACM Conference on Computer Supported Cooperative Work, (CSCW 2002 Demonstration)*.
18. SMART Table SDK (July 2010) SMART Technologies Homepage [Online]. http://downloads01.smarttech.com/media/products/sdk/smart-table-sdk-summary.pdf.
19. Microsoft Surface SDK 1.0 SP1 (2009) Surface 1.0 Platform Homepage [Online]. http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=15532
20. D. Garlan, and M. Shaw, "An Introduction to Software Architecture," CMU Software Engineering Institute Technical Report (Janurary 1994).
21. F. Echtler, and G. Klinker, "A multitouch software architecture," In *Proceedings of the 5th Nordic Conference on Human-Computer Interaction: Building Bridges* (Lund, Sweden, October 20-22, 2008). NordiCHI '08, ACM Press, New York, NY, USA, 463-466.
22. Infrared Cameras. (2012) Point Grey Research Homepage [Online]. http://www.ptgrey.com/products/index.asp
23. J. Axelson, *USB Complete Fourth Edition*. Lakeview Research LLC, Madison, WI, USA, 2009, 2, 180-182, 231, 277-278, 280-281, 286-287, 295, 317.
24. M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza, "TUIO: A Protocol for Table Based Tangible User Interfaces," In *Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation*, Vannes, France, 2005, Springer-Verlag, Berlin, Germany.
25. M. Wright, A. Freed, and A. Momen, "OpenSound Control: State of the Art 2003," In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression* (Montreal, QC, Canada, 2003). NIME '03, McGill University, Faculty of Music, Montreal, Canada, pp. 153-159.

26. J. F. Kurose, K. W. Ross, *Computer Networking: A Top-Down Approach*. 3rd Edition, Addison-Wesley, USA, 2005, pp. 4, 8-9, 85-86, 196-199.

27. M. MacDonald, *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4*. Apress 3rd edition, USA, 2010, 149-157.

28. Multitouch and gesture support on the Flash Platform (2009) Flash Developer Center [Online]. http://www.adobe.com/devnet/flash/articles/multitouch_gestures.html.

29. Multitouch Latency Reduction (2008) PeterKaptein: Technology, human nature and everything else [online]. http://peterkaptein.wordpress.com/2008/06/05/multitouch-latency-reduction/

30. Microsoft Research shows off touchscreen prototype with only 1 ms latency (2012) WMPoweruser: Windows Phone Tech Site Article [Online]. http://wmpoweruser.com/microsoft-research-shows-off-touchscreen-prototype-with-only-1ms-lag/

31. M. Montag, S. Sullivan, S. Dickey, and C. Leider, "A Low-Cost, Low-Latency Multi-Touch Table with Haptic Feedback for Musical Applications," In *Proceedings of the 2011 Conference on New Interfaces for Musical Expression* (Oslo, Norway, May 30 – June 1, 2011), University of Oslo and Norwegian Academy of Music, pp.8-13.

32. How do USB based-Touch Screens Communicate with Windows 7 (2009) TUIO Latency and Suitability Comment on NUI Group Community Forums [Online]. http://nuigroup.com/forums/viewthread/4159/

33. Kinect. (2010) TUIO Latency Comment on Multi-Touch Vista Discussion Board [Online]. http://multitouchvista.codeplex.com/discussions/235913

34. Limitations of TUIO? (2012) TUIO Latency Comment on NUI Group Community Forums [Online]. http://nuigroup.com/forums/viewthread/13644/

35. Tbeta, cross-platform multitouch (2008) Hack a Day Article [online]. http://hackaday.com/2008/11/28/tbeta-cross-platform-multitouch/

36. Latency performance tests for CCV (2010) NUI Group Community Forums Post [Online]. http://nuigroup.com/forums/viewthread/9530/

37. Microsoft Surface Simulator. (2009) Surface Simulator MSDN Article for the MS Surface SDK 1.0 SP1 [Online]. http://msdn.microsoft.com/en-us/library/ee804952(Surface.10).aspx

38. Multi-Touch Vista. (2009) Multi-Touch Vista Project Homepage [Online]. http://multitouchvista.codeplex.com/

39. Ethereal. (2006) Ethereal Network Protocol Analyzer Homepage [Online]. http://www.ethereal.com/

40. Wireshark. (2010) Wireshark Network Protocol Analyzer Homepage [Online]. http://www.wireshark.org/

41. U. Hinrichs, S. Carpendale, and S. D. Scott, "Evaluating the Effects of Fluid Interface Components on Tabletop Collaboration," In *Proceedings of the International Working Conference on Advanced Visual Interfaces* (Venezia, Italy, May 22-26, 2006), ACM Press, pp. 27-34.

