Exploring Human-Computer Interaction Through Bluetooth Low Energy Enabled Human Interface Devices

A Senior Project
presented to
the Faculty of Liberal Arts and Engineering Studies
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts in Liberal Arts and Engineering Studies

by
Tyler Durkin
December 2013

1. Introduction

With the prevalence of digital interactive consumer and enterprise technology, a larger focus is being placed on how people interact with their computers. Historically, productivity hinged solely on one's ability to complete work, but now an employee's ability to control their computer greatly influences how much he or she is able to accomplish in a typical work day. Both design and engineering professions are seeing a shift in interest toward the end user experience. With this shift the importance of the interaction design field is becoming increasingly apparent. The Interaction Design Foundation penned a succinct description of the profession: "Interaction design is about shaping digital things for people's use."[1] Bill Verplank, who originally coined the term interaction design, stated that it was the adaptation of user interface design to industrial design[2], unmistakably highlighting the importance of both the physical and digital. While most interaction design today focuses on the digital user interface design, physical world human-computer interaction is arguably the most important consideration when creating a new product.

Since the development of a graphical user interface for computers, input devices have had very little growth. Ever since the original graphical user interface (GUI) was first displayed in 1968 by Douglas Englebart[3], a computer scientist at the Stanford Research center with backing from the United States Air Force, the mouse has been the primary form of input. Efforts have been made to improve upon the mouse, and new devices were invented, but most of those fell by the wayside as people reverted back to a familiar standard. The only invention that has enjoyed a prolonged existence is the trackpad due to its inclusion in all laptops. The creation of so many different input devices demonstrates a need to improve on the existing pointing devices, but there has yet to be a general consensus on the future mouse replacement.

A wave of new technologies in recent years has brought with it new interactions, and out of those needs to come new human input devices. This paper will outline from ideation to creation a new device that combines existing technologies in an effort to change the way consumers use their computers. It will show how with the combination of touchscreen technology, gesture recognition, and the Bluetooth LE wireless standard, the project aims to affect computer interaction. The paper will also discuss how the project communicated with a central computer to control it, how it was implemented, and the feedback from preliminary testing.

2. Background

The original GUI displayed in 1968 was simple; it could only display lines and text, with the alphabet limited to uppercase letters. It was incredibly rudimentary when compared to anything that has been created in the past 20 years, but at the time was completely new and revolutionary. However, the first descriptions of a modern GUI came before Englebart's demo. Vannevar Bush wrote a paper in the 1930s (and then revised it in 1945) that described a device consisting of several touch screen displays, a keyboard, and a scanner[3]. While all that was a dream in the 1930s — digital computers had yet to be invented — it is a perfect description of the computing technology available to consumers today. Tablets are the fastest growing segment of personal computing technology, and all of them come equipped with touch screen interfaces for navigation. The recommended minimum system requirements for the Windows 8 operating system lists a multi-touch screen for a display. While this shows that more and more computers are moving toward touch screen technology and realizing Bush's vision, the general consensus is that touch screens on computers are impractical and generally gimmicky. The upside of touch screens embedded directly into computers is that consumers are able to directly interact with on screen content. Much of the reason why the iPad took off in popularity despite it launching with a high price, few apps, and a stripped down mobile operating system was the experience. It spawned a new category of devices that few had ever used before. Holding an iPad and getting to touch the Internet, control apps with one's fingers, and directly manipulate on screen content in a way the smaller iPhone screen couldn't do turned many skeptics into believers. When both the iPhone and the iPad took off, Apple began to migrate its multi-touch technology from the iPhone to their laptop line, bringing gesture support to trackpads for more personal experience. Even so, there is demand for new devices that harness modern technology to revolutionize how consumers interact with their computers on a daily basis.

When Apple brought the multi-touch technology from their iPhone and iPad lines to their laptops they also brought new gesture based interactions utilizing as many as four fingers at one time. Apple has even had success bringing the trackpad platform to the desktop with the launch of their Magic Trackpad in 2010. They have sold hundreds of thousands units, with roughly half of new iMac owners opting for the Magic Trackpad as opposed to the Magic Mouse it comes configured with by default. The Trackpad allows for interactions with desktops that include swiping, pinching, dragging, and rotating, among others. However, even this does not go far enough. There are multiple downloadable applications such as MagicPrefs[4] and Better Touch Tool[5] that extend the amount of gestures supported by the Trackpad and bring support to other applications and system functions. Apple's online store sells sticker overlays with supporting software to simulate a number pad and calculator. Support has been overwhelming; consumers love the Magic Trackpad, but they want more.

The Magic Trackpad was a leap of faith when originally introduced; it was unknown how consumers would react to a standalone unit made for the desktop. Trackpads have been criticized for being less accurate and slower than a mouse, but the ability for consumers to trigger system functions with a flick of the wrist quickly gained traction and made the Magic Trackpad incredibly successful. Even though it was brought to market only 3 years ago, its slim form factor pushed the limits of technology. In the few years since its launch, however, technology has

advanced to the point where more can be incorporated into the same package and take the user experience to new heights.

