Providing Acceleration for Mobile Gaming as A Service

by

Elliott (Jiaqi) Wen

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2016

Abstract

In recent years, the mobile gaming industry has made rapid progress. Developers are now producing numerous mobile games with increasingly immersive graphics. However, these resource-hungry applications inevitably keep pushing well beyond the hardware limits of mobile devices. The limitations causes two main challenging issues for mobile game players. First, limited computational capabilities of smart devices are preventing rich multimedia applications from running smoothly. Second, the minuscule touchscreens impede the players from smoothly interacting with devices as they can do with PCs.

This thesis aims to address the two issues. Specifically, we implement two systems, one for the application accelerations via offloading and the other for alternative interaction approach for mobile gaming. We identify and describe the the challenging issues when developing the systems and describe our corresponding solutions.

Regarding the first system, it is well recognized the performance of GPUs on mobile devices is the bottleneck of rich multimedia mobile applications such as 3D games and virtual reality. Previous attempts to tackle the issue mainly mirgate GPU computation to servers residing in remote datacenters. However, the costly network delay is especially undesirable for highly-interactive multimedia applications since a crisp response time is critical for user experience. In this thesis, we propose GBooster, a system that accelerates GPU-intensive mobile applications by transparently offloading GPU tasks onto neighboring multimedia devices such as SmartTV and Gaming Consoles. Specifically, GBooster intercepts and redirects system graphics calls by utilizing the Dynamic Linker Hooking

technique, which requires no modification of the apps and mobile systems. Besides, GBooster intelligently switches between the low-power Bluetooth and the high-bandwidth WiFi interface to reduce energy consumption of network transmissions. We implemented the GBooster on the Android system and evauluate its performance. The results demonstrate that GBooster can boost applications' frame rates by up to 85%. In terms of power consumption, GBooster can achieve 70% energy saving compared with local execution.

Second, we investigate the potential of built-in mobile device sensors to provide an alternative interaction approach for mobile gaming. We propose UbiTouch, a novel system that extends smartphones with virtual touchpads on desktops using built-in smartphone sensors. It senses a user's finger movement with a proximity and ambient light sensor whose raw sensory data from underlying hardware are strongly dependent on the finger's locations. UbiTouch maps the raw data into the finger's positions by utilizing Curvilinear Component Analysis and improve tracking accuracy via a particle filter. We have evaluate our system in three scenarios with different lighting conditions by five users. The results show that UbiTouch achieves centimetre-level localization accuracy and poses no significant impact on the battery life. We envisage that UbiTouch could support applications such as text-writing and drawing.

Acknowledgments

Undertaking this degree has been truly life-changing experience for me and it would not have been possible for me to do so without the valuable support and guidance I received from many people.

First and foremost, I would like to express my profound gratitude to my supervisors Prof Winston Seah and Dr Bryan Ng for all the support and encouragement they gave me. During my quest to the master degree, they have constantly provided me with systematic research guidance and valuable financial support, which enable me to obtain the degree. Their kindness and patience towards me gave me an indelible impression that having such an invaluable opportunity to learn from them must be one of the most fortunate things in my life.

My thanks also go to Miss Victoria Wong, who has been the stalwart supporter constantly regardless of my fickle moods and immatureness. Moreover, I wish to acknowledge support from Miss Madelyn Ma, without whom I could have obtained this degree much earlier. Finally, my deepest appreciation belongs to my family members for their constant support and encouragement. Without them, I would have never been able to go so far.

Contents

1 Introduction **1**

1.1 Problem Statements 2

1.1.1 Limited Computational Resources and Battery Life . 2

1.1.2 Limited Smartphones' Input Space 3

1.2 Research Framework 4

1.3 Research Questions and Contributions 6

1.4 Organization of the Thesis 8

2 Literature Review **9**

2.1 Literature Review on Mobile Application Acceleration . . . 9

2.1.1 Mobile Cloud Computing and Mobile Cloud Gaming 9

2.1.2 Offloading Methods 12

2.1.3 Energy Efficiency 14

2.2 Literature Review on Mobile Interaction 16

2.2.1 Extending Smartphones' Input Space 16

2.2.2 Built-in Sensor Based Input Approaches 17

2.2.3 Virtual Touch Surface 17

3 GBooster **19**

3.1 Motivations 19

3.2 System Overview 22

3.3 Enable GPU Task Offloading 24

3.3.1 Intercepting and Rewriting OpenGL ES Functions . . 24

3.3.2 Forwarding Graphics Commands 26
3.3.3 Executing Commands and Retrieving Results 27
3.4 Energy-saving Network Transmission 27
3.4.1 Eliminating Redundancy of Network Traffic 28
3.4.2 Enabling Transmission via Low-Power Interfaces . . 29
3.5 Harnessing Capacities from Multiple Service Devices 32
3.5.1 Parallelizing GPU Computation 33
3.5.2 Maintaining State Consistency among Devices 34
3.5.3 Assigning Requests to Devices 35
3.6 System Evaluation 35
3.6.1 Sample Games and Devices 35
3.6.2 Application Acceleration 36
3.6.3 Power Saving 39
3.6.4 Multiple Devices 40
3.6.5 Performance on non-gaming apps 42
3.6.6 Comparison with Cloud-based Solutions 43
3.6.7 System Overhead 43
3.7 Conclusion 44

4 UbiTouch **45**
4.1 System Overview 45
4.2 A Peek at Trailing a Finger Using ALS and PS 47
4.3 Obtaining Finger Locations from Raw Sensor Data 50
4.4 Improving Tracking Accuracy via a Hidden Markov Model . 55
4.5 Touch Action Detection 57
4.6 Runtime Calibration and Adaptation 60
4.7 System Evaluations 61
4.7.1 Accuracy of Finger Tracking 62
4.7.2 Accuracy of Touch Action Detection 65
4.7.3 Power Consumption 67
4.7.4 Application Tests 67

4.8 Conclusion 68

5 Conclusions and Suggestions for Future Research 69

5.1 Conclusions 69

5.2 Suggestions for Future Research 70

5.2.1 GBooster 71

5.2.2 UbiTouch 72

Chapter 1

Introduction

The mobile gaming industry is booming rapidly. According to Global Games Market Report [25], mobile gamers worldwide will generate a total of $99.6 billion in revenues in 2016, up 8.5% compared to 2015. For the first time, mobile gaming will take a larger share than personal computer (PC) with $36.9 billion, up 21.3% globally and mobile games have become the most important digital platform for gamers and publishers alike. The game developers are producing mobile games with increasingly high-definition 3D graphical interfaces and visual effects. On the other hand, mobile gamers become more demanding and continue to increase their expectations.

Last decades have witnessed tremendous increase in mobile device performance. Smart devices are now equipped with the increased processing power, better battery life, vastly improved networking speeds, and sensitive touch-screens. According to the survey [26], the mainstream central process units (CPUs) of mobile devices provide a 50 times increase in performance over chips from five years ago, consuming 75 percent less energy than chips from three years ago. However, despite the promising progress, when handling the tasks formerly reserved for the desktop PCs, especially gaming, the mobile devices are still facing two main challenges. The first challenge lies in limitations of computational resources and battery life. While the other one arises from the limited size of touchscreens, which prevent players from smoothly interacting with devices as they can do with PCs.

1.1 Problem Statements

In this section, we will elabroate the two main challenging issues related to mobile gaming, which motivate this work.

1.1.1 Limited Computational Resources and Battery Life

Multimedia applications such as gaming and augmented reality are proliferating in mobile devices nowadays. As current devices are still facing constraints of processing capabilities and battery power due to their minuscule physical sizes, the resource-hungry applications are inevitably pushing the limit of the devices, causing the applications' running in a low frame rate and leading to a short battery lifetime.

Considerable research works such as MAUI [40] and CloneCloud [37] attempt to alleviate these issues by offloading CPU computation tasks to cloud. However, the existing systems barely benefit mobile gaming applications. It is due to that although a multi-media mobile application generally involves CPU and GPU computation, it is often the case that the application is limited by GPU performance. To bridge the gap, a small number of studies focusing on offloading GPU tasks have been carried out. For instance, OnLive [13] and G-Cluster [46] feature a remote-rendering architecture, in which, multimedia applications run in cloud servers and the screen-shots of the applications are delivered to users through the Internet. Meanwhile, the users' control inputs are transmitted and replayed in the servers. Recently, a more sophisticated Component-based offloading architecture has been proposed in [35]. Instead of executing entire applications in cloud servers, it distributes the application's independent components to either cloud servers or local devices for execution, as determined by the devices' current workloads and their network connectivities.

Despite the promising results obtained, these systems mentioned above possess certain limitations. In particular, the existing platforms only allow users to use a limited number of applications that have been deployed in the cloud beforehand. Besides, as the cloud tends to be remotely located, the network delay incurred by transferring screen-shot frames in remote-rendering platforms can be very costly. This is especially undesirable for some highly-interactive applications such as action games involving multiples players, where a crisp response time is critical to the user's experience. Though the Component-based architecture may alleviate the latency issue by avoiding screen-shot transmission, it causes unbearable burden to developers who are required to modify and recompile source codes for legacy software.

It can be seen that there exists a research gap for acclerating multimedia applications and reduce their energy consumption via offloading.

1.1.2 Limited Smartphones' Input Space

Unlike desktop computers which mainly use a mouse for input, mobile devices and tablets mainly use the touchscreens as the input source. Although the touchscreen provides a straightforward way for users to interact with the devices, it indeed possesses several significant drawbacks compared with the mouses, which are listed in table.1.1. These limitations may impede players from enjoying the smooth interaction with the mobile games.

This difficulty has motivated various researchers to explore alternative interaction technologies for mobile devices. Notable systems such as RF-IDraw [75] and TypingRing [66] have been able to achieve very good tracking accuracy. However, they rely heavily on peripheral devices such as extra RFID tags and cameras, which may significantly limit the porta-

Table 1.1: Comparsion between mouses and touchscreens

	Mouse	Touchscreen
Precision	High	Low (fat-finger problem)
Obscures view of screen	No, thus allowing for continuous visual feedback	Yes
Suitable for huge screens	Yes	No
Number of controls	3: left/right button, scroll wheel	1

bility of smart devices. To address the portability issue, researchers have attempted to take advantage of built-in sensors to achieve the same purpose. For example, Finger-in-Air [64] utilizes a smartphone camera to estimate users' finger gestures. UbiK [76] leverages microphones on a smartphone to detect keystroke locations, enabling text-input on a sheet of paper where a keyboard outline is printed. UbiK [76] is solely designed for distinguishing different keys a user presses. It may not support applications that require continuous finger tracking, for instance, handwriting input or drawing. Despite the promising results obtained, vision-based systems [64] require the camera module to be continuously switched on. This could drain the battery quickly and its performance may be negatively impacted by the low-light environment.

Therefore, how to utilize built-in low-power sensors to constantly localize and track a users finger, enabling touchpad-like input experience to better support mobile gaming remains unsolved.

1.2 Research Framework

In this thesis, we propose a framework that aims to address the two challenging issues mentioned above. Specifically, the framework aims to the following objectives.

1. Run GPU-intensive multimedia applications. The framework enables smart devices to run a multimedia application with high demanding requirement of graphics processing, regardless of their hardware capabilities.

2. Extend battery life. The framework reduces energy consumption incurred by heavy GPU utilization, thus extend the battery life. It would be particularly useful when the battery is running low, but the users still want to use their devices for a longer time.

3. Provide smooth interaction experience for players. The framework aims to enhance smartphones with virtual touchpads through the built-in smartphone sensors to provide smoother interaction for mobile gaming.

In this thesis, we implement our framework with two main systems named *GBooster* and *UbiTouch* respectively. Specifically, to address the issue resulted from limited computational resources, we introduce GBooster, a system that accelerates multimedia mobile applications by seamlessly leveraging ambient computation capacities. GBooster utilizes a novel GPU computation offloading technique to accelerate GPU-intensive mobile applications. Without modification and deployment of the applications, this technique transparently offloads GPU tasks onto neighboring multimedia devices such as gaming consoles, personal computers, and Smart TVs. As the devices are located at the close physical proximity to users, the network connections tend to have tiny communication delays and a high bandwidth, thereby guaranteeing the user's experience for highly-interactive applications.

To tackle the issue resulted from the limited input space of mobile devices, we propose UbiTouch, a novel system that achieves the goal of augmenting smartphones with virtual touchpads by leveraging built-in smartphone sensors. Specifically, the smartphone's proximity sensor (PS) and ambient light sensor (ALS) are utilized to sense the movement. Meanwhile, the microphone and gyroscope sensors are used to detect touch actions such as tapping and dragging. These sensors are readily available in almost every smart device and of low-power consumption.

1.3 Research Questions and Contributions

In this thesis, we elaborate research issues of the two systems and demonstrate the corresponding solutions for them.

Regarding the GBooster, the practical implementation entails substantial challenges. First of all, we expect to accelerate every mobile game which may be implemented using different graphics engines or even different programming languages. It is considered to be challenging to propose a universal GPU task offloading technique for the apps. Secondly, Though offloading compute-intensive GPU tasks save considerable amounts of power, the extra energy cost incurred by the network transmission can still significantly drain a cell phone's battery. How to reduce energy overhead of network communication without degrading system performance remains a non-trivial issue. Finally, in a real-world environment where multiple users simultaneously access the services provided by a number of multimedia devices, how to efficiently incorporate distributed computation capacities of the devices and meet the requirement of the multi-users is not a simple task.

In this thesis, we develop practical solutions to cope with the above challenges. To enable offloading for all multimedia applications, we intercept system graphics calls and redirect them to nearby devices. This approach requires no modification and deployment of the apps beforehand. To lower down the energy overhead of offloading, GBooster first eliminates redundant data in network traffic and intelligently switches between the high-throughput high-power WiFi interface and the low-throughput low-power Bluetooth interface based on the traffic volume. To handle requests from multiple users efficiently, GBooster adopts a task allocation mechanism that takes requests' workload and priority into consideration. We will discuss the details in Chapter .3.

As for the second system UbiTouch, the key challenge lies in achieving fine-grained localization and tracking using sensors that are originally designed to provide coarse-grained information. An ALS is typically used to measure the luminance of ambient lighting [59]. Readings from this sensor are not directly related to a finger's movement. Similarly, a PS is designed to detect the presence of nearby objects and merely reports binary distance values representing "near" or "far" to smartphone operating systems [47], which does not benefits the tracking. In this paper, we take a closer examination of the internal structures and underlying mechanisms of the sensors [30] and uncover the potential for these sensors to be used to achieve our goal.

Specifically, we implement UbiTouch as a system with three key components: finger tracking, touch action detection, and run-time calibration. We introduce a finger localization framework that adapts Curvilinear Component Analysis for mapping the raw sensor data to two-dimensional coordinates. It then estimates the finger movement trajectories by utilizing a particle filter which integrates the historic finger locations to boost the tracking accuracy. We also propose a touch action detection algorithm to capture touch tapping and dragging events on ordinary surfaces by applying hypothesis testing techniques on audio signals from microphones. To provide reliable results, we leverage the built-in gyroscope to combat the environment noises. Additionally, we design a runtime calibration mechanism that takes advantage of extra run-time user feedback data to re-calibrate the tracking algorithm and combat minor variation of background light conditions. The details will be elaborated in Chapter. 4.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents a literature review for mobile gaming. Chapter 3 elaborates the design of the GBooster prototype. Chapter 4 demonstrates the implementation of UbiTouch system, which serves as a human computer interface for mobile gamers. Finally, we conclude the thesis and discuss the directions of future works in Chapter 5.

Chapter 2

Literature Review

This chapter provides an overview of mobile application acceleration and a review of mobile interaction technologies.

2.1 Literature Review on Mobile Application Acceleration

We start this section with a short overview of Mobile Cloud Computing, which is the core technique of mobile application acceleration. We then narrow our focus on Mobile Cloud Gaming, which has drawn a great amount of attention from various researchers recently. We elaborate existing works in several research areas such as offloading techniques, energy efficiency, network communication, and security and privacy.

2.1.1 Mobile Cloud Computing and Mobile Cloud Gaming

Mobile devices are increasingly becoming an essential part of human life. Mobile users nowadays are offered various services from mobile applications in various categories such as entertainment, health, games, business, social networking, travel and news. Nevertheless, the mobile devices are still facing many challenges in their resources (e.g., battery life, storage, and bandwidth) and communications (e.g., mobility and security). The limited resources significantly impede the improvement of service qualities.