42. M. Hancock, S. Carpendale, and A. Cockburn, "Shallow-Depth 3D Interaction: Design and Evaluation of One-, Two-, and Three-Touch Techniques," In *Proceedings of the SIGCHI conference on Human factors in computing system* (San Jose, CA, USA, April 30 – May 3, 2007), ACM Press, pp. 1147-1156.

43. P. Brandl, C. Forlines, D. Wigdor, M. Haller, and C. Shen, Combining and Measuring the Benefits of Bimanual Pen and Direct-Touch Interaction on Horizontal Interfaces, In *Proceedings of the International Working Conference on Advanced Visual Interfaces* (Napoli, Italy, May 28-30, 2008), ACM Press, pp. 154-161.

44. A. D. Wilson, S. Izadi, O. Hilliges, A. Garcia-Mendoza, and D. Kirk, "Bringing Physics to the Surface," In *Proceedings of the 21st annual ACM symposium on User interface software and technology* (Monterey, CA, USA, October 19-22, 2008), ACM Press, pp. 67-76.

45. Touchlib. (2008) TouchLib Homepage [Online]. http://www.nuigroup.com/touchlib/

46. Windows Pointer Device Data Delivery Protocol. (2011) MSDN Article [Online]. http://msdn.microsoft.com/en-us/library/windows/hardware/br259100.aspx

47. Community Core Vision. (2012) NUI Group Community Homepage [Online]. http://ccv.nuigroup.com/

48. Y. Satoshi, Y. Iwaya, and Y. Suzuki, "Investigation of System Latency Detection Threshold of Virtual Auditory Display," In *Proceedings of the 12th International Conference on Auditory Display* (London, UK, June 20-23, 2006), pp. 217-222.

49. Windows Touch Gestures (2010) Windows Touch Gesture List on MSDN [Online] http://msdn.microsoft.com/en-us/library/windows/desktop/dd940543(v=vs.85).aspx

50. D. Chappell, *Understanding .NET A Tutorial and Analysis*, Addison-Wesley, New York, 2002, pp. 128-130.

51. Windows SDK 7.1 (2010) Windows SDK Homepage [Online]. http://msdn.microsoft.com/en-us/library/ms717422.aspx

52. TouchInput Data Structure (2011) MSDN Article [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/dd317334(v=vs.85).aspx

53. Raw Input (Windows) (2011) MSDN Article on Accessing the Input Buffer [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/ms645536(v=vs.85).aspx

54. Windows Driver Kit 7600.16385.1 (2011) WDK Homepage [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487428

55. Device Class Definition for Human Interface Device (HID) (2001) USB.org HID Documentation for Developers [Online]. http://www.usb.org/developers/devclass_docs/HID1_11.pdf

56. HID Usage Tables (2001) USB.org HID Documentation for Developers [Online]. http://www.usb.org/developers/devclass_docs/Hut1_11.pdf

57. Introduction to Windows Touch (2008) MSDN Article [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487480

58. The Windows Touch Test Lab (2009) MSDN Article [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487482

59. Developer Enhancements to Windows Touch and Tablet PC (2010) MSDN Article [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487466

60. How to Design and Test Multitouch Hardware Solutions for Windows 7 (2010) MSDN Article [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487461

61. Digitizer Drivers for Windows Touch and Pen-Based Computers (2010) MSDN Article [Online]. http://msdn.microsoft.com/en-us/windows/hardware/gg487437

62. M. Kaltenbrunner and R. Bencina, "reacTIVision: A Computer-Vision Framework for Table-Based Tangible Interaction," In *Proceedings of the First International Conference on Tangible and Embedded Interaction* (Baton Rouge, LA, USA, Feb 15-17, 2007), pp. 69-74.

63. TUIO Protocol Overview (2012) TUIO Homepage [Online]. http://www.tuio.org/

64. TUIO Protocol 1.0 (2010) TUIO Homepage [Online]. http://www.tuio.org/?tuio10

65. TUIO Protocol 1.1 (2010) TUIO Homepage [Online]. http://www.tuio.org/?tuio11

66. TUIO Software (2011) TUIO Homepage [Online]. http://www.tuio.org/?software

67. Lego History Timeline (2012) Lego Homepage [Online]. http://aboutus.lego.com/en-us/lego-group/the_lego_history/

68. C. Severance and K. Dowd, *High Performance Computing*, Connexions, Rice University, Houston, TX, USA, 1998 & 2010, pp. 3-5.