Recently, Cloud computing (CC) [88] has been widely recognized as the next generation computing infrastructure. CC allows users to utilize infrastructure (e.g., servers, networks, and storages), platforms (e.g., middleware services and operating systems), and software (e.g., application programs) provided by cloud providers at low cost. Moreover, CC entitles users to utilize resources elastically in an on-demand fashion. Therefore, applications can be rapidly provisioned and released with the minimal management efforts.

With the explosion of mobile applications and the support of CC for a variety of services for mobile users, mobile cloud computing (MCC) [43] is introduced as an integration of CC into the mobile environment.

Definition of MCC

Formally, MCC is defined by The MCC forum [11] as follows.

> Mobile cloud computing at its simplest, refers to an infrastructure where both the data storage and data processing happen outside of the mobile device. Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing applications and MC to not just smartphone users but a much broader range of mobile subscribers.

In short, MCC provides mobile users with the data processing and storage services in clouds. The mobile devices do not need a powerful specification since all the complicated computation can be done in the cloud.

Advantages of MCC

MCC brings about a great number of advantages as follows.

1. Improving processing power. Mobile cloud computing benefits in reducing the running time of compute-intensive applications performed on the limited-resource devices. With MCC, the complicated computation can be moved to cloud with powerful capabilities and processed efficiently.

2. Extending battery lifetime. Battery is one of the main constraint of mobile devices. Offloading technique is proposed to migrate the complicated computations from resource-limited devices to resourceful machines in clouds. It eliminates a long application execution time which results in a large amount of power consumption.

3. Enhancing data storage capacity. Storage capacity is another constraint for mobile devices. MCC could enable mobile users to store and access the data on the cloud. Therefore, the users can save considerable amounts of storage space of there smart devices.

4. Dynamic provisioning. MCC provides a flexible way for elastic on-demand provisioning of resources on a fine-grained, self-service basis. It enables mobile users to run their applications without advanced reservation of resources.

5. Scalability. Owning to dynamic provisioning, the deployment of mobile applications can scale to meet the unpredictable user demands. Service providers can freely expand or shrink an application service.

MCC Applications

MCC supports various kinds of mobile applications such as mobile commerce, mobile learning and mobile health-care. Take mobile commerce as an example. Conventional m-commerce applications face various challenges such as low network bandwidth and and security. By leveraging MCC techniques, these issues can be greatly addressed. For instance, the research work [84] proposes a 3G E-commerce platform based on CC. This paradigm combines the advantages of both third generation (3G) network and CC to increase data processing speed and security level.

Among all these applications, mobile cloud gaming has drawn a great amount of interests from various research. Mobile gaming enables mobile devices to run rich media applications, which requires computational power far beyond the capabilities of a smart device. In the following chapter, we will focus on research questions stemming from mobile cloud gaming and elaborate the related existing works.

2.1.2 Offloading Methods

The essential operation in any mobile cloud computing would be the offloading of jobs that perform on the resource constrained mobile devices to the cloud. Due to issues such as the physical distance between the mobile device and the cloud as well as the heterogeneity of the underlying systems, various research works have attempted to address this in a variety of ways. Current research discusses offloading methods in two main directions; Client Server Communication methods and Virtualization [44].

Client Server Communication. In this diagram, process offloading is achieved via protocols such as Remote Procedure Calls (RPC), Remote Method Invocation (RMI) and Sockets.

For instance, Spectra [45] leverage RPC to invoke functionality in remote and local Spectra servers. When a mobile device needs to offload an application, the Spectra client consults a database that stores information about Spectra servers such as their current availability, CPU load. These servers are pre-deployed with application code acting as services. Developers need to manually partition the applications by specifying which methods might be candidates for offloading. Spectra decides whether a method will be offloaded at runtime depending on the available resources.

The Cuckoo framework [57] proposes a system to offload mobile device applications onto a cloud via a Java stub/proxy model. Cuckoo can be deployed on any resource that runs the Java Virtual machine. The objectives of the system are to enhance performance and reduce battery usage. To use Cuckoo, the applications need to be modified so that the application supports remote execution as well as local execution.

Virtual machine (VM) migration. VM migration refers to transferring the memory image of a VM from a source server to the destination server without stopping its execution [38]. In this diagram, the memory pages of the VM are copied without interrupting the OS or any of its applications, thereby providing an illusion of seamless migration. This methods enable offloading without requiring modification of legacy applications. However, VM migration is somewhat time consuming and the workload could prove to be heavy for mobile devices.

MAUI [40] utilizes a combination of VM migration and code partitioning in order to reduce energy consumption. Mobile applications are offloaded from phones to surrounding devices (i.e. local and remote servers). To use the system, developers must annotate which methods can be offloaded at the time of execution beforehand. MAUI automatically decides whether or not to offload these methods based on the current connectivity and workload.

Similarly, CloneCloud [37] also uses VM migration to offload part of their application workload to a resourceful server. Because the system relies on device clones, the mobile applications need not to be modified and there is no need of even annotating methods such as done in MAUI [40]. CloneCloud utilizes a cost model to analyze the migration cost and compares the cost against local execution.

Despite the promising results obtained, the systems mentioned above only consider migrating CPU-based computation rather than GPU-based tasks, which are the major components for most mobile games. Thus the existing systems barely benefit the GPU-intensive applications. To enable offloading for the applications, a number of architectures featuring offloading games have been proposed.

Remote Rendering Architecture. Considerable research works on cloud gaming such as OnLive [13] adopt a remote rendering architecture. Specifically, video games are executed in cloud servers and the video frames are transmitted to users through the Internet. At the same time, the players' inputs are delivered and relayed in the corresponding server. This approach enables the players to run sophisticated games regardless of their restricted hardware. However, transmitting huge volume of video could consume huge amounts of network bandwidth and lead to high delays in gaming responses. Though a great number of works such as [79] and [78] attempt to alleviate the latency issue by optimzing video encoding and data compression, the intrinsic delay constraints imposed by the long-range network connections are still non-negligible.

Component-based Architecture. Recently, a component-based cloud gaming solution have been proposed and implemented in [35]. This approach first divides a game into several sub-components and dispatches the selected components from cloud to players' devices as determined by devices' current conditions and their network connectivity. Though the Cognitive Resource Allocation architecture may alleviate the latency issue by avoiding video transmission, it causes unbearable burden to game developers who are required to modify source codes for existing games.

2.1.3 Energy Efficiency

One of the essential challenges of mobile cloud gaming is energy efficiency. In fact, limited battery life has been recognized as the greatest bottleneck for smart devices. Two main factors contribute to the energy issue. One is relatively limited capacity of batteries due to the minuscule size of mobile devices. The other is the increasing demand of resource-hungry gaming applications. Obviously, offloading provides a promising solution. However, as the network transmission incurred by the offloading process may consume extra energy, there are a number of existing works focusing on energy efficiency.

Offloading or Local Execution With the help of the offloading techniques, mobile gaming applications can be either performed in the mobile device or in the cloud. Executing computation-intensive applications consumes a great amount of power at the mobile device. It is a misconception that offloading and executing the applications in the cloud preserves the most energy. The fact is that bulky data transmissions, especially under unfavorable wireless channel conditions, could also result in waste of a large amount of battery power. The research work [80] describes the problem as follows. Given the size of data packet L, the wireless channel condition and the application completion deadline T, there is an optimal policy to determine where to execute the current application so that the energy consumption of the mobile device is minimized. The work proposes a solver which generates decisions in real-time.

MAUI [40] tackles this issues by requesting programmers to annotate the methods of an application that can be offloaded for remote execution. MAUI then constantly measures the network condition and estimates the bandwidth and latency. When a remote server is available, MAUI uses its solver to make a decision for offloading or local execution.

WiFi or 3G With the evolution of wireless communications and mobile devices, mobile devices nowadays are equipped with multiple wireless interfaces (e.g., Wi-Fi and 3G networks) simultaneously to transmit data. It is natural to raise a question: which network interface to use in order to minimize energy consumption?

Generally, the signal strength of a WiFi network is stronger than that of 3G networks. Hence, the data rate and energy saving performance of Wi-Fi usually outperform 3G networks. An experiment carried out in [40] showed that smartphones might consume three times more energy using 3G than using Wi-Fi with 50 ms RTT (Round Trip Time), or even five times more energy using 3G than using Wi-Fi with 25 ms RTT. However, 3G networks provide ubiquitous access while the coverage of Wi-Fi is much more limited. Therefore, the balance between Wi-Fi and 3G networks could shift from side to side depending on time and location. Research work [80] proposes a solver that adjusts packet transmission durations and control the transmit power on each interface according to the current channel conditions, so that the overall energy consumption of the mobile device is minimized.

2.2 Literature Review on Mobile Interaction

In this section, we review past work on exploring new interaction technologies and position our contribution of providing a virtual touchpad without any extra sensors.

2.2.1 Extending Smartphones' Input Space

One problem with touchscreens is that the users' fingers occupy valuable input space. Moreover, this problem aggravates when devices are shrinking in size. To tackle this issue, a great number of research work shift the interaction away from the touchscreen to nearby areas by augmenting smartphones with extra hardware such as keyboards [54], touchpads [32], cameras [52], and other specially-designed sensors [85, 67]. Despite the remarkable achievement, these system inevitably degrades devices' portability. Considering that smartphone platforms include increasingly sophisticated sensors, researchers start investigating the potential input approaches supported by the built-in smartphone sensing capabilities.

2.2.2 Built-in Sensor Based Input Approaches

A great number of research works utilize the built-in smartphone sensors such as inertial, compass, microphone, and camera sensors to provide alternative input approaches. Research work [53] leverages inertial sensors and touch input to provide motion-enhanced touch gestures. MagiTact [58] utilizes built-in compass sensor to extend interaction space of small mobile devices. Systems [69] detect users' tap events by analyzing the audio signals captured by microphones. LucidTouch [81] enables users to control the applications by touching the back of the device using computer vision techniques. More recently, wireless technologies have drawn a great amount attention of researchers. Some of them managed to extend smartphones input by utilizing wireless signals such as WiFi and RFID. For instance, Wigest [27] enables users' gesture input to smart devices by sensing the WiFi signal strength.

2.2.3 Virtual Touch Surface

Among various interaction technologies, touching is still most straightforward and natural approach for smart devices. A number of researchers managed to augment smart devices with virtual touch surfaces. For instance, RF-IDraw [75] relies on a RFID ring to infer the finger movement, allowing drawing in the air. TypingRing [66] leverages a ring equipped with motion sensors to trail a finger, enabling handwriting text-input experience. Okuli [86] augments smartphones with a LED light array to capture the finger movement, serving as a virtual trackpad for the user. Besides, Canesta [70] and OmniTouch [52] achieve the same goal by utilizing external cameras and image process techniques.

Despite of the promising results achieved, these systems inherit some limitations. Systems that rely on the extra devices may severely degrade the portability of the smart devices. More importantly, they also incur extra costs for the users. Recently, some researchers attempt to achieve finger tracking by purely leveraging the smartphone built-in sensors. For instance, Finger-in-Air [64] leverages smartphone cameras to detect the movement of a finger. UbiK [76] takes advantage of microphones on a smartphone to detect keystroke locations, serving as a virtual keyboard. However, the vision-based system [64] continuously turning on cameras may drain the battery swiftly and the performance may be negatively affected by low-light environment. As for the UbiK [76], it is solely built to distinguish among different keys a user presses, which may not be able to support applications such as handwriting text-input and drawing that require continuous finger movement tracking.

To address these issues, we propose a novel system UbiTouch which uncovers the potential of the built-in smartphone sensors to extend smartphones with virtual touchpads. Unlike existing works, it achieves continuous finger movement tracking while does not rely on any external devices or cameras.

Chapter 3

GBooster

In this chapter, we explore the possibility of utilizing neighboring multimedia devices to accelerate mobile multimedia applications. This is chapter is organized as follows: Section 3.1 demonstrates the motivation of this work. We then illustrate the challenges in the design and implementation, and present several techniques to cope with them in the consecutive sections. After that, we elaborate the experiments and results in Section 3.6. Finally, Section 3.7 concludes this chapter.

3.1 Motivations

Though a multimedia mobile application generally involves CPU and GPU computation, it is often the case that the GPU is the bottleneck of the application due to its limited processing power and high energy consumption.

Limited GPU Capacities: Table. 3.1 demonstrates the recommended CPU/GPU requirements of the most demanding games in recent years including *Modern Combat 5: Blackout* [12] in 2014, *GTA San Andreas* [8] in 2015, and *The Walking Dead: Michonne* [23] in 2016. The table also shows the CPU/GPU capabilities of the mainstream smartphones in those years including *Samsung Galaxy S5* (2014), *LG G4* (2015), and *LG G5* (2016) respectively. Note that the CPU and GPU capabilities are demonstrated in

	2014	2015	2016
CPU/GPU requirement	1.5 GHz 3.6 GP/s	1 GHz 4.8 GP/s	1.2 GHz 2-Core 6.7 GP/S
CPU/GPU capability	2.5 GHz 4-Core 3.6 GP/s	1.8 GHz 6-Core 4.8 GP/s	2.15 GHz 4-Core 6.7 GP/s

Table 3.1: Game Requirement versus Smartphone Capability. The recommended requirements represent the capabilities needed for running those games in the highest graphics settings and achieving a frame rate of at least 30 frames per second.

terms of CPU clock rate and GPU fillrate (GPixel/s) respectively. It can be seen that the devices' CPU capacities are commonly beyond the requirements of the most demanding games. On the other hand, the games are pushing the limit of the devices' GPUs, which become the performance bottleneck.

To make matters worse, the performance is usually downgraded due to the overheating issue. Nowadays, GPUs tend to create abundant heat energy when they are heavily utilized. To prevent overheating, mobile device systems have to reduce the GPU's operating frequency and suppress its performance when its temperature exceeds certain thresholds. Fig. 3.1 demonstrates how the GPU frequency and temperature changes with time when the device LG G4 is running the game *GTA San Andreas*. The GPU frequency initial reaches 600Mhz and remains steady for the first 10 minutes. After that, the GPU temperature meets the threshold and the operating frequency drops drastically to 100Mhz. As a consequence, the application's performance is significantly downgraded, resulting in an unacceptable user experience.

High Energy Consumption: Heavy GPU utilization also leads to swift battery drain. To demonstrate this, we run a test program [15] that renders a static triangle at a default frame rate of the Android system, which is 60 frames per second (FPS) on the three mobile devices mentioned above. We then measure the energy consumption incurred by CPU and GPU using approaches introduced in [71] and [33]. The results show that the power usage for each GPU is approximately 3 W, almost 5 times higher than the energy used by the CPU. As the program only performs elementary GPU computation, the power consumption of complex multimedia applications can be far more than this result, which can significantly shorten the battery lifetime.

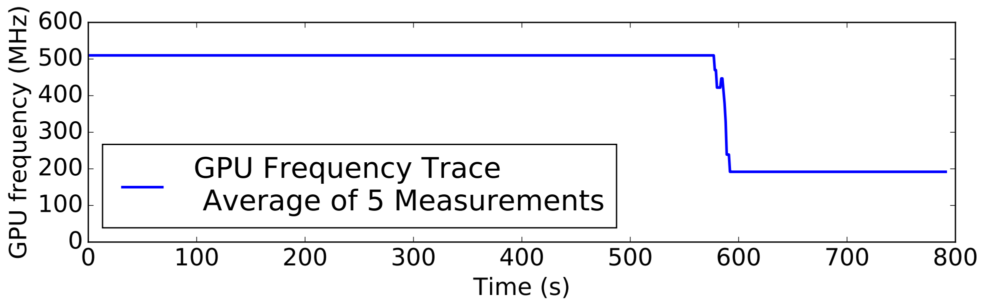

Figure 3.1: GPU frequency trace.

To alleviate these issues, conventional solutions typically offload GPU tasks to remote cloud servers. However, these cloud-based solutions usually require an Internet connection with huge bandwidth, which is not always available for users. Moreover, the long physical proximity of the cloud centers usually leads to high network latency, which is undesirable for highly-interactive multimedia applications.

In light of the above issues, we introduce GBooster, a novel GPU-task offloading system that aims to meet the following objectives.

High FPS, Low Latency, and No Requirement of the Internet. GBooster enables smart devices to run a GPU-intensive multimedia application at a high frame rate and low latency without an Internet connection.

Extend Battery Life. GBooster reduces energy consumption of multimedia applications and extends the battery lifetime.