69. SMART Table Interactive Learning Center (2009) SMART Table Product Homepage [Online]. http://smarttech.com/table

70. Evoluce Tabletop (2011) Evoluce Homepage [Online]. http://www.evoluce.com/en/hardware/multi-touch_table.php

71. How can we send Tablet Event with SendInput (2007) Developer Post on MSDN Development Center Forum [Online]. http://social.msdn.microsoft.com/Forums/sa/tabletandtouch/thread/55af1652-557f-4e75-af86-515baa273484

72. Simulate WM_TOUCH (2009) Developer Post on MSDN Development Center Forum [Online]. http://social.msdn.microsoft.com/Forums/en-US/tabletandtouch/thread/e5c87731-86e4-4c25-9278-86cb0dd4ff8a/

73. wm_touch and wm_gesture Message in VC MFC (2009) MS Groups Post [Online]. http://msgroups.net/microsoft.public.vc.mfc/wm_touch-and-wm_gesture-message/76715

74. How to: Simulate Mouse and Keyboard Events in Code (2009) MSDN Article [Online]. http://msdn.microsoft.com/en-us/library/ms171548.aspx

75. Simulating Touch Input in Windows Developer preview using Touch Injection API (2012) Microsoft TechNet Article [Online]. http://social.msdn.microsoft.com/Forums/en-US/tabletandtouch/thread/e5c87731-86e4-4c25-9278-86cb0dd4ff8a/

76. J. Rudd, K. Stern, and S. Isensee, Low vs. High Fidelity Prototyping Debate, In *Interactions* Vol. 3, No. 1 (January 1996). ACM Press, pp. 76-85.

77. J. Neter, W. Wasserman, G.A. Whitmore, *Applied Statistics*, Third Edition, Allyn and Bacon Inc., Boston, MA, USA, 1988, pp. 329, 481, 502-506.

78. T. Colton, *Statistics in Medicine*, Little, Brown and Company, Boston, MA, USA, 1974, pp. 54.

79. IBM SPSS Statistics (2012) IBM SPSS Product Homepage [Online]. http://www-01.ibm.com/software/analytics/spss/products/statistics/

80. R. B. Miller, "Response time in man-computer conversational transactions", In *Proc. Fall Joint Computer Conference* (December 9-11, 1968) ACM Press, 267-277.

Appendix A: Workload Calculations

# Updates per Touch	# of Touches	Calculation	Workload
10	1	(1×2) + (10×1)	12
	2	(2×2) + (10×2)	24
	5	(5×2) + (10×5)	60
	10	(10×2) + (10×10)	120
	20	(20×2) + (10×20)	240
	40	(40×2) + (10×40)	480
	80	(80×2) + (10×80)	960
	100	(100×2) + (10×100)	1200
50	1	(1×2) + (50×1)	52
	2	(2×2) + (50×2)	104
	5	(5×2) + (50×5)	260
	10	(10×2) + (50×10)	520
	20	(20×2) + (50×20)	1040
	40	(40×2) + (50×40)	2080
	80	(80×2) + (50×80)	4160
	100	(100×2) + (50×100)	5200
100	1	(1×2) + (100×1)	102
	2	(2×2) + (100×2)	204
	5	(5×2) + (100×5)	510
	10	(10×2) + (100×10)	1020
	20	(20×2) + (100×20)	2040
	40	(40×2) + (100×40)	4080
	80	(80×2) + (100×80)	8160
	100	(100×2) + (100×100)	10200

200	1	(1×2) + (200×1)	202
	2	(2×2) + (200×2)	404
	5	(5×2) + (200×5)	1010
	10	(10×2) + (200×10)	2020
	20	(20×2) + (200×20)	4040
	40	(40×2) + (200×40)	8080
	80	(80×2) + (200×80)	16160
	100	(100×2) + (200×100)	20200

Appendix B: Benchmark Harness Indexing Algorithm

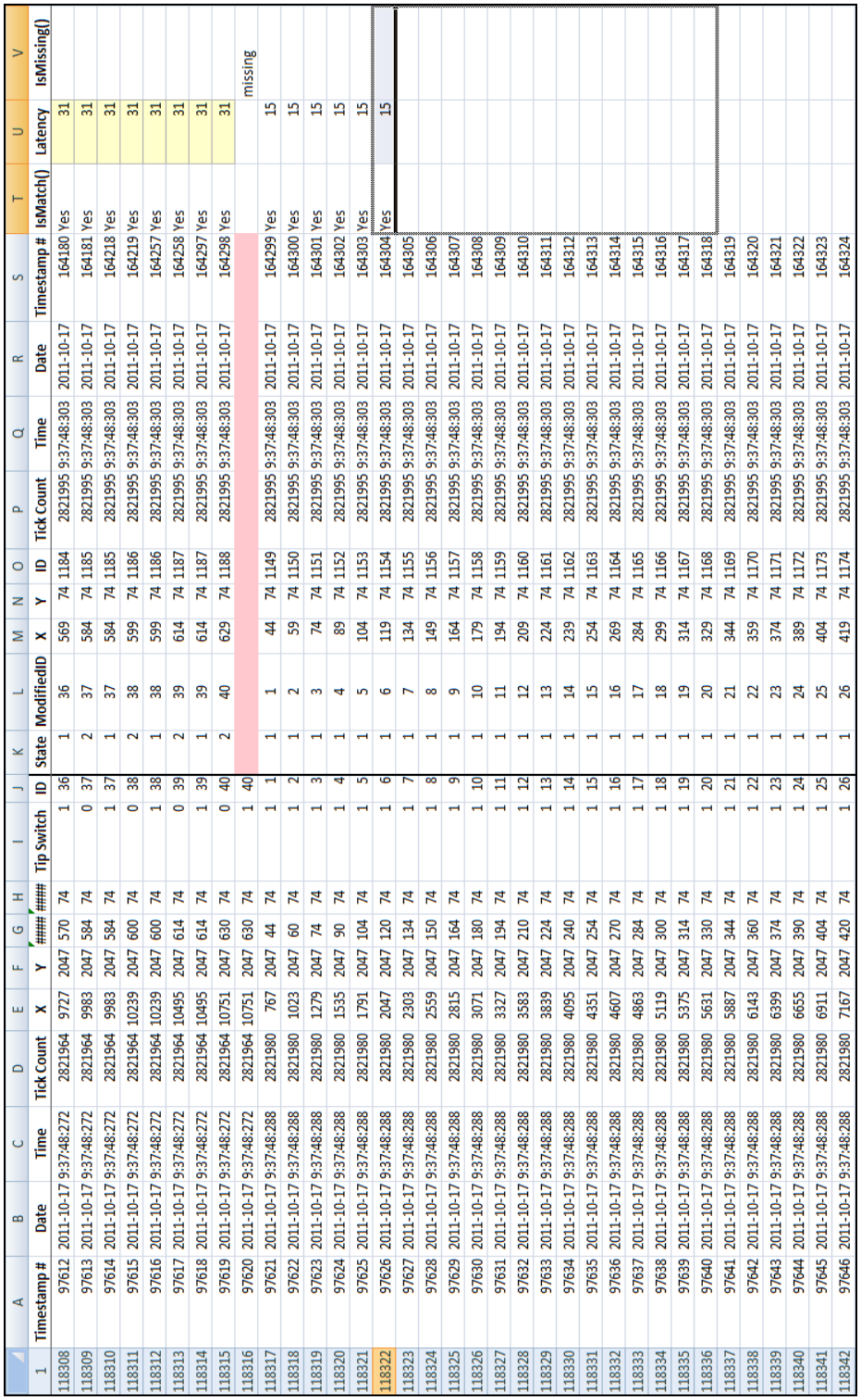
Assume the following Data Structures and Variables:

1. A pre-loaded Buffer containing all Bounding Boxes in Benchmark Harness.
2. A CurrentBufferIndex to track the current location in the Buffer.
3. A BenchmarkSuite containing a list of all benchmark parameters. Each parameter has a number of Bounding Boxes that will be sent for the Benchmark.
4. A CurrentBenchmarkIndex to track the current location the BenchmarkSuite.
5. A RunningCount to track the Number of Bounding Boxes already sent during Current Benchmark.
6. A BenchmarkSuiteComplete boolean, initially set to False, to mark the end of the transfer process.

For all incoming frames, execute the following instructions:

- If !BenchmarkSuiteComplete && RunningCount != NumBBoxes in CurrentBenchmark parameter, then Send data in Current Benchmark:
    - Define a list of CurrentTouches
    - For each touch defined in the Current Benchmark do the following:
        - Take Bounding Box from Buffer at CurrentBufferIndex and add to CurrentTouches List
        - Increment CurrentBufferIndex
    - Running count += NumTouches in CurrentBenchmark parameter
    - Send CurrentTouches list to the OS
- Otherwise if !BenchmarkSuiteComplete && RunningCount == NumBBoxes in CurrentBenchmark parameter, then Move onto the next Benchmark:
    - Send an EmptyList of touches to the OS
    - Put Tracker to Sleep for pre-defined duration // From Input Parameter File
    - Reset RunningCount to 0
    - Increment CurrentBenchmarkIndex
    - If CurrentBenchmarkIndex exceeds the size of the BenchmarkSuite, then:
        - Set BenchmarkSuiteComplete to True // Stopping Condition