GBooster achieves these goals by exploiting the processing power of neighboring multimedia devices including game consoles, smart TVs, and

PCs. These devices possess two essential advantages compared with cloud servers. First, they are prevalent and equipped with abundant processing power. According to [7] and [17], 80% of U.S households own a game console and half of them own a smart TV. All these devices are usually equipped with powerful GPU chipsets. For instance, the game console *Nvidia Shield* [18], is equipped with a GPU with a fillrate up to 16 GP/s, making it an ideal offloading destination. Besides, traditional PCs could be another sound option as modern computers generally possess GPUs that are 10 times more powerful than mobile devices' [10]. Second, these devices are typically connected with a local area network (e.g., in-home WiFi), which provides significantly larger bandwidth and smaller latency compared with an Internet connection.

GBooster works on every commercial Android device and supports all multimedia applications without changing or recompiling any source codes of the applications and the Android operating system. Note that GPU is not only heavily used by high-end 3D applications like games, but also widely used for rendering 2D user interfaces for various non-gaming applications. In this paper, we mainly focus on gaming applications, but we will show that non-gaming applications can also benefit from our system.

3.2 System Overview

Figure. 3.2 depicts the system architecture, which contains the essential procedures to migrate graphics computation from a user mobile device to an offloading destination. In the following sections, we refer to a user's mobile device as a **User Device**. Besides, we refer to an offloading destination as a **Service Device**.

As shown in Fig. 3.2, GBooster first dynamically inserts one wrapper layer to the user device while a multimedia application starts running. The wrapper enables the system to intercept all graphics commands from the

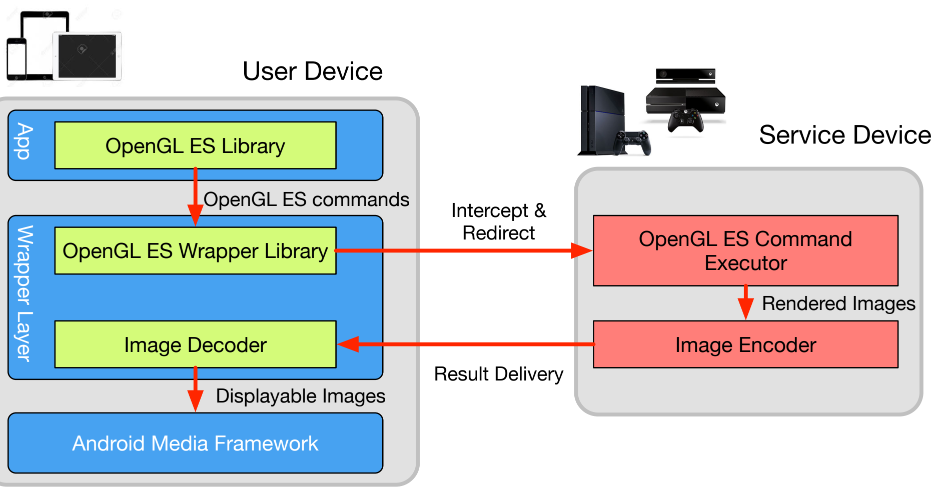

Figure 3.2: System Architecture

application and redirect them to a remote service device. Based on the received commands, the service device conducts the graphics computation using its own GPU. Once the computation is done, the rendered results will be encoded and delivered to the user device. Finally, the user device decodes and displays the images on the device's screen. We will describe the whole process in details in Section 3.3.

In this architecture, the network communication between the user device and service device plays an essential role. In order to improve network performance and reduce energy consumption, we propose an approach that eliminates redundant data and intelligently switches among multiple wireless interfaces based on the traffic volume. The details will be elaborated in Section 3.4.

Note that Fig 3.2 only depicts the scenario with one user device and one service device for better demonstration purposes. In Section 3.5, we extend the system such that it can aggregate distributed processing capabilities from multiple service devices to obtain further performance improvement.

3.3 Enable GPU Task Offloading

The Android system provides high performance graphics processing support for multimedia applications with the help of Open Graphics Library named OpenGL ES [65]. OpenGL ES is a cross-platform graphics API that specifies a standard interface for GPU and the Android applications could invoke the APIs to directly interact with the GPU. OpenGL ES features a client-server model as shown in Fig. 3.3. The application that invokes OpenGL ES APIs behaves as a client. It keeps generating a series of graphics commands to the server component. The server, which is typically executed in the GPU, interprets the commands and performs the actual graphic computation.

Based on this model, we propose an offloading approach that we intercept the command streams from a OpenGL client and redirect them to a OpenGL ES server residing in a remote machine. This approach contains two key advantages. First, it works universally for every multimedia application regardless of its implementation details (e.g., programming languages or graphics engines), since all of them internally invoke OpenGL ES calls. Besides, it requires no source code modification of the legacy applications.

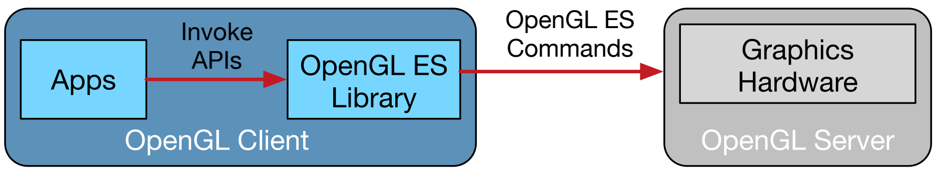

Figure 3.3: The Client/Server Model of OpenGL ES.

3.3.1 Intercepting and Rewriting OpenGL ES Functions

Although this approach seems straightforward, implementing it entails a challenging issue; the OpenGL implementation in Android OS is closedsource and thus we cannot revise it to add functionalities for interception and redirection. To address this issue, we adopt a technique named Dynamic Linker Hooking [3].

Specifically, hooking is the process of intercepting a program's execution at a specific point, typically entries of functions, in order to alter or augment the program's behavior. The dynamic linker hooking technique enables hooking in the runtime by forcing a program to load shared libraries specified by the user instead of the original ones provided by operating systems. In our case, rather than the genuine OpenGL ES library provided by the Android multimedia framework, we instruct the applications to load a wrapper library, which intercepts all the graphics command calls.

In detail, we notice that an application could invoke the OpenGL ES graphics APIs in three different ways:

1. An application may link to an OpenGL ES library so that it can directly call the OpenGL ES APIs.

2. An application may utilize the *eglGetProcAddress* function to get pointers to the OpenGL ES APIs.

3. Less likely, an application uses system calls *dlopen* and *dlsym* to dynamically load the OpenGL ES APIs.

Therefore, we have to intercept the OpenGL ES APIs in all these situations. For the first case, we simply implement wrapper functions for all the OpenGL ES APIs in our wrapper library. We then force the application to use the wrapper library by applying the Dynamic Linker Hooking technique. It is worth to note that the hooking can be easily done by setting the application's LD_PRELOAD environment variable in the Android system. Regarding the second case, we intercept and rewrite the *eglGetProcAddress* function such that it directly returns the pointers pointing to our wrapper functions. Similarly, we handle the third case by rewriting the *dlopen* and *dlsym* functions so that they load our wrapper library in preference of the original OpenGL ES library.

3.3.2 Forwarding Graphics Commands

We are now able to capture the OpenGL commands and ready to forward them to a remote service device. To facilitate network transmissions, we first need to serialize the commands' parameters.

OpenGL ES commands contain two types of parameters; one is the basic data types (e.g., integer and string) and the other one is the pointer type. It is straightforward to handle basic data types as we can easily calculate the length of the data. On the other hand, the situation becomes complicated when dealing with pointers. A pointer parameter typically refers to a sequence of data stored in RAM. Generally, the length of the sequence is either provided as a parameter or could be calculated with prior knowledge of its data structure layout. However, a heavily-invoked function *glVertexAttribPointer* contains a pointer parameter whose size could not be determined at the moment we intercept the function. Instead, the actual length is only revealed in consecutive drawing commands (e.g., *glDrawElements*) which render geometries using the pointer. To enable correct serialization, our system defers the transmission of the *glVertexAttribPointer* command until the pointer size is obtained in the later calls. We found that the reorder does not influence the final results so long as *glVertexAttribPointer* appears before the drawing calls.

Once the serialization is done, we could start transmitting the data over a network connection. Since the graphics commands must be delivered to a remote service device in a reliable and in-order manner, we may select TCP as the transmission protocol. However, due to its complex retransmission mechanism, TCP possesses an inherent delay, which is approximately 40 ms in general settings [20] and could be significantly higher under a poor network condition. To alleviate the delay, instead of TCP, we select the UDP transportation protocol to provide fast delivery of the graphics commands. To prevent packet loss and out-of-order delivery, we implement a light-weight and reliable transmission mechanism in the application layer [49].

3.3.3 Executing Commands and Retrieving Results

Upon receiving the graphics commands, the service device delivers them to its local GPU for execution. Since GPUs in the majority of multimedia devices provide native support for OpenGL ES, the service device simply acts as a relay and feeds the commands into the GPUs directly. Regarding a small number of devices such as Mac OS X that lack support for OpenGL ES, we could still bypass the restriction by utilizing OpenGL ES emulators [14] that translate OpenGL ES API calls to other natively graphics API calls.

When the computation is completed, the rendered images are transmitted back to the user device for display purposes. The display system of Android adopts a double-buffering mechanism to reduce image flicker and tearing [16]. As a sequence, when an application decides to redraw the screen, it has to invoke a graphic API named *SwapBuffer*. The API will notify the Android system to retrieve rendered images from the GPU and draw them on the screen. However, in our case, the rendered images are obtained from the network rather than the local GPU. To tackle this issue, our system intercepts and changes the behavior of the *SwapBuffer* command; upon intercepting the command, our system directly forwards an image received from network to the Android system for display.

3.4 Energy-saving Network Transmission

By offloading GPU tasks, GBooster reduces the power consumption of high-power GPUs. However, it comes at the energy expense of network transmissions, which may negatively impact the battery life. Considering that the energy cost of a WiFi interface is nearly proportional to the traffic load [50], we propose several approaches to reduce traffic volume. Besides, we reveal the potential of the low-power low-throughput BlueTooth interfaces to further suppress energy consumption.

3.4.1 Eliminating Redundancy of Network Traffic

GBooster transmitting unoptimized traffic data consumes enormous bandwidth (approximately 200 Mbps) even with a low-quality graphics setting (i.e., a resolution of 600$\times$480 with 25 FPS). We investigate this issue and notice that the traffic data including graphics commands and rendered images contains vast redundancy.

First, the sequences of graphics commands to generate consecutive frames tend to contain huge similarities. For example, an application might draw a same object with two different rotation angles, in which, the corresponding sequences may only differ slightly in the rotation command. We eliminate the redundancy by applying the LRU caching algorithm; the system caches the latest and frequent commands on the user device and the service device. Thereby, the user device can skip transmitting the commands which are cached. Besides, we further reduce the redundancy by using a light-weight general stream compression algorithm named *LZ4* [39], which achieves a compression ratio of 70% while barely incurs extra CPU workload.

Besides, the raw rendered images contain enormous redundancy, since the consecutive frames are typically similar to each other, especially when the images are static or barely vary. One straightforward solution is to encode the images into a video stream using the video encoder *x264* [28], which is considered the most efficient one. However, because the majority of multimedia devices other than PCs are equipped with ARM-based CPUs that the encoder is not optimized for, the encoding process is unacceptably slow. The normal speed is only around 1 MegaPixels/sec, far less than the speed of 7 MegaPixel/sec in which the application generates raw frames. Clearly, this approach fails to meet the requirement of real-time encoding. Rather than using a video encoder, we adopt a lightweight image encoding algorithm named Turbo [6]. The image encoder eliminates the redundant data by only transmitting incremental updates between consecutive frames and utilizing the JPEG image compression algorithm. It can provide a much more rapid encoding speed (up to 90MegaPixel/sec) and a high compression ratio (up to 25:1) without incurring heavy CPU load.

3.4.2 Enabling Transmission via Low-Power Interfaces

Nowadays, mobile devices are typically equipped with Bluetooth and WiFi. Wi-Fi interfaces offer a high-bandwidth data-link (up to 450 Mbps) while at the cost of high energy consumption (around 2 W when transmitting at the highest rate) [50]. On the other hand, Bluetooth is an order of magnitude more power efficient (less then 0.1 W) than WiFi, but with an order of magnitude lower bandwidth (approximately 21 Mbps) [55]. This presents us a chance to reduce power consumption by leveraging Bluetooth for network transmission on the premise of meeting demand of network traffic. In our system, we implement a mechanism that dynamically switches between the Bluetooth and the WiFi to meet the traffic demand while to preserve energy as much as possible.

However, implementing it entails a challenging issue resulted from the latency of switching the state of the WiFi interfaces [73]. Our preliminary experiments show that it takes at least 100 ms to wake up a disabled WiFi interface. More frequently, the interface has to re-associate with its access point after being in sleep mode awhile, making the wakeup time much longer (more than 500 ms). Consider a scenario that a system is transmitting data via its Bluetooth interface. If the increasing traffic load exceeds the throughput of the Bluetooth interface, the system has to enable the WiFi interface immediately in order to meet the demand. As the WiFi interface can not be fully functional instantly, the exceeding packets may be lost and retransmitting them will result in high network latency and frame jitter.

We address this issue by applying time-series analysis techniques, which enable us to foresee the escalating traffic trend and to turn on the WiFi interface beforehand. In other word, our objective is to predict traffic volume y_{T+h} given the information available at time T for $h > 0$. Mathematically speaking, we would like to obtain a forecast:

$$y_{T+h|T} = E(y_{T+h}|y_1, ..., y_T), \tag{3.1}$$

such that $y_{T+h|T}$ has minimum mean square forecast error (MSFE). To achieve this purpose, we first attempt to model the traffic volume with the widely-used Auto Regressive Moving Average (ARMA) model [51]. Specifically, $ARMA(p, q)$ with p autoregressive terms and q moving average terms can be described as follows:

$$y_t = \epsilon_t + \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \tag{3.2}$$

where ϵ_t are white noise terms and $\epsilon_t \overset{iid}{\sim} Normal(0, \sigma_w^2)$, φ_i and θ_i are parameters for this model.

We conduct preliminary experiments to measure the prediction performance including False Negative (FN) rate and False Positive (FP) rate. The FNs refer to the scenarios that the model fails to predict a soaring traffic demand that exceeds BlueTooth throughput. Conversely, FPs describe the cases that the model wrongly forecasts a traffic demand overpassing the Bluetooth throughput. Clearly, a small FN rate is more important to the system compared with a small FP rate, because a FN case results in elevated network latency while a FP scenario just causes slight increase in energy consumption. Our experiments show that the ARMA model provides a FP rate of 23.7% and a FN rate of 35.1%.

We notice that the FN rate is rather high and negatively impacts the system performance. We investigate the cause and realize that ARMA attempts to recognize and fit the time series pattern solely based on historic traffic data. However, the pattern beneath the traffic demand of our system is also affected by other exogenous factors. For instance, burst touching events from users may lead to drastic changes in game scenes and transmitting the varying scenes may escalate the network traffic. However, this abrupt change caused by external factors may not be modeled by the ARMA instantly, resulting in a FN scenario.

To tackle this issue, we adopt the Auto Regressive Moving Average with Exogenous Inputs model (ARMAX). Specifically, the $ARMAX(p, q, b)$ with extra b exogenous input terms can be formulated as:

$$X_t = \epsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \sum_{i=1}^{b} \eta_i d_{t-i}, \tag{3.3}$$

where $\eta_1, \ldots, \eta_b$ are the parameters of the exogenous input d_t. The model enables us to model deterministic and stochastic parts of the system independently. Thereby, we now can take some external inputs of the system into consideration and achieve better prediction performance.

To fit the traffic data in the ARMAX model, we first need to identify the effective exogenous input attributes for our system. We have examined the following potential attributes:

1. Touchstroke frequency: As we mention above, the touchstroke information may be informative. We could obtain the touchstroke information from a system file which is located in /proc/interrupts.

2. Length of graphics command sequences for each frame. A frame composed by a large number of commands likely has a complicated scene. It generates more traffic when delivering the frame.

3. Number of textures used in each frame. A frame filled out with a great number of textures tends to have a complicated scene. Transmitting it may consume more bandwidth.

4. Number of different graphics commands between two consecutive frames. If the command sequences composing two consecutive frames have immense difference, the scenes of the frames tend to vary significantly. Transmitting them may request more bandwidth.

We evaluate the qualities of the models consisted of different combinations of attributes by accessing the Raw Akaike Information Criteria (AIC) [29]. The results show that the best approximating model for the traffic is the one with the attribute 1 and 3.

In our implementation, we apply a recursive algorithm [63] for online estimating and updating the order (i.e., p, q, and b) and the corresponding parameters (i.e., φ_i, θ_i, and η_i) of the model. We forecast traffic demand for 500 ms and the experiment results show that the model could achieve a FP rate of 23% and a FN rate of 17%, outweighing the conventional ARMA model. When a soaring traffic trend that will exceed the Bluetooth throughput is predicted, our system turns on the WiFi interface and then configures the default route to direct the traffic through the interface. This process is performed smoothly and barely incurs packet loss.

3.5 Harnessing Capacities from Multiple Service Devices

As it is fairly common that there exist multiple service devices within a network, we may naturally raise a question: whether it is feasible to harness capacities from a cluster of service devices? Specifically, whether it is possible to parallelize and distribute GPU tasks to multiple service devices such that we can obtain a further speedup (i.e., a higher FPS)? The answer turns out to be positive.

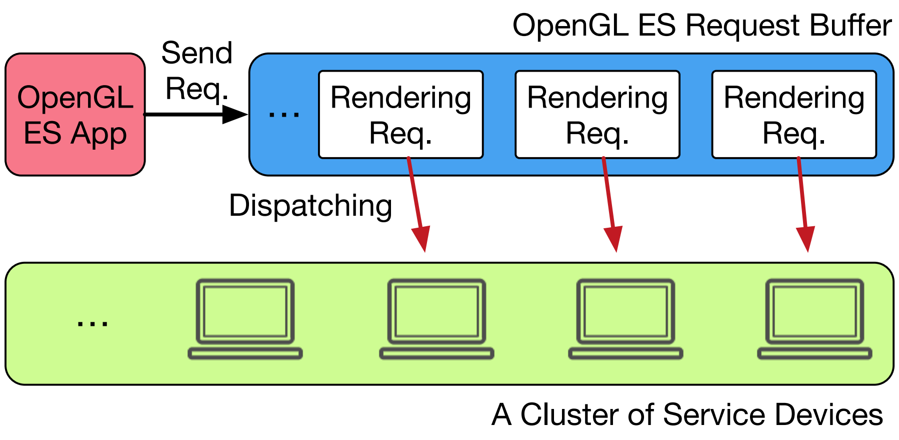

Figure 3.4: Distributed Computation of GPU Tasks.

3.5.1 Parallelizing GPU Computation

The key of parallelization lies in the OpenGL ES internal mechanism for handling rendering requests. A rendering request is defined as a sequence of graphics commands for rendering a frame and will be executed in a non-preemptive way according to the modern GPU architecture [56]. Generally, OpenGL ES handles rendering requests in an asynchronous manner. In other words, when an application issues a request to initiate rendering, it is not guaranteed that the rendering request is delivered to GPU and executed right away. Instead, the request may be buffered by the OpenGL ES client to optimize system performance, since it avoids frequent time-consuming input/output operations between CPU and GPU. We take advantage of this mechanism and achieve distributed computation as shown in Fig. 3.4; whenever there are multiple pending requests in the internal buffer, we distribute and simultaneously execute them in different machines so that we could obtain a higher FPS.

However, in reality an application, after submitting a rendering request, tends to issue a *SwapBuffer* command. The command halts the application and waits for the results from the GPU. In this way, the application forces the GPU to execute its requests immediately and a new rendering request will be issued only if the preceding one is finished. As a consequence, there is at most one request in the internal buffer, rendering the parallelization infeasible.

We overcome this problem by altering the behavior of the *SwapBuffer* command. Invoking the modified command returns immediately and does not halt the application. In this manner, the application will generate rendering requests at its quickest rate and multiple requests could be buffered.

3.5.2 Maintaining State Consistency among Devices

To enable distributed computation, we have to overcome another technique hitch due to the stateful nature of OpenGL ES APIs. All OpenGL ES calls are implicitly associated with an OpenGL context parameter, which is essentially a state machine that stores all data related to the rendering process such as the cached textures and vertex programs. Invoking an OpenGL API call on different contexts likely generates different results or even leads to unexpected errors. Since each service device possesses its own OpenGL context, simply distributing requests to them does not guarantee the correctness of the distributed computation.

To ensure the correctness, we have to maintain the consistency of the states among different service devices. We achieve this by first identifying the graphics commands which may alter the OpenGL states. Upon intercepting such commands, we replicate and deliver them to all service devices such that the states are consistent among all the devices. As we need to transmit duplicated data to multiple devices, a unicast connection is not an optimal option since it could result in waste of network bandwidth and limited system scalability. Instead, we take advantage of the multi-cast capability of UDP, which allows a stream of data to be sent to multiple destinations with a single transmission operation to reduce network traffic.

3.5.3 Assigning Requests to Devices

The last question left is which service device a request should be assigned to. Clearly, our objective is to assign each request to a service device that can deliver the result in the least time. Mathematically speaking, considering there exist N service devices, we dispatch each request to a node n that satisfies the following criterion:

$$n = \arg\min_{j} (w^j + r)/c^j + l^j, \text{ for } j \in [1, ..., N], \tag{3.4}$$

where the workload of the request is denoted as r and the computation capability of the service device n is denoted as c^n. Besides, we let l^n denote the round-trip delay time between the user device and the service device n and let w^n be the workload of preceding tasks in its service queue. Note that the workload of each graphics command is profiled using the approach in [56] beforehand.

As this mechanism does not guarantee that a preceding request is finished earlier than a subsequent request, our system keeps track of the sequence numbers of the requests, such that we can display their results in a proper order.

3.6 System Evaluation

In this section, we provide the detailed performance evaluation on GBooster.

3.6.1 Sample Games and Devices

Applications: We select six popular mobile games spanning three major game genres as shown in Table. 3.2. The majority of them have a large installation package size (above 500 MB) and a high requirement for graphics processing power.

User Devices: We run these applications on a set of smartphones including LG Nexus 5 (2013) and LG G5 (2016), which correspond to old-

	Genre	Package Size
G1: GTA San Andreas [8]	Action	2.41 GB
G2: Modern Combat [12]	Action	0.89 GB
G3: Star Wars [19]	Role playing	2.4 GB
G4: Final Fantasy [5]	Role playing	3.05 GB
G5: Candy Crush [1]	Puzzle	0.17 GB
G6: Cut the Rope [2]	Puzzle	0.12 GB

Table 3.2: Games for experiments and their package size.

generation and latest device models respectively. These smartphones are installed with different versions of Android, demonstrating that our system has good system compatibility.

Service Devices: We deploy and test our prototype on different types of multimedia appliances including a game console (Nvidia Shield), a smart TV box (Minix Neo Ui), a ladtop (Dell M4600), and desktop computers (Dell Optiplex 9010 with Nvidia GTX 750 Ti GPUs). All these service devices and smartphones are fully connected via a TP-Link WR802 router providing a 150 Mbps 802.11n WiFi network. To simplify the evaluation process, only one service device (the game console) is utilized in the following experiments for application acceleration and power saving. The PCs are only used in the evaluation for the scenarios with multiple service devices.

3.6.2 Application Acceleration

We first evaluate the effectiveness of application acceleration. In our experiment, we choose two FPS metrics that are widely used for measuring user experience of gameplay [77]. The first one is **median FPS** that represents the commonest frame rate experienced in the game and broadly correlates to what the player observes as graphical smoothness. A key advantage of using the median FPS that it naturally omits fringe results, for

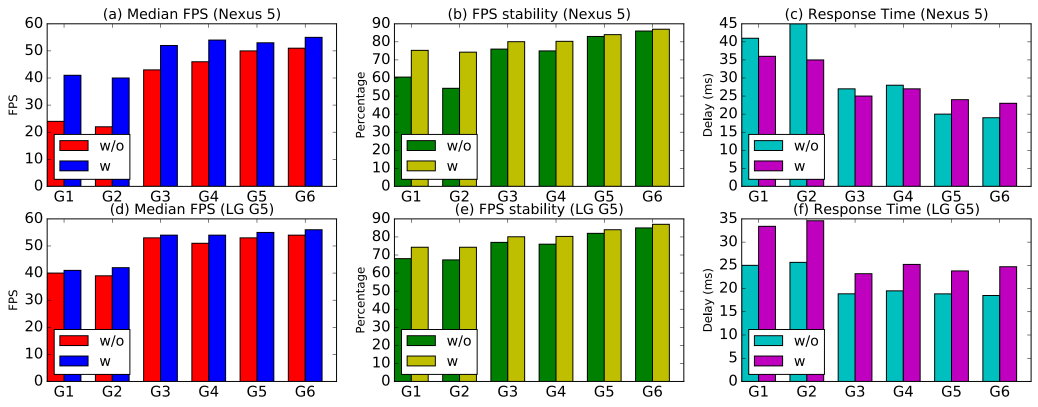

Figure 3.5: Median FPS and FPS stability.

instance, 0 FPS or 60 FPS which commonly occur during a game's loading screens and menus. Besides, we are also interested in the **FPS stability** which is defined as how much of a game session is played within a 20 percent range of median FPS. If the stability is low, it can serve as an indicator that gameplay is prone to frequent occurrence of FPS jitters, which typically lead to poor gaming experience. Apart from the FPS metrics, another essential factor that affects the gaming experience is the **average response time**. This metric t_r represents the average timespan between the moment a rendering request is issued and the time its result is displayed on its screen. Clearly, when the application is executed locally, this metric is equal to the reciprocal of the FPS (i.e., $t_r = 1000/FPS$). If the computation is offloaded, the metric also includes the time t_p spent on the offloading intermediate steps such as network transmissions and image encoding. In other words, this metric can be represented as:

$$t_r = 1000/FPS + t_p. \tag{3.5}$$

We conduct the experiments in controlled conditions that we play a game for 15 minutes with the same graphical settings (where configurable) and on the same levels (where there is a choice), meanwhile shutting down other applications on the phone. For comparison purposes, the experiments are conducted twice; one is with our system enabled while the other is not. The results are demonstrated in Fig. 3.5.

Effectiveness on Old-Generation Devices: One first observation from Fig. 3.5 (a) and Fig. 3.5 (b) is that the system boosts the median FPS and increases the FPS stability for each game on the old-generation device Nexus 5. In particular, the performance improvement for the action game G1 and G2 is rather significant. The median FPS drastically rises from 23 and 22 to 37 and 40 respectively. Generally speaking, a minimum standard for good playability is a median frame rate of 24 FPS, as this means that most of the game is played at a frame rate similar to a standard animation or film. However, action games such as shooting games tend to have a slightly higher FPS requirement (usually above 30 FPS) in order to display smooth motion of onscreen objects and maintain the illusion of being real for players [22]. Our results indicate that with the help of our system, the players now can enjoy decent playability of the two action games. We also notice that although the remaining games receive performance improvement as well, it is somewhat less significant than the action games. Specifically, the median FPS of the puzzle game G5 merely improves from 50 FPS to 52 FPS. It may be due to that the puzzle games, which contain only a small amount of animation, are less GPU-intensive than the action games. Thus, the local GPU can handle the computation efficiently and the benefits of remote execution are less obvious.

Regarding the FPS stability, we can spot similar patterns as the median FPS. The system improves the FPS stability for all the games. In particular, the FPS stability for the two action games soars form 60% and 55% to 75% and 74% respectively. This phenomenon can be explained by the better overheating-proof design of the service device. As described in the Section. 3.1, one major reason that results in unstable FPS is the GPU overheating. Since the GPU in the service device is usually equipped with cooling fans, it is less prone to overheating issue and can deliver stable processing power for the applications.

We now turn our attention to the average response time. It can be seen in Fig. 3.5 (c) that the response time for all the games is below 36 ms. As the average response time for human being is generally above 100 ms [9], the result indicates that the players can barely perceive any response lag when running an application with our system enabled. Another interesting observation is that the impact on response time varies according to the genres of the games. The response time for the action games drops approximately 10 ms, while the time only decreases around 2 ms for the role-playing games. Conversely, the response time for the puzzle games increases 4 ms. The phenomenon can be explained by the trade-off between the gain of FPS and the extra time t_p for the offloading intermediate steps as described in Equation. 3.5. For the action games, the gain of FPS is significant, thus the drop of response time largely outweighs the t_p. Regarding the role-playing games, the FPS gain seems to neutralize the time t_p, barely causing any change in the response time. In contrast, there is bare FPS gain for the puzzle games, in which case, the t_p is largely attributed to the increase of the response time.

Effectiveness on New-Generation Devices: It can be seen in Fig. 3.5 (d) and Fig. 3.5 (e) that our prototype barely benefits the two metrics when running in the new-generation device LG G5. It is mainly due to that the device now possesses an powerful GPU and can efficiently handle all the computation tasks locally. Even for the GPU-intensive action games, the device can achieve a considerable frame rate of 40 FPS, which is approximately 2 times higher than that on the Nexus 5. Therefore, there is little room for performance improvement via remote execution. The tiny gain of FPS then results in the increase of the response time for all the games.

3.6.3 Power Saving

We evaluate how much power can be saved with the help of our prototype. Specifically, we run the sample games on the two smartphones mentioned above and measure the system power using a tool introduced in [71]. To obtain accurate results, we first turn the phone into airplane mode, reduce the backlight brightness to 50%, and shutdown other background activities. We also cool down the phones before each test to make sure that the GPU can keep working at a stable frequency during the experiment. We select a specific repeatable scene as the test case and each is repeated for five times. In order to conveniently demonstrate the effectiveness of energy saving for different games, we normalize the results to the case of local execution.

Figure. 3.6 (a) shows the experiment results. One major observation is that the prototype reduces the power consumption for all the games and smartphones. It is expected since offloading avoids the heavy utilization of high-power GPUs. Clearly, the more intensive the GPU tasks are, the more benefits we can obtain from the offloading. It explains the phenomenon that the GPU-intensive action game G2 could achieve a normalized energy saving of around 70%, while the energy-saving for the puzzle game G6 is less effective, which is approximately 30%.

In addition, to demonstrate the effectiveness of the energy-saving interface switching mechanism, we also measure the power consumption with the optimization disabled. As shown in Fig. 3.6 (b), the overall system power significantly increases. In particular, the power consumption of the G1 soars from around 40% to 65%. It indicates that the system could preserve considerable amounts of energy by leveraging the low-power Bluetooth interface for network transmissions.

3.6.4 Multiple Devices

We now evaluate the system performance when multiple service devices are available. Specifically, we measure the FPS performance metrics of the action game G1 on the Nexus 5 Meanwhile, we gradually increase the number of service devices.

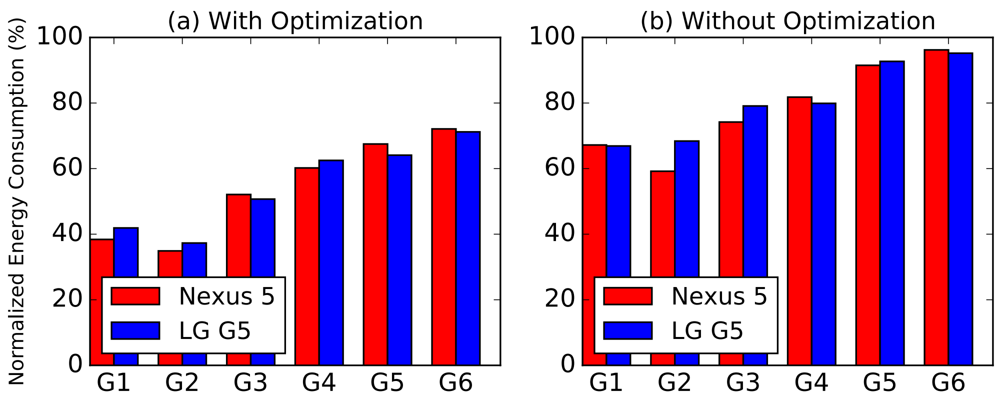

Figure 3.6: Normalized Energy Consumption for Different Games.

Figure. 3.7 demonstrates the experiment results. When the device number is zero, the game is executed locally. As the device number changes to one, the game obtains the most FPS improvement owning to the offloading. When two more devices are available, the FPS gains a significant increment from 40 to 51 by taking advantage of the distributed computation. However, the FPS barely increases and remains stable when more than 3 devices are available.