Appendix C: Manual Data Analysis Screenshot



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Timestamp #	Date	Time	Tick Count	X	Y	####	####	Tip Switch	ID	State	ModifiedID	X	Y	ID	Tick Count	Time	Date	Timestamp #	IsMatch()	Latency	IsMissing()
118308	97612	2011-10-17	9:37:48:272	2821964	9727	2047	570	74		1 36	1	36	569	74	1184	2821995	9:37:48:303	2011-10-17	164180	Yes	31	
118309	97613	2011-10-17	9:37:48:272	2821964	9983	2047	584	74		0 37	2	37	584	74	1185	2821995	9:37:48:303	2011-10-17	164181	Yes	31	
118310	97614	2011-10-17	9:37:48:272	2821964	9983	2047	584	74		1 37	1	37	584	74	1185	2821995	9:37:48:303	2011-10-17	164218	Yes	31	
118311	97615	2011-10-17	9:37:48:272	2821964	10239	2047	600	74		0 38	2	38	599	74	1186	2821995	9:37:48:303	2011-10-17	164219	Yes	31	
118312	97616	2011-10-17	9:37:48:272	2821964	10239	2047	600	74		1 38	1	38	599	74	1186	2821995	9:37:48:303	2011-10-17	164257	Yes	31	
118313	97617	2011-10-17	9:37:48:272	2821964	10495	2047	614	74		0 39	2	39	614	74	1187	2821995	9:37:48:303	2011-10-17	164258	Yes	31	
118314	97618	2011-10-17	9:37:48:272	2821964	10495	2047	614	74		1 39	1	39	614	74	1187	2821995	9:37:48:303	2011-10-17	164297	Yes	31	
118315	97619	2011-10-17	9:37:48:272	2821964	10751	2047	630	74		0 40	2	40	629	74	1188	2821995	9:37:48:303	2011-10-17	164298	Yes	31	
118316	97620	2011-10-17	9:37:48:272	2821964	10751	2047	630	74		1 40												missing
118317	97621	2011-10-17	9:37:48:288	2821980	767	2047	44	74		1 1	1	1	44	74	1149	2821995	9:37:48:303	2011-10-17	164299	Yes	15	
118318	97622	2011-10-17	9:37:48:288	2821980	1023	2047	60	74		1 2	1	2	59	74	1150	2821995	9:37:48:303	2011-10-17	164300	Yes	15	
118319	97623	2011-10-17	9:37:48:288	2821980	1279	2047	74	74		1 3	1	3	74	74	1151	2821995	9:37:48:303	2011-10-17	164301	Yes	15	
118320	97624	2011-10-17	9:37:48:288	2821980	1535	2047	90	74		1 4	1	4	89	74	1152	2821995	9:37:48:303	2011-10-17	164302	Yes	15	
118321	97625	2011-10-17	9:37:48:288	2821980	1791	2047	104	74		1 5	1	5	104	74	1153	2821995	9:37:48:303	2011-10-17	164303	Yes	15	
118322	97626	2011-10-17	9:37:48:288	2821980	2047	2047	120	74		1 6	1	6	119	74	1154	2821995	9:37:48:303	2011-10-17	164304	Yes	15	
118323	97627	2011-10-17	9:37:48:288	2821980	2303	2047	134	74		1 7	1	7	134	74	1155	2821995	9:37:48:303	2011-10-17	164305			
118324	97628	2011-10-17	9:37:48:288	2821980	2559	2047	150	74		1 8	1	8	149	74	1156	2821995	9:37:48:303	2011-10-17	164306			
118325	97629	2011-10-17	9:37:48:288	2821980	2815	2047	164	74		1 9	1	9	164	74	1157	2821995	9:37:48:303	2011-10-17	164307			
118326	97630	2011-10-17	9:37:48:288	2821980	3071	2047	180	74		1 10	1	10	179	74	1158	2821995	9:37:48:303	2011-10-17	164308			
118327	97631	2011-10-17	9:37:48:288	2821980	3327	2047	194	74		1 11	1	11	194	74	1159	2821995	9:37:48:303	2011-10-17	164309			
118328	97632	2011-10-17	9:37:48:288	2821980	3583	2047	210	74		1 12	1	12	209	74	1160	2821995	9:37:48:303	2011-10-17	164310			
118329	97633	2011-10-17	9:37:48:288	2821980	3839	2047	224	74		1 13	1	13	224	74	1161	2821995	9:37:48:303	2011-10-17	164311			
118330	97634	2011-10-17	9:37:48:288	2821980	4095	2047	240	74		1 14	1	14	239	74	1162	2821995	9:37:48:303	2011-10-17	164312			
118331	97635	2011-10-17	9:37:48:288	2821980	4351	2047	254	74		1 15	1	15	254	74	1163	2821995	9:37:48:303	2011-10-17	164313			
118332	97636	2011-10-17	9:37:48:288	2821980	4607	2047	270	74		1 16	1	16	269	74	1164	2821995	9:37:48:303	2011-10-17	164314			
118333	97637	2011-10-17	9:37:48:288	2821980	4863	2047	284	74		1 17	1	17	284	74	1165	2821995	9:37:48:303	2011-10-17	164315			
118334	97638	2011-10-17	9:37:48:288	2821980	5119	2047	300	74		1 18	1	18	299	74	1166	2821995	9:37:48:303	2011-10-17	164316			
118335	97639	2011-10-17	9:37:48:288	2821980	5375	2047	314	74		1 19	1	19	314	74	1167	2821995	9:37:48:303	2011-10-17	164317			
118336	97640	2011-10-17	9:37:48:288	2821980	5631	2047	330	74		1 20	1	20	329	74	1168	2821995	9:37:48:303	2011-10-17	164318			
118337	97641	2011-10-17	9:37:48:288	2821980	5887	2047	344	74		1 21	1	21	344	74	1169	2821995	9:37:48:303	2011-10-17	164319			
118338	97642	2011-10-17	9:37:48:288	2821980	6143	2047	360	74		1 22	1	22	359	74	1170	2821995	9:37:48:303	2011-10-17	164320			
118339	97643	2011-10-17	9:37:48:288	2821980	6399	2047	374	74		1 23	1	23	374	74	1171	2821995	9:37:48:303	2011-10-17	164321			
118340	97644	2011-10-17	9:37:48:288	2821980	6655	2047	390	74		1 24	1	24	389	74	1172	2821995	9:37:48:303	2011-10-17	164322			
118341	97645	2011-10-17	9:37:48:288	2821980	6911	2047	404	74		1 25	1	25	404	74	1173	2821995	9:37:48:303	2011-10-17	164323			
118342	97646	2011-10-17	9:37:48:288	2821980	7167	2047	420	74		1 26	1	26	419	74	1174	2821995	9:37:48:303	2011-10-17	164324			