We examine the cause and notice that the internal buffer possesses at most 3 requests most of the time. Therefore, having more than 3 devices barely benefits the performance. The limited number of requests is possibly due to two reasons. First of all, most of graphics engines have a mechanism to ensure that the FPS does not exceed the device's maximum frame rate (60FPS). Thus, the speed of generating rendering requests may be limited. Besides, generating the requests consumes CPU resources and the number may also be constrained by the CPU.

In terms of FPS stability, it shows a similar pattern as the FPS metrics; the stability increases steadily as the device number is less than 3, and remains stable after that.

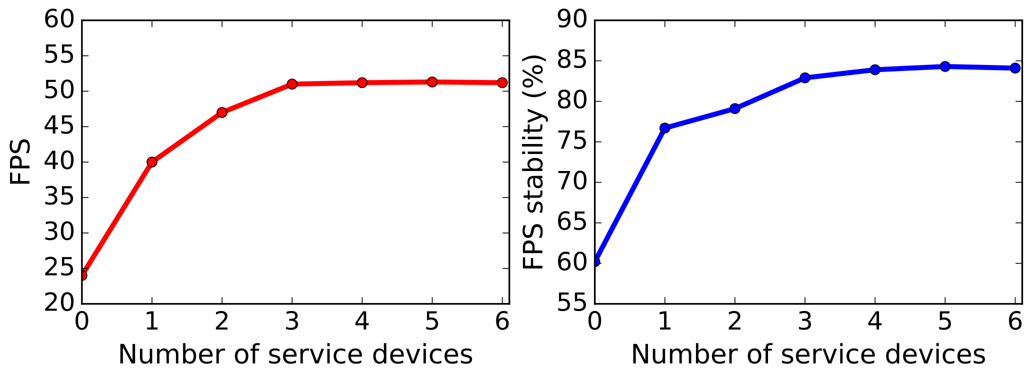

Figure 3.7: FPS Metrics with Multiple Service Devices.

Application Name	FPS Boost	Energy Consumption
Ebook Reader	0	92.1%
Yahoo Weather	0	93.6%
Tumblr	0	93.3%

Table 3.3: FPS Boost and Normalized Energy Consumption for Non-gaming Applications.

3.6.5 Performance on non-gaming apps

Although we mainly focus on GPU-intensive mobile games in this work, we also evaluate what non-gaming applications can benefit from our prototype. We measure the effectiveness of application acceleration and power saving of three popular non-gaming applications including *Ebook Reader* [4], *Yahoo Weather* [24], and *Tumblr* [21]. We utilize *MonkeyRunner* [42] to generate same sets of touch events for repeatable tests including reading an article, viewing weather information, and browsing a post for ten times.

Table. 3.3 demonstrates the experiment results. It can be seen that our prototype provides tiny energy saving (7% on average) and no FPS boost for the applications. It is expected since these applications generate much less GPU workload, compared to the games. However, The power saving is still valuable, considering that battery resource is rather scarce on smartphones.

3.6.6 Comparison with Cloud-based Solutions

For comparison purposes, we also evaluate the performance of the most popular cloud-based solution *OnLive* [13]. Specifically, we measure the median FPS and response time by adopting the measurement method in [36]. It is worth to note that unlike our system that universally supports all mobile multimedia applications, the platform offers a limited number of application choices. We conduct our tests on ten games and report the average results: with an Internet connection of 10 Mbps bandwidth, the platform can stream games at a resolution of 1280 $\times$ 720 with a frame rate of 30 FPS and average response time of approximately 150 ms. We notice that the FPS is capped at 30 FPS because of the setting of the video encoder used by the platform. Moreover, the average response time is almost 5 times longer than our prototype's due to the long proximity to the cloud server.

3.6.7 System Overhead

Memory Overhead. Our system allocates extra memory in user devices. To quantify the memory overhead, we measure the extra memory consumption in the games shown in Table. 3.2. The experiment results show that the average memory footprint is fairly small, which is 47.8 MB. Considering typical smart devices are equipped with gigabytes of memory space, the memory overhead is almost negligible.

CPU Overhead. Our system consumes extra CPU resources for intermediate procedures of offloading such as data compression and image decoding. We measure the extra CPU usages on the Nexus 5 phone. The results show that when running locally, the most compute-intensive application G1 accounts for an average CPU usage of 68%. When the offloading is enabled, the CPU load increases to 79%. Clearly, the device's CPU is still underutilized and the tiny increment of CPU usage barely impacts the system performance.

3.7 Conclusion

In this paper, we propose GBooster, a system that accelerates GPU-intensive mobile applications by transparently offloading GPU computation onto ambient multimedia devices such as SmartTV and Gaming Consoles. We implement GBooster on the Android platform and demonstrate that the prototype can significantly increase applications' frame rates and reduce their energy consumption.

Chapter 4

UbiTouch

In this chapter, we explore a novel interaction technology that extends smartphones with virtual touchpads on ordinary desktops by purely utilizing smartphone built-in sensors. This chapter is organized as follows: Section 4.1 is the overview of this work. We then elaborate the implementation of the system and present our preliminary findings in the subsequent sections. We demonstrate the experiment results in Section 4.7. Finally, we conclude this chapter in Section 4.8.

4.1 System Overview

UbiTouch extends touch-screen based devices with an external virtual touchpad on a common surface (e.g., wooden desktop). Similar to a conventional touchpad, it accepts general touch actions including tapping and dragging.

Fig. 4.1 demonstrates the typical use case of UbiTouch. For the purpose of illustration, a 7 cm $\times$ 13 cm rectangle zone is marked on a desktop, serving as a virtual touch area. An off-the-shelf Android smartphone (LG Nexus 5) is then placed vertically on the top border of the zone, enabling the ALS and PS to sense the movement inside the virtual touch area. Before running UbiTouch, an initial training is needed, whereby the

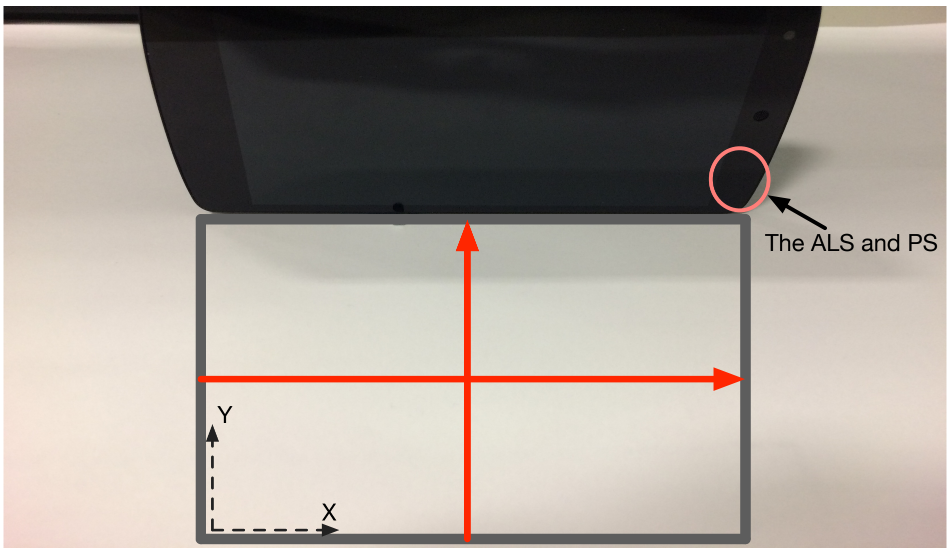

Figure 4.1: Typical system deployment.

user moves his finger along some trajectories instructed by the system to generate some training data. UbiTouch then learns a mapping function from the training data through a neural network. It then uses the function to convert the subsequent sensor readings into two-dimensional coordinates, enabling continuous tracking.

In this research, our design goal is for UbiTouch to work in a portable, accurate, and robust manner. UbiTouch should only rely on smartphone built-in sensors to achieve centimetre-scale finger tracking and touch action detection. Meanwhile, UbiTouch should be resilient against minor changes of the ambient light. To achieve these goals, UbiTouch adopts a work-flow shown in Fig. 4.2. It can be seen that the architecture of UbiTouch comprises the following three major components: (i) finger tracking, (ii) touch action detection, and (iii) run-time calibration and adaptation.

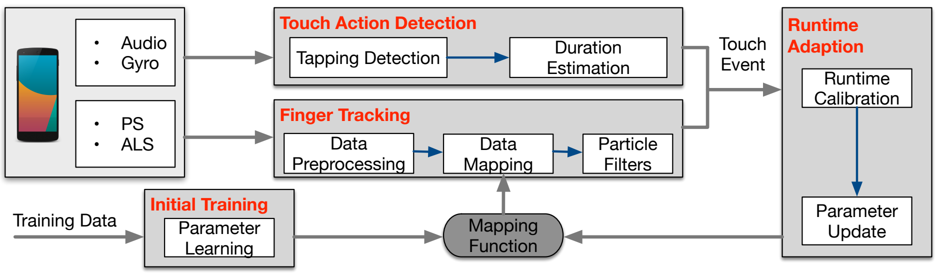

Figure 4.2: System Logic Flow for UbiTouch

4.2 A Peek at Trailing a Finger Using ALS and PS

In this section, we first provide some rationales about how raw readings from ALS and PS sensors could be utilized to track a user's finger. Then we conduct some preliminary experiments to determine the utility of the ALS and PS information.

Figure. 4.3(b) demonstrates the underlying mechanisms of the ALS. It can be seen that the ambient light sensor consists of two photodiodes (CH0 and CH1) in two separate locations. The first photodiode is sensitive to both visible and infrared light while the second photodiode is primarily sensitive to infrared light. The fusion of the two readings is used to estimated luminance of the visible light. Assuming the settings of background light sources (i.e., positions and brightness) remain unchanged, movement of an object or finger may alter the path in which the light propagates. It then affects the intensity of the light received by the two photodiodes. Therefore, abundant information can be derived on how an object moves from the light intensity readings.

Meanwhile, the proximity sensor depicted in Fig. 4.3(c) also provides extra movement information. A LED emitter broadcasts a certain number of infrared pulses. The pulses strike a nearby object (which is a user's fin-

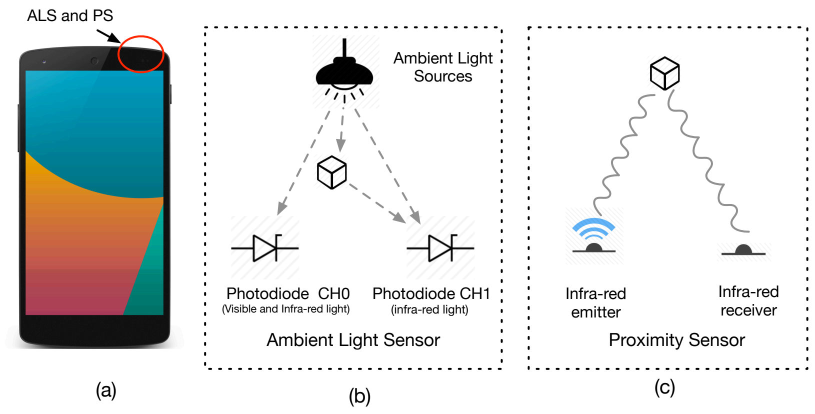

Figure 4.3: (a) shows the positions of the PS and ALS on a smart phone. (b) and (c) show the underlying mechanisms of the ALS and PS respectively.

ger in our scenario) and get reflected to a receiver. The receiver counts the number of pulses and estimates the distance based on a simple principle: the farther the object is, the fewer the number pulses received, since the infrared light has to travel in a longer path and loses more energy. It is easy to see that variation of the pulse number has a strong link with the object movement, which may assist in our goal.

We start with some simple tests to determine the utility of the ALS and PS information. We first setup the experiment environment which complies with the UbiTouch's typical use case shown in Fig 4.1. All experiments are conducted in an ordinary office environment where the settings of background light sources (i.e., positions and luminance) do not vary.

For the purpose of illustration, we specify a coordinate system for the area where the origin is located at the lower-left corner and two axes are marked by the dash lines in the figure. To understand how the movement of a finger affects the readings of the ALS and PS, we instruct a user to move his finger along the trajectories in red. The trajectories are designed to be either parallel to the axis 'X' or axis 'Y' such that we can better observe the variation trend by altering one variable each time.

During the movement, we record five types of readings: (1) luminance (LUX); (2) distance measurement (a binary value representing 'far' or 'near'); (3) raw sensor reading from the photodiode CH0; (4) raw sensor reading from the photodiode CH1; (5) count of pulses received by the PS. The first two kinds of data can be easily retrieved via the sensor API provided by the Android platform. However, the system blocks the access to the remaining three types of data. To overcome this constraint, a patch is applied on the OS kernel, which enables us to bypass the Android framework and access the raw readings from the hardware directly. Specifically, the patch first replaces the original proprietary PS and ALS driver with an open-source one [31]. The new driver is designed to stream the data of registers in the hardware to user-space applications through SYSFS interfaces [61]. The UbiTouch application continuously polls the samples from the driver and process them. In our implementation, we set the polling frequency for the PS and ALS to 100Hz, which is the maximum frequency the hardware can support [30].

We display these readings for the two trajectories in Fig. 4.4(a) and Fig. 4.4(b) respectively. From top to bottom, the sub-figures demonstrate how the luminance, binary distance measurements, CH0 readings, CH1 readings, and pulse counts change over the finger's positions. It can be seen that, regardless of the moving directions, there is no obvious link between the movement and the luminance. We analyze the possible causes and find that the resolution for the noisy luminance readings is somewhat low because the values are integers and only vary between 20 to 26 in our experiments. It renders the luminance an unsuitable feature for our goal.

As for the distance measurements, the binary values will turn to one when the finger moves within a certain radius (around 5cm) of the PS. Nevertheless the binary information is still coarse-grained to realize accurate localization and tracking.

We now turn our attention to the three remaining types of data. The first observation we can easily identify is that the readings for CH1 and

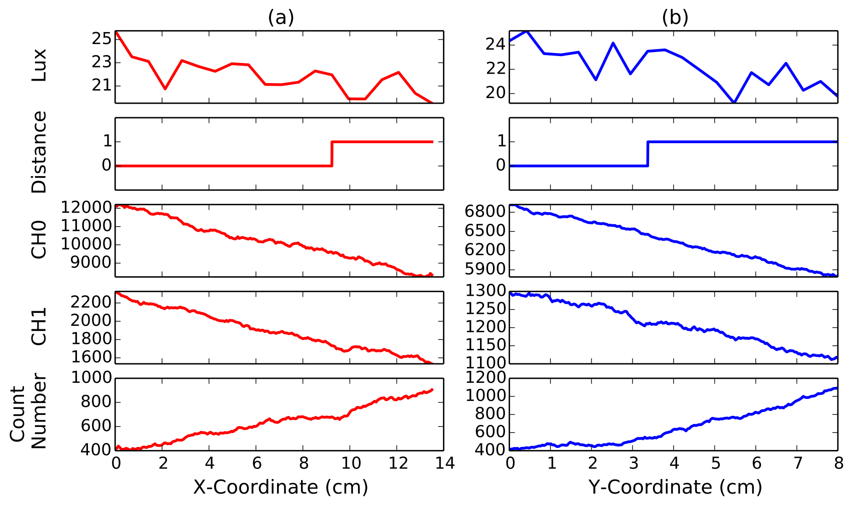

Figure 4.4: Recorded sensor readings during the movement along the aforementioned trajectories. The readings have been interpolated and outliers have been filtered out.

CH0 have a strong correlation with the finger's movement. Specifically, as the finger moves closer to the ALS, the readings decrease. Another important observation is that the pulse counts also have a strong link with the movement. When a finger is getting further away from the PS, the pulse number received significantly decreases.

It can be seen that these data are location-dependent and have the potential to enable localization and tracking. We explore the practical solutions to obtain finger locations in the next section.

4.3 Obtaining Finger Locations from Raw Sensor Data

Let tuple $s_t : (r_t^0, r_t^1, c_t)$ denote the readings for the photodiode CH0, photodiode CH1, and pulse count at time t. Let $l_t : (x_t, y_t)$ denote the finger position, which is the coordinate on the aforementioned virtual touch area. To locate the finger from the raw sensor data, we use a mapping function f that maps the s_t into l_t.

Ideally, a well-chosen deterministic mapping model that takes the settings of the background light sources and placement of the smartphone into consideration will achieve best performance. However, as the background environment and smartphone deployment vary from time to time, it would be fairly difficult and laborious for users to decide the numerous parameters of the model when they are using the system. Thus, we seek a model that automatically tunes its parameters and adapts to different environments.