Appendix D: Eva's Matching Algorithm

Assume the following Data Structures and Variables:

1. Two lists of timestamps (BeginningList and EndingList) both containing one uniform (Windows 7 Touch) format for comparison.
2. 4 Benchmark Edge Indicies: 2 for the BeginningList (Starting and Ending) and 2 for the EndingList (Starting and Ending). They represent the locations, in both lists, where the Current Benchmark is located.
3. An empty MatchList which contains a list of all matched timestamps found. The MatchList also contains the MeanLatency, MissingCount, and DuplicateCount which are stored in the Benchmark's last MatchTimestamp.
4. A MatchTimestamp containing a BeginningTimestamp and its corresponding EndingTimestamp and the Latency between both timestamps. If a match could not be found, the EndingTimestamp is empty and the IndividualLatency is 0.
5. A CandidateList containing the indices of all possible matches in the EndingList for the current BeginningTimestamp.
6. Assume a MissingCount of 0.
7. Assume a RunningLatencySum of 0.

To Match the Timestamps in both lists, do the following:

- Locate the Starting and Ending indices in both lists for the first Benchmark
- For each timestamp in the BeginningList do the following:
    - For each timestamp in the EndingList do the following:
        - If IDs, States, and Positions in both timestamps ALL match, then add CurrentIndex (for EndingList) to CandidateList
        - Otherwise If end of benchmark has been reached in EndingList, then increment the MissingCount for benchmark
    - For each matched candidate do the following:
        - Take the first available timestamp in the Candidate List
    - Clean CandidateList
    - Add the data in both timestamps to a MatchTimestamp
    - Calculate the Individual Latency (EndingTimestampTickCount - BeginningTimestampTickCount) and store result in MatchTimestamp
    - RunningLatencySum += MatchTimestampLatency
    - If the end of Current Benchmark has been reached in BeginningList, then conduct the following end of benchmark operations
        - Calculate MeanLatency for Benchmark
        - Calculate DuplicateCount for Benchmark
        - Store MeanLatency, MissingCount and DuplicateCount in MatchTimestamp
        - Locate Starting and Ending indices in both lists for the next Benchmark
        - Reset MissingCount, DuplicateCount, RunningLatencySum to 0
    - Add MatchTimestamp to MatchList
- Return MatchList