In this paper, we adopt a nonlinear mapping model called Curvilinear Component Analysis (CCA) [41]. The CCA is a self-organizing neural network which has the ability to learn the mapping between a high-dimensional space to a low-dimensional one. Mathematically speaking, CCA attempts to train a network (i.e. a mapping function) to optimize the following criterion that explicitly measures the preservation of the pairwise distances (denoted by E):

$$E = \sum_{j=1}^{N} \sum_{i=1}^{N} (\delta(s_i, s_j) - d(l_i, l_j))^2 F(d_{l_i l_j}, \lambda) \tag{4.1}$$

where N denotes the total number of the training data. The $\delta(s_i, s_j)$ is the curvilinear distance in the input space and $d(l_i, l_j)$ is the Euclidean distance in the output space between the i-th and j-th data points. The factor F weighs the contribution of each pair in the criterion, which is usually implemented as the Heaviside step function:

$$F(d(l_i, l_j), \lambda) = \begin{cases} 0, & \text{if} \quad d(l_i, l_j) - \lambda < 0 \\ 1, & \text{if} \quad d(l_i, l_j) - \lambda \geq 0 \end{cases} \tag{4.2}$$

where λ is a neighboring radius and set slightly larger than the maximum curvilinear distance measured in the input data set (i.e., $\max(\delta(s_i, s_j)|i < j \leq n)$).

The rationale behind this method is that if two readings are quite similar to each other, the points they are mapped into should also have a small distance. Conversely, a huge difference between sensor readings indicates a large distance between the points. Thus, this method attempts to obtain a mapping function f by minimizing the difference between the pairwise distances in the input space and output space.

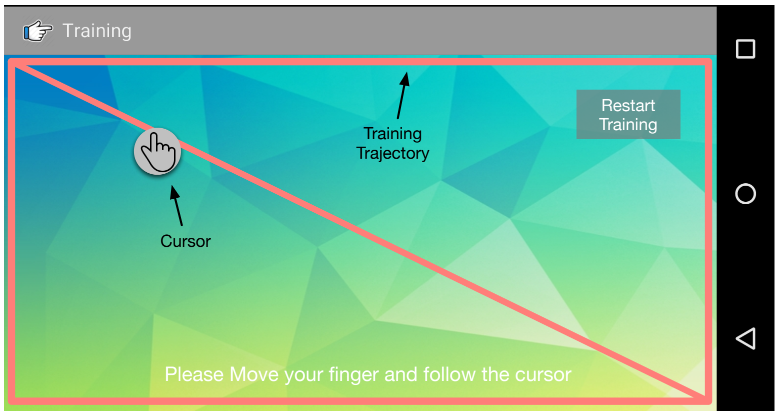

Figure 4.5: The application for gathering training data.

To apply this approach, we collect some training data as a series of s_t and the corresponding l_t. Thus, a data gathering application shown in Fig. 4.5 is built to facilitate this process. The application first requests the user to input the size of the touch area and displays a trajectory along which a cursor gradually moves in a certain speed. The user is then requested to move his finger inside the touching area on the desktop, imitating the movement indicated by the cursor.

The user may repeat the process for several times in order to achieve a good synchronization between the finger movement and the cursor motion. In this manner, the coordinates of the cursor on the smartphone screen could be transformed to the coordinates of the finger on the touch area by a simple scaling transformation. Compared with single point calibration, our calibration process can obtain far more training data. The training process typically lasts one minute.

Before feeding the training data to the neural network, we notice that the training data contain many outliers and heavy noises. Three preprocessing tasks are used to clean the training data.

Outlier Removal. We find that there are some abrupt changes of the data that are obviously not caused by the finger movement. Fig. 4.6(a) shows an example of the training data and it can be seen that there are some significant abrupt change in the l_0 and l_1 around the positions 2, 3 and 6.5. There are likely to be outliers and should be eliminated.

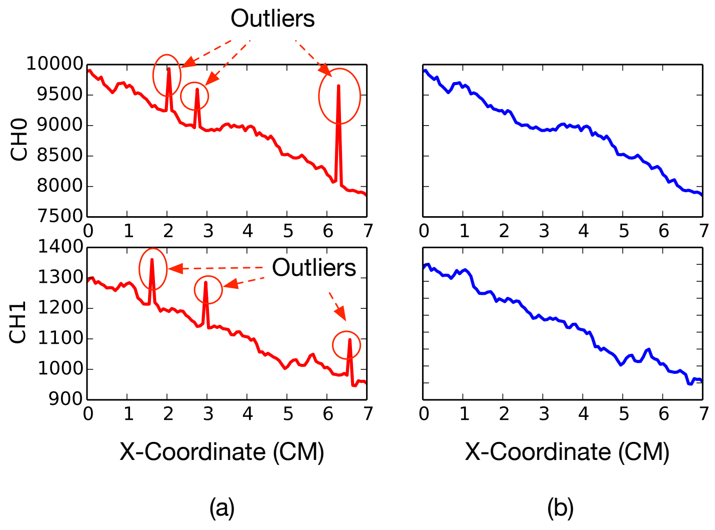

Figure 4.6: (a) The original CH0/CH1 data. (b) The CH0/CH1 data after the outliers are removed using the Hampel filter.

To achieve that purpose, we first attempt the widely-used Standard Deviation (SD) method. However, this approach highly relies on the mean and standard deviation which are extremely sensitive to the presence of outliers, hence does not perform well. Therefore, we utilize a more sophisticated outlier-removal algorithm called the Hampel Identifier [60]. The algorithm declares any data points not within the range $[\mu - \gamma * \sigma, \mu + \gamma * \sigma]$ as outliers, where u and σ denote the mean and the median absolute deviation of the sequence respectively, while γ is a constant and the typical value is 3 for general cases. Fig. 4.6(b) demonstrates the results for outlier removal. It can be seen that the Hampel Identifier eliminates the outliers and preserves the desire CH0 and CH1 responses.

Interpolation. We program the application to poll the sensor data from the hardware every 10ms (i.e., 100Hz). However, we cannot guarantee that a reading is retrieved at the desired frequency, because the Linux kernel used by typical smartphones does not provide real-time task scheduling. A detailed look at the data shows that sampling jitters appear frequently and the sampling latency can be up to 150 ms when the smartphone is having a high computation load. Therefore, the data must be interpolated before further processing. Here, we apply linear interpolation at the data to retrieve a re-sampled series with the desired sampling frequency of 100Hz.

Noise Filtering. The final step of pre-processing is to suppress the heavy noises in the data series. In our implementation, we utilize the wavelet filter proposed in [74] to smooth away the high-frequency noises. Specifically, we apply 4-level 'db4' wavelet transform on sensor data and only use the approximation coefficients to 're-construct' the filtered signal. Fig 4.7 demonstrates the data sequences before and after filtering. It can be seen that the noisy signal is getting much cleaner.

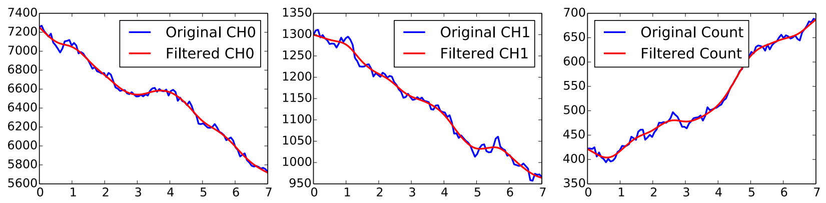

Figure 4.7: The effect of noise filtering.

The filtered data is fed to the CCA neural network. In the training phase, a tunable parameter called learning rate (denoted by $\alpha \in [0, 1]$) decides the convergence speed of the learning process. Here, we adopt a linear learning rate function which linearly decreases the rate with time.

4.4 Improving Tracking Accuracy via a Hidden Markov Model

In the previous section, we attempt to obtain the finger locations from the sensor readings by utilizing a CCA network. It simply maps the sensor reading s_t into a location l_t, without taking previous readings into consideration.

This method does not take advantage of an important observation: a user typically moves his finger smoothly, thus the finger position at a consecutive moment is likely not far away from the current location. The observation indicates that the finger movement can be modeled by a Hidden Markov Model (HMM) shown in Fig. 4.8. The model contains a set of hidden state variables (i.e., finger positions) and observable variables (i.e., sensor measurements). The current position l_t depends on the previous location l_{t-1} according to the probabilistic transition model $p(l_t|l_{t-1})$. The sensor measurement s_t can be regarded as a stochastic projection of the hidden state l_t generated via the probabilistic observation model $p(s_t|l_t)$. We now want to sequentially estimate the values of the hidden location l_k, given the values of the observation process $s_0, \cdots, s_k$, at any time step k, which follows the posterior density:

$$p(l_k|s_0, s_k, ..., s_k). \tag{4.3}$$

It can be seen that this model enables us to further improve the tracking accuracy by incorporating the historic sensor measurements.

To estimate the hidden states, we adopt an approach called Particle Filter [68] which can efficiently deal with a large number of possible states in our scenarios. The algorithm is described in three steps:

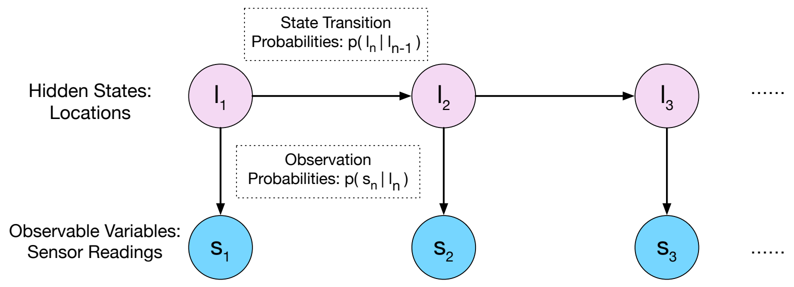
State Transition Probabilities: p(l_n | l_{n-1})
Hidden States: Locations
l_1
l_2
l_3
......
Observation Probabilities: p(s_n | l_n)
Observable Variables: Sensor Readings
s_1
s_2
s_3
......

Figure 4.8: HMM-based Movement Modeling.

1. Initialization:
   - For $i = 1, ..., M$, randomly initialize l_0^i, and set $t = 1$.
2. Importance sampling step:
   - For $i = 1, ..., M$, sample $\hat{l_t^i} \sim p(l_t|l_{t-1}^i)$.
   - For $i = 1, ..., M$, evaluate the importance weights by $\hat{w_t^i} = p(s_t|\hat{l_t^i})$.
   - Normalize the importance weights.
3. Selection step:
   - Re-sample with replacement M particles $(l_t^i; i = 1, ..., M)$ from the $(\hat{l_t^i}; i = 1, ..., M)$ according to the importance weights.
   - Set $t \leftarrow t + 1$ and return to the step 2.

This algorithm requires two probabilistic models: the transition model $p(l_t|l_{t-1})$ and the observation model $= p(s_t|l_t)$.

We first try to compute the transition model $p(l_t|l_{t-1})$. Based on the aforementioned observation, we can safely assume the user moves his finger in a reasonable speed not exceeding a constant value V_{max}. We now express the transition model as follow:

$$p(l_t|l_{t-1}) = \begin{cases} 0, & \text{if} \quad d(l_t, l_{t-1}) - V_{max} * T < 0 \\ \frac{1}{2\pi V_{max}^2 T^2}, & \text{if} \quad d(l_t, l_{t-1}) - V_{max} * T \geq 0, \end{cases} \tag{4.4}$$

where T is the sampling period (10ms). V_{max} here is set to $0.05m/s$ based on the previous finger touch research [34]. The intuition behind the transition model is that a finger located at l_{t-1} can uniformly move to any points within the circle whose center and radius are l_{t-1} and $V_{max} * T$ respectively.

Next we have to figure out the observation model $p(s_t|l_t)$. Noting that we have obtained a mapping function f that maps a sensor measurement s_t to a position l_t in the previous section, we could obtain the following equation:

$$s_t = f^{-1}(l_t) + W, \tag{4.5}$$

where f^{-1} is the reverse mapping function that projects l_t to s_t and can be obtained from the CCA network. W denotes the measurement noise and has a Gaussian distribution, that is $W \sim \mathcal{N}(0, \Sigma)$ and $\Sigma = \text{diag}(\sigma^2)$. The σ is a standard deviation and set to 10 which is measured from the real sensory data.

Therefore, the measurement model can be expressed as:

$$p(s_t|l_t) = C * \exp\big((s_t - f^{-1}(l_t))^T \Sigma^{-1} (s_t - f^{-1}(l_t))\big), \tag{4.6}$$

where C is a constant and does not affect the final results due to normalization in the importance sampling step.

It should be noted that, the algorithm's time complexity is linear with respect to the number of samples M. Obviously, the more samples, the better the accuracy, so there is a trade-off between speed and accuracy. To find an optimal value of M for the system, we adopt a strategy described in [72] which dynamically adjusts the number of samples and is adaptive to the available computational resources.

4.5 Touch Action Detection

The system is now able to accurately track the finger movement. However, to mimic a virtual touchpad, the system requires to detect two typical touch actions including tapping and dragging.

Our system realizes the detection by leveraging a smartphone built-in microphone. Tapping on a desktop typically generates audible stroke sounds. Similarly, dragging gestures request the user to rub on a desktop, producing constant audio noises as well. These signals could be received by the sensor and facilitate the touch action detection.

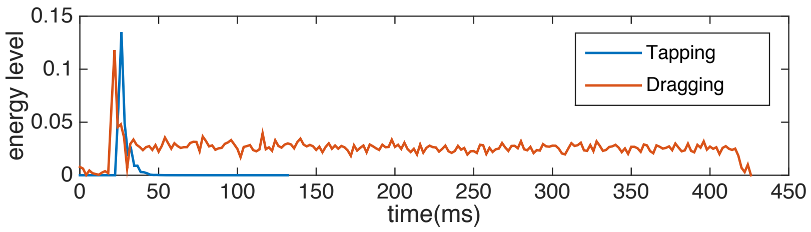

Figure 4.9: Energy level of the audio signals for tapping and dragging. It is obtained via short-team average on the audio signal with a moving window.

Fig 4.9 shows the energy level of the audio signals received when a user is tapping and dragging on a desktop. The audio energy generated by a tapping has a pattern similar to an impulse response. It starts with a outstanding peak in the beginning and fades away in a short period. Regarding the dragging action, it can be divided to two sub-actions: initial tapping and subsequent rubbing. It incurs a peak at the very beginning and causes a long lasting disturbance to the audio signals. It can be seen that the first step for our detection algorithm is to identify the tapping event.

In our implementation, we utilize a hypothesis testing approach called the generalized likelihood ratio test (GLRT) [82] to determine whether there is a tapping event. We store the incoming audio energy readings for 100ms in a First-In-First-Out buffer and keep testing the following two hypotheses on the data. The null hypothesis H_0 is that there is no tapping on the desktop and hence the average of the energy level should remain stable. Thus, we have

$$H_0 : u(t) = \mu_0 + w(t) \tag{4.7}$$

The variable $u(t)$ is the mean of the energy level before the time t. The μ_0 denotes the average energy and the variable $w(t)$ is the Gaussian noise with zero mean value and unknown variance σ_0.

The alternative hypothesis H_1 is that there is a tapping such that the mean of the energy level will significantly increase due to the tapping impulse. Thus, we have

$$H_1 = \begin{cases} u(t) = \mu_0 + w(t) & \text{if } t < t_c \\ u(t) = \mu_1 + w(t) & \text{if } t \geq t_c \text{ and } \mu_1 \neq \mu_0, \end{cases} \tag{4.8}$$

where the t_c represents the time when the tapping occurs, μ_1 denotes the new average energy after the tapping. Note that u_0, σ_0, μ_1, and t_c are unknown parameters.

where $p(u \mid H_1, \theta_1)$ is the probability density function of u under hypothesis H_1 and θ_1. The $p(u \mid H_0, \theta_0)$ is defined in a similar manner. The r is the threshold and set to 100 which works well in general cases.

However, the energy based detection system is likely to be disturbed by ambient noises. According to the field test, human voices or sudden random burst of sound nearby can frequently mis-trigger the detection system. To remove the false positives, we utilize a cross-checking approach that leverages a smartphone's gyroscope.