To Locate the Ending Index of a Benchmark of either list, pass the list and CurrentLocationIndex in the list and do the following:

- For each Timestamp in the list, do the following:
    - If the CurrentLocationIndex + 1 < Size of TimestampList, then:
        - If CurrentTimestamp's State == TouchUp AND NextTimstamp's State == TouchDown
            - Return Index
    - Otherwise: // The end of the TimestampList has been reached
        - Return Index

To Calculate the MeanLatency use the Benchmark Edge Indices for the BeginningList, and do the following:

- Calculate LatencyCount (((BeginningListEndingIndex + 1) - BeginningListStartingIndex) – MissingCount)
- Calculate MeanLatency (RunningLatencySum / LatencyCount)

To Calculate DuplicateCount use the Benchmark Edge Indices in both lists, MissingCount, and do the following:

- (((EndingListEndingIndex + 1) - EndingListStartingIndex) - ((BeginningListEndingIndex + 1) - BeginningListStartIndex)) + MissingCount

Appendix E: Average Latencies in Milliseconds

Intended Workload		Hold		Down		Right		Diagonal	
		HID	TUIO	HID	TUIO	HID	TUIO	HID	TUIO
10	1	91.00	103.17	7.83	15.60	12.11	13.78	10.90	7.80
	2	0.62	0.00	0.58	0.00	0.00	2.31	0.62	0.00
	5	0.23	0.74	0.25	1.92	0.25	0.49	0.23	0.74
	10	0.12	17.57	0.12	22.99	0.12	24.23	0.12	20.75
	20	0.06	6.26	0.12	44.18	0.06	50.07	0.06	19.67
	40	0.03	25.99	0.03	64.94	11.37	51.75	0.03	67.10
	80	0.35	42.73	1.39	100.61	6.54	90.39	0.63	91.40
	100	3.06	26.68	3.80	105.49	3.19	111.29	4.61	110.19
50	1	412.28	424.02	2.40	1.00	1.27	1.92	0.94	1.00
	2	0.15	0.00	0.15	0.00	0.15	0.00	0.15	0.00
	5	0.06	0.12	0.06	0.38	0.06	0.12	0.06	0.18
	10	0.03	4.34	0.00	7.86	0.03	8.90	0.03	12.13
	20	0.02	5.45	0.02	26.93	0.01	25.34	0.02	22.31
	40	0.01	9.26	0.01	50.39	0.01	47.13	0.01	50.18
	80	0.93	18.26	0.94	95.23	0.91	100.58	1.51	97.61
	100	2.85	76.99	3.91	122.96	2.63	129.68	3.88	139.10
100	1	816.76	813.42	0.46	0.49	0.63	0.64	0.31	0.67
	2	0.07	0.69	0.08	0.17	0.07	0.15	0.07	0.40
	5	0.00	2.10	0.03	1.72	0.03	2.20	0.03	2.99
	10	0.16	4.98	0.15	12.28	0.13	13.18	0.29	12.81
	20	0.34	5.26	0.22	25.23	0.15	27.45	0.12	28.55
	40	0.32	3.89	0.30	54.44	13.30	58.28	0.44	58.26
	80	2.06	15.61	2.52	100.55	3.08	112.99	2.47	110.41
	100	4.79	636.33	5.62	177.35	4.18	270.63	5.49	169.74

200	1	1614.00	2164.26	0.16	0.41	0.24	0.47	0.31	0.68
	2	0.08	6.90	0.08	0.00	0.04	0.49	0.07	0.38
	5	0.02	21.96	0.06	3.77	0.05	3.32	0.03	3.18
	10	0.07	28.13	0.12	14.86	0.10	17.25	0.08	15.64
	20	0.19	13.86	0.16	29.61	0.17	31.72	0.24	29.23
	40	1.98	4.70	1.77	67.04	1.76	69.33	1.54	69.16
	80	4.66	24.15	4.59	122.23	18.23	132.37	4.44	125.60
	100	8.08	842.81	8.00	14586	14.11	18022	8.66	1110.2

Appendix F: Missing Message Percentages

Intended Workload		Hold		Down		Right		Diagonal	
		HID	TUIO	HID	TUIO	HID	TUIO	HID	TUIO
10	1	0.0%	0.0%	8.3%	25.0%	33.3%	33.3%	25.0%	25.0%
	2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	5	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	20	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	40	0.0%	0.0%	0.0%	0.0%	1.3%	0.0%	0.0%	0.0%
	80	0.0%	0.0%	0.0%	0.5%	0.0%	0.2%	0.0%	2.7%
	100	0.0%	12.3%	0.2%	23.5%	0.0%	18.1%	0.2%	11.8%
50	1	0.0%	0.0%	1.9%	11.5%	7.7%	7.7%	5.8%	11.5%
	2	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	5.8%
	5	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	5.8%
	10	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	5.8%
	20	0.0%	0.0%	0.0%	6.4%	0.0%	0.0%	0.0%	5.8%
	40	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	5.8%
	80	0.0%	0.0%	0.0%	5.9%	0.0%	5.3%	0.0%	6.0%
	100	0.0%	32.2%	2.0%	42.7%	2.0%	10.6%	2.0%	23.8%
100	1	0.0%	0.0%	2.9%	8.8%	3.9%	3.9%	2.9%	8.8%
	2	0.0%	0.0%	0.0%	6.4%	0.0%	1.5%	0.0%	6.9%
	5	0.0%	0.0%	0.0%	6.7%	0.0%	2.2%	0.0%	8.8%
	10	0.0%	0.0%	0.0%	9.5%	0.2%	3.5%	0.0%	14.7%
	20	0.0%	0.0%	0.0%	6.1%	0.0%	3.6%	0.1%	8.7%
	40	0.0%	0.0%	0.0%	5.9%	0.1%	1.1%	0.0%	5.9%
	80	0.0%	0.0%	3.0%	6.0%	2.0%	2.2%	2.0%	6.9%
	100	0.0%	0.0%	6.9%	25.1%	3.0%	50.2%	5.9%	18.7%

200	1	0.0%	0.0%	1.5%	6.9%	2.0%	2.0%	1.5%	8.9%
	2	0.0%	0.0%	0.0%	8.2%	0.0%	5.7%	0.0%	7.7%
	5	0.0%	0.0%	0.0%	19.1%	0.0%	18.0%	0.0%	20.8%
	10	0.0%	0.0%	0.0%	18.8%	0.0%	22.1%	0.0%	20.9%
	20	0.0%	0.0%	0.0%	21.4%	0.0%	19.5%	0.0%	21.6%
	40	0.0%	0.0%	1.5%	11.1%	0.5%	9.7%	1.5%	8.7%
	80	0.0%	0.0%	4.5%	6.0%	4.5%	3.5%	6.4%	43.1%
	100	0.0%	28.0%	10.4%	50.7%	8.2%	55.5%	8.9%	58.1%

Appendix G: Duplicate Message Counts

Intended Workload	Hold		Down		Right		Diagonal	
	HID	TUIO	HID	TUIO	HID	TUIO	HID	TUIO
12	0	0	0	8	0	5	0	7
24	0	66	0	64	0	58	0	60
60	15	479	15	479	15	519	15	475
120	80	954	80	974	80	988	80	970
240	360	1075	360	956	360	912	360	1107
480	1520	2297	1520	2059	1526	2470	1520	2158
960	6240	8105	6240	8793	6240	9434	6240	8915
1200	9800	25995	9702	33361	9800	20534	9702	19736
52	0	0	0	47	0	47	0	50
104	0	300	0	304	0	276	0	308
260	15	2179	15	2304	15	2296	15	2324
520	80	3996	80	4070	80	4201	80	4233
1040	360	3507	360	3628	360	3784	360	3539
2080	1520	4697	1520	4969	1520	6043	1520	5594
4160	6240	16790	6240	18147	6240	19509	6162	17803
5200	9600	103253	9702	42944	9702	76898	9702	116460
102	0	0	0	101	0	97	0	102
204	0	574	0	605	0	593	0	600
510	15	4059	15	4270	15	4167	15	4054
1020	80	7019	80	7375	72	7004	80	6506
2040	340	5592	360	5933	360	6288	342	6378
4080	1520	9170	1520	8723	1524	9676	1520	10118
8160	6080	33666	6162	29385	6162	30456	6162	29971
10200	9200	311749	9702	173091	9702	229655	9702	159419

202	0	0	0	206	0	192	0	198
404	0	794	0	1179	0	1077	0	1185
1010	15	6107	15	7048	15	6888	15	7124
2020	80	9821	80	12113	80	10961	80	12182
4040	360	9890	360	10712	360	10857	360	11882
8080	1440	16545	1482	15787	1520	16724	1482	18522
16160	5200	53764	6162	55703	6164	56896	6162	100743
20200	8709	347814	9702	568001	9766	651692	9702	568869