The algorithm takes advantage of an observation that a tapping on a desktop generates vibrations which could be easily captured by the gyroscope, while the ambient noises do not. Thus, whenever a tapping event is detected by the audio signal, we further check the presence of the vibrations in the gyroscope signals. To detect the vibrations, we adopt the GLRT algorithm in a similar manner to the audio signal. This approach significantly suppresses the false alarms caused by the ambient noises.

After a tapping event is detected, we further estimate the duration of the touch action in case there is a following dragging event. The duration is estimated between the point when the tapping occurs and the point when the power drops to below the threshold $\mu_0 + (\mu_{max} - \mu_0) * 10\%$. The μ_{max} is the maximum power level that the tapping impulse can reach. Capping the duration at the power level slightly above the normal noise energy helps the system combat the fluctuation of the noise energy floor.

It should be noted that the user is suggested to perform the touch actions gently, since a large movement of the finger may have negative impact on the finger tracking accuracy. In practice, the detection algorithm can achieve great performance when the user gently hits or rubs the desk using his nail tip.

4.6 Runtime Calibration and Adaptation

We assume that the smartphone is placed at an environment where the settings of ambient light including positions and luminance remain unchanged. However, in practice, the lighting conditions may vary slightly due to various reasons (e.g., a bulb may slightly decrease its brightness due to insufficient voltage), rendering the mapping function retrieved from the initial training outdated. In our system, rather than simply restarting the whole training phrase, we adopt a run-time calibration mechanism that enables the system to adapt to the minor change of the environment with a tiny effort. In some rare situations when the lighting conditions change drastically, the user can still manually restart the entire training phrase.

The system executes runtime adaptation by allowing the user to provide extra feedback on the localization results. When the user notice minor localization error, he could pause the tracking algorithm for a short period and manually input the correct coordinate by clicking on the right position on the touchscreen. Through this run-time calibration process, the system can harness extra training samples to adapt to the slightly varying environment. Note that we should carefully adjust the learning rate

α associated with the new sample. A tiny α may render the update useless, while a huge α may amplify the noise in the sample and make the mapping function unstable.

In our implementation, we design the α based on the localization error. Specifically, if the original mapping function $\hat{f}$ maps the s_n into a coordinate $\hat{l_n}$, which has a relatively small distance to the coordinate l_n provided by the user, it means the function performs well and the update rate α should remain low. Conversely, a high update rate is selected when the distance is large. Based on the principle, we set α as:

$$\alpha_t = L(||l_t - \hat{f}(s_t)||^2) \tag{4.9}$$

L is a shifted logistic function that can be expressed as:

$$L(x) = \frac{1}{1 + \exp(6 - 3x)} \tag{4.10}$$

The logistic function converts the localization error into a value between (0,1) and holds nice properties. First, when the localization error is pretty small (less then 0.5cm), the value remain relatively low. Second, as the localization error continuously increases, the alpha rate will increase significantly. Finally, if the localization error goes beyond a certain value (3cm), the function returns a α value close to 1.

4.7 System Evaluations

In this section, we provide detailed performance evaluation of UbiTouch in terms of finger tracking accuracy, touch gesture detection accuracy, and system energy efficiency. We conducted our experiments in three common scenarios with dim, normal, and strong lighting condition by five users. Then we showcase UbiTouch's effectiveness when it works in conjunction with a handwriting recognition system.

4.7.1 Accuracy of Finger Tracking

There are several underlying factors that may affect the tracking performance of UbiTouch. In this section, we first carried out a baseline test in a typical usage scenario. Then we consider the effect of two crucial factors including (1) ambient lighting condition and (2) the tilted angle of the smartphone.

Accuracy for baseline experiment

In the baseline experiment, instead of running UbiTouch on an ordinary desktop, we deploy the system on top of a Macbook Pro's physical touchpad whose size is 7.5 cm $\times$ 10.5 cm. It collects the finger locations and trajectories which serve as the ground truth.

We then carried out a total of 100 static finger localization tests in two different setup: (1) tracking merely with the CCA mapping function. (2) tracking with both the mapping function and particle filter.

With only the mapping function, the average localization error is 1.69 cm with a standard deviation of 0.53 cm. The CDF shown in Fig 4.10 reveals that over 80% points have an error of less than 2.1 cm. When the mapping function works in conjunction with the particle filter, the mean error reduces to 1.53 cm with a 0.42 cm standard deviation. The corresponding CDF demonstrates that 80% of the error values is now less than 1.8 cm. The performance improvement is attributed to the particle filter which takes extra historic information into consideration.

Besides, the performance can also be observed through the spatial error distributions shown in Fig 4.11. An obvious observation is that the remote spots far away from the PS and ALS sensor tend to possess larger error than the area around the sensors does. We investigate this phenomenon and discover that when the finger is not within the optimal detection range of the PS (10cm), the sensor readings are saturated with noises, which might severely degrade the localization performance. Noticing that the

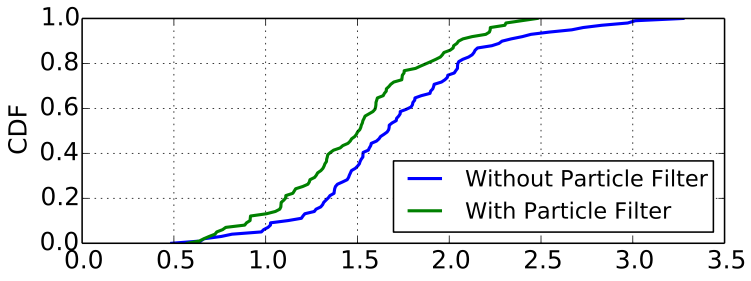

Figure 4.10: CDF for the localization error (cm).

transmission power of the PS is adjustable, we may be able to increase the detection range by consuming more energy.

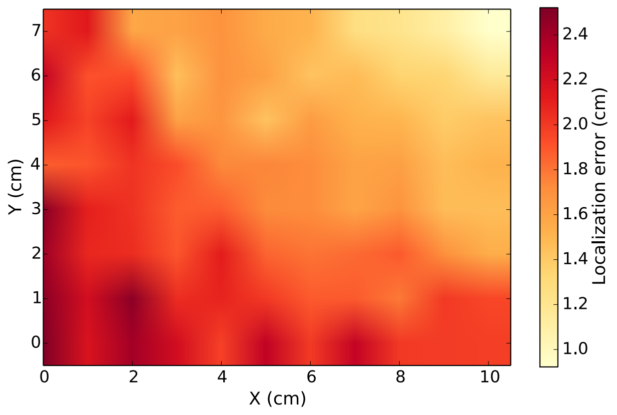

Figure 4.11: The spatial error distribution (interpolated). The position for the ALS and PS is around (12, 7.5).

Impact of ambient lighting condition

To measure the impact of the ambient lighting condition, we deploy our system in three typical scenarios including a storage room with a dim light bulb, an office room at campus with moderate light level, and an outdoor place with excessive sun light. Note that the settings (e.g., positions and illumination) of background light sources for these scenarios remain unchanged during our experiment. The experiments run on each scenarios, each repeated 100 times. Each time we conduct an experiment, we re-calibrate the system to achieve accurate performance measurements. Fig. 4.12 plots the corresponding localization accuracy. The results show that UbiTouch can maintain a localization error less than 1.54 cm across all the scenarios. It indicates that the system can perform well regardless of the illumination of the ambient lighting.

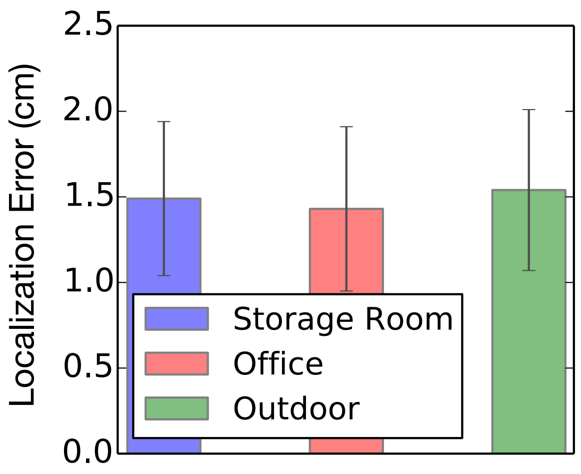

Figure 4.12: Localization accuracy under differnt scenearios.

Tilted Angle

In the previous experiments, the smartphone is deployed vertically on a desk. However, in practice, some users prefer to place their phones in a slightly tilted angle for a better user experience or perspective. Hence, we carried out multiple tests under different inclined angles, which can be directly measured by the built-in orientation sensors.

Fig. 4.13 demonstrates how the tracking accuracy changes with the titled angle. It can be seen that the localization error mildly increases when the titled angle starts rising. It indicates that the system still performs well

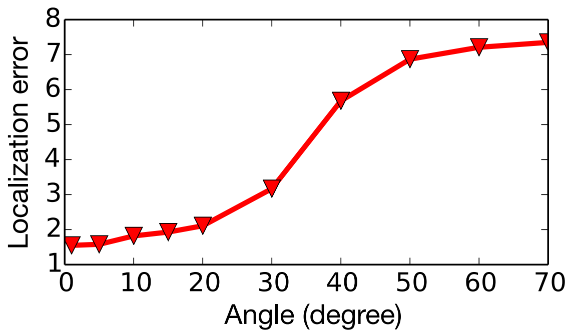

Figure 4.13: Localization error vs. smartphone inclined angle.

when the smartphone is deployed at a slightly tilted angle (less than 10 degrees). However, the slope drastically grows starting from 25 degrees and the localization error soars into xcm when the angle reaches 60 degrees. It may be due to that when the smartphone is significantly inclined, the AS and PLS turn into a direction far away from the finger, which has negative impact on the localization accuracy.

4.7.2 Accuracy of Touch Action Detection

We now evaluate the effectiveness of the tapping and dragging detection component. We carried out the test in three scenarios: a library, an office at campus, and a pub. These situations represent environments with small, normal, and extreme background noise respectively.

First, we measured the accuracy of the tapping detection. In every experiment, we instructed a user to tap on a desk to make 300 clicks, each on a randomly selected position. The tapping strength is maintained to be a moderate level which is audible to the user.

Table 4.2 display the experiment results for false positive (false alarm) and false negative (mis-detection) rate. We can notice that the error rates remains relatively low (3.5% in the worst case). The results indicate the

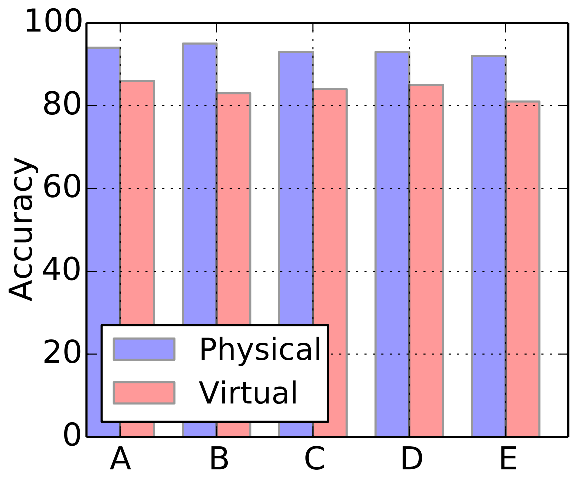

Figure 4.14: Recognition rates for 5 users.

tapping detection algorithm is accurate and reliable.

We conduct the experiments again without using the gyroscope. The results show that in a quiet environment including the library and office, the system performs equally well even with the gyroscope disabled. However, in a noisy pub environment, the false positive rate soars into 15%, making the system unusable. It indicates that the gyroscope can greatly suppress the false positive rate in some noisy situation and improve the performance.

Next, we evaluate the effectiveness of the duration estimation algorithm. We requested a user to perform a 10-second dragging action after a tapping. The experiments were repeated for 50 times in the aforementioned three situations. Table 4.2 shows the dragging duration estimation and demonstrates fair accuracy. Even in a noisy environment, the error is still below a reasonable value (0.5 seconds).

	Library	Office	Pub
Error rate with gyro (FP/FN)	0.5% (0/0.5)	0.8% (0/0.8)	3.5% (0.3/3.2)
Error rate w/o gyro (FP/FN)	0.7% (0.3/0.4)	1.3% (0.5/0.8)	21% (15/6)

Table 4.1: Accuracy of tapping detection.

	Library	Office	Pub
Est. Duration(sec)	9.7	10.2	10.5

Table 4.2: Accuracy of touch-event duration estimation.

4.7.3 Power Consumption

UbiTouch merely leverages built-in sensors including the PS, ALS, microphone, and gyroscope which are of low power consumption. Especially, the average operating current for the PS and ALS is only 176 μA [30]. We profile the power consumption of UbiTouch by utilizing a professional power monitor application called Powertutor [87]. Specifically, we measure the power cost in two situations: (1) idle with screen on (2) running UbiTouch with continuous touch input. Each situation lasts for 5 minutes. The average power consumption for each state is 6021 mW and 6930 mW respectively. Thus, UbiTouch incurs extra 15.1% power cost, that is slightly less than the power consumption of UbiK which is 18.5% [76].

4.7.4 Application Tests

UbiTouch can support various types of applications such as drawing and text-inputing. Here, we further evaluate the system performance by replaying the user input into the Google handwriting text-input application [48] and assessing the recognition rate. We carried out the experiments by requesting three users to input 100 random English alphabets. Fig. 4.15

shows two handwriting examples retrieved from the physical and virtual touchpads. Note that instead of feeding the finger trajectories to the application, we conduct some pre-processing to increase the recognition rate. Specifically, we apply a clustering algorithm called DBSCAN [83] to filter out some outliers and extract the skeleton of the input character, which is then fed into the recognition software.

Fig. 4.14 demonstrates the recognition rates for five users. The average recognition rate is 79%. In the worst case, the recognition rate for UbiTouch is 72%, still comparable to the accuracy of the physical touchpad which is 92%.

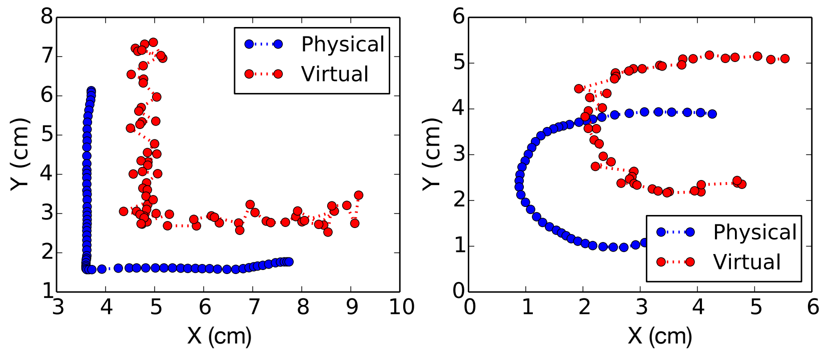

Figure 4.15: Handwriting examples for English characters L and C.

4.8 Conclusion

In this chapter, we present UbiTouch, a prototype system that enhances smartphones with virtual touchpads through the built-in smartphone sensors. We demonstrate that UbiTouch achieves centimeter-level localization accuracy and poses no significant impact on battery life of a smartphone.

Chapter 5

Conclusions and Suggestions for Future Research

In this chapter, we conclude this thesis and outline some possible future works.

5.1 Conclusions

The mobile gaming industry is involving constantly and rapidly. The game developers are now producing mobile games with increasingly immersive graphics. Nevertheless, the resource-hungry gaming tasks, which are formerly reserved for the desktop PCs, inevitably push the limit of the devices, causing the applications' running in a low frame rate and leading to a short battery lifetime. Besides, the minuscule touchscreens of smart devices also prevent players from smoothly interacting with devices as they can do with PCs. In this thesis, we propose a system framework for mobile gaming to address the challenging issues. We implement two systems that enable gaming application acceleration and provide a novel mobile interaction technology for game players. We analyzed the challenging issues while implementing these systems and demonstrated our solutions as well.

As for the gaming acceleration systems, we present GBooster, a system that accelerates GPU-intensive mobile applications by transparently offloading GPU tasks onto neighboring multimedia devices such as SmartTV and Gaming Consoles. We implemented a prototype system on the Android system and evauluate its performance with 6 mobile games of 3 geners. The results demonstrate that GBooster can boost applications' frame rates by up to 85%. In terms of power consumption, GBooster can achieve 70% energy saving compared with local execution.

To enable players' smooth interaction with mobile devices, we propose UbiTouch, a prototype system that enhances smartphones with virtual touchpads by merely utilizing the built-in energy-conservative sensors. UbiTouch trails a finger by analyzing the raw sensory data from the AS and PLS and detects typically touch actions such as tapping and dragging by leveraging the microphone and gyroscope sensor. We have evaluate our system in three scenarios with different lighting conditions by five users. The results show that UbiTouch achieves centimetre-level localization accuracy and poses no significant impact on the battery life. We envision that UbiTouch could support applications such as text-writing and drawing.

The two parts of the thesis heavily utilize techniques including signal processing and statistical analysis. We modified and adopt them based on the specific usage scenarios. By conducting extensive experiments, we show that the methodologies successfully solved the problems we encountered. As the methodologies are designed for general scenarios, we believe that the thesis could shed light on future related research.

5.2 Suggestions for Future Research

We close this thesis by providing some suggestions for the future research. Specifically, we believe that the following aspects are worth further investigation.

5.2.1 GBooster

GBooster has made some advances towards acceleration of GPU-intensive mobile applications. Still, it bears several limitations that needs further improvement.

Scenarios without Available Devices Although GBooster outweighs existing cloud-based solutions in terms of response time and frame rate, it does not imply that GBooster can take place of the cloud-based solutions. Under some rare circumstances where there is no available multimedia device nearby, the cloud-based platforms could still provide service to users.

Different Mobile Operating Systems. Our current prototype is only implemented on the Android operating systems. We are investigating how to enable GBooster in other mobile platforms such as iOS and Windows Phone. Since the iOS utilizes OpenGL ES as the Android does, we may be able to directly port GBooster to iOS. Although Windows Phone uses a different graphics API named *Direct X* [62], we could still utilize the same API hooking technique and implement the corresponding wrapper library to support it.

Towards Multiple Users. The prototype is designed to serve multiple users simultaneously. All the service devices maintain a queue buffering the incoming requests and submit them to GPU for execution in a First-Come-First-Served (FCFS) manner. However, it takes no consideration of the tasks' priorities, which could be problematic for time-critical applications. For instance, when an fast-paced shooting game and a chess game that requests thoughtful consideration for each movement are running simultaneously, requests from the shooting game should receive higher processing priorities in order to provide the player with a fast response time. We plan to purpose sophisticated scheduling algorithms to meet requirement from multiple users.

Experiment Settings. Another limitation lies in the straightforward experiment settings. We only conduct experiments in an ideal local area network with stable connectivity. More tests should be carried out in different networks.

5.2.2 UbiTouch

As for the UbiTouch system, it still has several limitations that are pending for further investigation.

System Generalizability: Currently, the system is only implemented on a specific phone model. We found that smart phones from different vendors tend to equip different types of sensors, which means our system may only be able to run in a specific model till now. However, since almost every sensor could provide raw distance and luminance readings. Thus, we envision that the principles and methodologies explored in the paper could be generalizable and applicable on a number of devices.

Applicable Scenarios: The prototype of UbiTouch is only applicable under two basic conditions: (1) the smartphone is placed on a desk in a vertical or slightly inclined position and remains still, and (2) throughout the usage life-cycle, the ambient lighting conditions do not vary drastically. Regarding the first condition, users have to deploy the smartphone almost vertically, which is a somewhat unnatural position. The second condition not only requires that the positions and luminance of light sources basically remain unchanged, but also demands that there is no significant movement in the neighboring area of the user. We concede the rigidness of these conditions and envision that more sophisticated signal processing techniques will be useful for eliminating these limitations.

Finger Gestures: While interacting with the system, the user is suggested to stay consistent in terms of finger gestures. A drastically change of the finger gesture (e.g., finger rotation/tilting) may negatively affect the tracking accuracy. We expect some pre-processing jobs to remove the impact of the gestures in the future.

Bibliography

[1] Candy crush. http://candycrushsaga.com/en/.

[2] Cut the rope. http://www.cuttherope.net.

[3] Dynamic linker hooking. http://man7.org/linux/man-pages/man8/ld.so.8.html.

[4] Ebook reader. https://goo.gl/5tVbvX.

[5] Final fantasy. https://play.google.com/store/apps/details?id=com.square.enix.android.googleplay.FFVII.

[6] From tight to turbo and back again: designing a better encoding method for turbovnc. http://www.virtualgl.org/pmwiki/uploads/About/tighttoturbo.pdf.

[7] Gaming device statistics. https://goo.gl/7TCTKT.

[8] Gta san andreas. https://play.google.com/store/apps/details?id=com.rockstargames.gtasa.

[9] Human being response time benchmark. http://www.humanbenchmark.com/tests/reactiontime/statistics.

[10] List of nvidia graphics processing units. http://goo.gl/qcJgqJ.

[11] Mcc forum. http://www.mobilecloudcomputingforum.com/.

[12] Modern combat 5: Blackout. `https://play.google.com/store/apps/details?id=com.gameloft.android.ANMP.GloftM5HM`.

[13] Onlive. `http://www.onlive.com`.

[14] Opengl es emulator. `http://malideveloper.arm.com/resources/tools/opengl-es-emulator/`.

[15] Opengl es example program. `https://goo.gl/Zrnez4`.

[16] Opengl framebuffer. `https://www.opengl.org/wiki/default-framebuffer`.

[17] Smart tv statistics. `https://goo.gl/wh5aFm`.

[18] Specification of nvidia shield. `https://goo.gl/vejzCK"`.

[19] Star wars: Kotor. `https://play.google.com/store/apps/details?id=com.aspyr.swkotor`.

[20] Tcp delay. `https://access.redhat.com/solutions/407743`.

[21] Tumblr. `https://goo.gl/rLr5HJ"`.

[22] Understanding and optimizing video game frame rates. `https://www.lifewire.com/optimizing-video-game-frame-rates-811784`.

[23] The walking dead: Michonne. `https://play.google.com/store/apps/details?id=com.telltalegames.walkingdeadm100`.

[24] Yahoo weather. `https://goo.gl/N9mLVY`.

[25] Global games market report, 2013. `https://goo.gl/4aK0GN/`.

[26] In less than two years, a smartphone could be your only computer, 2013. `http://www.wired.com/2015/02/smartphone-only-computer/`.

[27] ABDELNASSER, H., YOUSSEF, M., AND HARRAS, K. A. Wigest: A ubiquitous wifi-based gesture recognition system. In *2015 IEEE Conference on Computer Communications (INFOCOM)* (2015), IEEE, pp. 1472–1480.

[28] AIMAR, L., MERRITT, L., PETIT, E., CHEN, M., CLAY, J., RULLGRD, M., HEINE, C., AND IZVORSKI, A. x264-a free h264/avc encoder, 2005.

[29] AKAIKE, H. Akaikes information criterion. In *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 25–25.

[30] AVAGO. Avago 9930 integrated proximity and ambient light sensor. `http://www.avagotech.com/products/`.

[31] AVAGO-DEVELOPER. Avago 9930 linux driver. `https://github.com/CyanogenMod/android_kernel_lge_hammerhead/blob/cm-13.0/drivers/misc/apds993x.c`.

[32] BAUDISCH, P., AND CHU, G. Back-of-device interaction allows creating very small touch devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2009), ACM, pp. 1923–1932.

[33] BEN-ZUR, L. Developer tool spotlight-using trepn profiler for power-efficient apps, 2011.

[34] BI, X., LI, Y., AND ZHAI, S. Ffitts law: modeling finger touch with fitts' law. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), ACM, pp. 1363–1372.

[35] CAI, W., ZHOU, C., LEUNG, V. C., AND CHEN, M. A cognitive platform for mobile cloud gaming. In *Cloud Computing Technology and*

Science (CloudCom), 2013 IEEE 5th International Conference on (2013), vol. 1, IEEE, pp. 72–79.

[36] CHEN, S.-W., CHANG, Y., TSENG, P., HANG, C., AND LEI, C. Cloud gaming latency analysis: Onlive and streammygame delay measurement. In *Proceedings of the 19th ACM international conference on Multimedia* (2014), pp. 1269–1272.

[37] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems* (2011), ACM, pp. 301–314.

[38] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 273–286.

[39] COLLET, Y. Lz4: Extremely fast compression algorithm. *code. google. com* (2013).

[40] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 49–62.

[41] DEMARTINES, P., AND HÉRAULT, J. Curvilinear component analysis: A self-organizing neural network for nonlinear mapping of data sets. *Neural Networks, IEEE Transactions on 8*, 1 (1997), 148–154.

[42] DEVELOPERS, A. Monkeyrunner, 2015.

[43] DINH, H. T., LEE, C., NIYATO, D., AND WANG, P. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing 13*, 18 (2013), 1587–1611.

[44] FERNANDO, N., LOKE, S. W., AND RAHAYU, W. Mobile cloud computing: A survey. *Future Generation Computer Systems 29*, 1 (2013), 84–106.

[45] FLINN, J., PARK, S., AND SATYANARAYANAN, M. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 217–226.

[46] FRAUENFELDER, M. G-cluster makes games to go. *The Feature: It's All About the Mobile Internet, http://www. thefeaturearchives. com/13267. html 3* (2001).

[47] GOOGLE. Environment sensor for android. `http://developer.android.com/guide/topics/sensors/sensors_environment.html`.

[48] GOOGLE-INPUT. Google handwriting input. `https://play.google.com/store/apps/details?id=com.google.android.apps.handwriting.ime`.

[49] GU, Y., AND GROSSMAN, R. L. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks 51*, 7 (2007), 1777–1799.

[50] HALPERIN, D., GREENSTEIN, B., SHETH, A., AND WETHERALL, D. Demystifying 802.11 n power consumption. In *Proceedings of the 2010 international conference on Power aware computing and systems* (2010), p. 1.

[51] HAMILTON, J. D. *Time series analysis*, vol. 2. Princeton university press Princeton, 1994.

[52] HARRISON, C., BENKO, H., AND WILSON, A. D. Omnitouch: wearable multitouch interaction everywhere. In *Proceedings of the 24th annual ACM symposium on User interface software and technology* (2011), ACM, pp. 441–450.

[53] HINCKLEY, K., AND SONG, H. Sensor synaesthesia: touch in motion, and motion in touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), ACM, pp. 801–810.

[54] HIRAOKA, S., MIYAMOTO, I., AND TOMIMATSU, K. Behind touch, a text input method for mobile phones by the back and tactile sense interface. *Information Processing Society of Japan, Interaction 2003* (2003), 131–138.

[55] HOQUE, M. A., SIEKKINEN, M., KHAN, K. N., XIAO, Y., AND TARKOMA, S. Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys (CSUR) 48*, 3 (2016), 39.

[56] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. Timegraph: Gpu scheduling for real-time multi-tasking environments.

[57] KEMP, R., PALMER, N., KIELMANN, T., AND BAL, H. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services* (2010), Springer, pp. 59–79.

[58] KETABDAR, H., YÜKSEL, K. A., AND ROSHANDEL, M. Magitact: interaction with mobile devices based on compass (magnetic) sensor. In *Proceedings of the 15th international conference on Intelligent user interfaces* (2010), ACM, pp. 413–414.

[59] LANE, N. D., MILUZZO, E., LU, H., PEEBLES, D., CHOUDHURY, T., AND CAMPBELL, A. T. A survey of mobile phone sensing. *Communications Magazine, IEEE 48*, 9 (2010), 140–150.

[60] LIU, H., SHAH, S., AND JIANG, W. On-line outlier detection and data cleaning. *Computers & chemical engineering 28*, 9 (2004), 1635–1647.

[61] LOVE, R. *Linux kernel development*. Pearson Education, 2010.

[62] LUNA, F. *Introduction to 3D game programming with DirectX 10*. Jones & Bartlett Publishers, 2008.

[63] LUO, W., AND BILLINGS, S. Adaptive model selection and estimation for nonlinear systems using a sliding data window. *Signal Processing 46*, 2 (1995), 179–202.

[64] LV, Z., HALAWANI, A., LAL KHAN, M. S., RÉHMAN, S. U., AND LI, H. Finger in air: touch-less interaction on smartphone. In *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia* (2013), ACM, p. 16.

[65] MUNSHI, A., GINSBURG, D., AND SHREINER, D. *OpenGL ES 2.0 programming guide*. Pearson Education, 2008.

[66] NIRJON, S., GUMMESON, J., GELB, D., AND KIM, K.-H. TypingRing: A wearable ring platform for text input. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 227–239.

[67] OAKLEY, I., AND LEE, D. Interaction on the edge: offset sensing for small devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), ACM, pp. 169–178.

[68] RISTIC, B., ARULAMPALAM, S., AND GORDON, N. *Beyond the Kalman filter: Particle filters for tracking applications*, vol. 685. Artech house Boston, 2004.

[69] ROBINSON, S., RAJPUT, N., JONES, M., JAIN, A., SAHAY, S., AND NANAVATI, A. Tapback: towards richer mobile interfaces in impoverished contexts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), ACM, pp. 2733–2736.

[70] ROEBER, H., BACUS, J., AND TOMASI, C. Typing in thin air: the canesta projection keyboard-a new method of interaction with electronic devices. In *CHI'03 extended abstracts on Human factors in computing systems* (2003), ACM, pp. 712–713.

[71] SOLUTIONS, M. Power monitor, 2016.

[72] SOTO, A. Self adaptive particle filter. In *IJCAI* (2005), pp. 1398–1406.

[73] TANG, S., YOMO, H., KONDO, Y., AND OBANA, S. Wake-up receiver for radio-on-demand wireless lans. *EURASIP Journal on Wireless Communications and Networking 2012*, 1 (2012), 1–13.

[74] VILLASENOR, J. D., BELZER, B., AND LIAO, J. Wavelet filter evaluation for image compression. *Image Processing, IEEE Transactions on 4*, 8 (1995), 1053–1060.

[75] WANG, J., VASISHT, D., AND KATABI, D. RF-IDraw: Virtual Touch Screen in the Air using RF Signals. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 235–246.

[76] WANG, J., ZHAO, K., ZHANG, X., AND PENG, C. Ubiquitous keyboard for small mobile devices: harnessing multipath fading for fine-grained keystroke localization. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services* (2014), ACM, pp. 14–27.

[77] WANG, S., AND DEY, S. Modeling and characterizing user experience in a cloud server based mobile gaming approach. In *Global*

Telecommunications Conference, 2009. GLOBECOM 2009. IEEE (2009), IEEE, pp. 1–7.

[78] WANG, S., AND DEY, S. Addressing response time and video quality in remote server based internet mobile gaming. In *2010 IEEE Wireless Communication and Networking Conference* (2010), IEEE, pp. 1–6.

[79] WANG, S., AND DEY, S. Rendering adaptation to address communication and computation constraints in cloud mobile gaming. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE* (2010), IEEE, pp. 1–6.

[80] WEN, Y., ZHANG, W., GUAN, K., KILPER, D., AND LUO, H. Energy-optimal execution policy for a cloud-assisted mobile application platform. *Nanyang Technol. Univ., Singapore, Tech. Rep* (2011).

[81] WIGDOR, D., FORLINES, C., BAUDISCH, P., BARNWELL, J., AND SHEN, C. Lucid touch: a see-through mobile device. In *Proceedings of the 20th annual ACM symposium on User interface software and technology* (2007), ACM, pp. 269–278.

[82] WILLSKY, A. S., AND JONES, H. L. A generalized likelihood ratio approach to the detection and estimation of jumps in linear systems. *Automatic Control, IEEE Transactions on 21*, 1 (1976), 108–112.

[83] WITTEN, I. H., AND FRANK, E. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[84] YANG, X., PAN, T., AND SHEN, J. On 3g mobile e-commerce platform based on cloud computing. In *Ubi-media Computing (U-Media), 2010 3rd IEEE International Conference on* (2010), IEEE, pp. 198–201.

[85] YU, N.-H., TSAI, S.-S., HSIAO, I.-C., TSAI, D.-J., LEE, M.-H., CHEN, M. Y., HUNG, Y.-P., ET AL. Clip-on gadgets: expanding multi-touch interaction area with unpowered tactile controls. In *Proceedings*

of the 24th annual ACM symposium on User interface software and technology (2011), ACM, pp. 367–372.

[86] ZHANG, C., TABOR, J., ZHANG, J., AND ZHANG, X. Extending mobile interaction through near-field visible light sensing. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 345–357.

[87] ZHANG, L., TIWANA, B., QIAN, Z., WANG, Z., DICK, R. P., MAO, Z. M., AND YANG, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (2010), ACM, pp. 105–114.

[88] ZHANG, Q., CHENG, L., AND BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications 1*, 1 (2010), 7–18.