3. Proposal

Figure 1. Magic Launchpad concept

The project has been dubbed the Magic Launchpad, a device named after the built-in OS X app Launchpad and based off of the existing Magic Trackpad, but incorporating a multitouch display for customizable onscreen content. The purpose of including a touch screen is twofold. First, it enables more flexibility, with the device pairing with the computer to show more relevant and meaningful content. Second, it puts touch functions where they belong: at the user's fingertips. The original idea for the Magic Launchpad UI is one composed of several screens, with the displayed content changing based on what the computer is doing and what the user wants to do. The default screen can be blank to save energy, or display customized wallpaper, and is nothing more than a standard trackpad. Beyond that, however, the Magic Launchpad begins to drastically diverge from the Magic Trackpad. Separate screens can be used to show a grid of computer app icons for quick launching, and another grid of icons for launching onboard Magic Launchpad widgets such as a number pad or calculator. With a developer (SDK) similar to that for iOS[6] the possibilities are endless, not just for onboard apps and widgets, but also for expanding functions and interactions with OS X applications. One other possible screen can take the OS X Notification Center off the computer screen and put it down below, for easy viewing and one click access to notification items that need attention. With a strategy like this, the potential of such a device is limited only to the imagination of developers.

For many years studies have shown that multiple monitors increase productivity[7] by allowing the user to see more content at one time[8]. There does get to be an extent where that content can be distracting and decrease productivity, which is why the Magic Launchpad is so useful: it moves content off screen to not only free up more space, but also to reduce distraction allowing for a more productive workflow. The human brain takes a significant amount of time to switch focus and adjust to a new activity. With the Magic Launchpad typical on screen clutter is reduced, but moved to a convenient location at one's fingertips.

The original motivation for the Magic Launchpad came after realizing the Launchpad app would be much easier to use if it was mirrored onto the iPad, with the hand able to quickly and easily launch programs. Instead, it currently takes a keystroke to open Launchpad, then find the application, and then a click to actually launch it; it's a lot of time spent on a task that is done many times a day. The Magic Launchpad can call up the familiar grid of app icons with the same

four finger pinch gesture the Magic Trackpad uses, and allow consumers to open applications in under a second. While users could also physically launch applications by using a touch screen display, the Magic Launchpad cuts down on time by placing the screen next to the keyboard, making it a natural extension with quicker access. The distance from keyboard to trackpad is a fraction of the distance to the screen, and the motion to the right is familiar and natural. Without affecting usability, the Magic Launchpad would add an incredible amount utility.

4. Technology

The concept and original design of the Magic Launchpad device was based around a hardware accessory. However, access to many of the components needed is restricted to manufacturing companies and getting the necessary parts is not very feasible. Using a micro controller such as the Arduino or RaspberryPi was considered, but touchscreen accessories are limited and of such low quality they were quickly ruled out. Instead, Apple's iOS platform proved to be a better and more reliable platform for creating a Magic Launchpad proof of concept. Apple's iPhone and iPad product lines include the necessary hardware technology, and Apple publishes an iOS SDK to allow for the creation of custom apps. Utilizing Apple technology brought the Magic Launchpad to life. The end result was a custom iOS app with a companion application running on whichever computer the user wished to control. In order to build both apps, however, a number of frameworks needed to be harnessed. The project made extensive use of Core Bluetooth and Quartz, and laid the foundation for future expandability through IOKit and Apple Script.

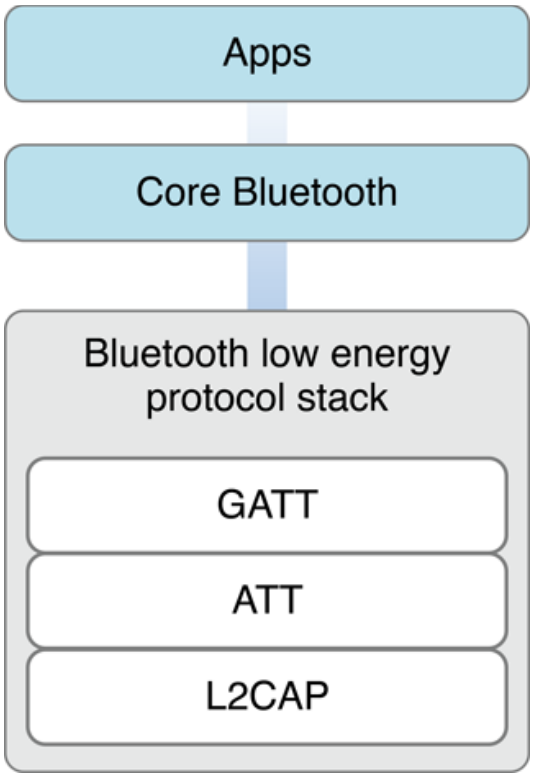

Figure 2. Core Bluetooth framework structure

4.1 Core Bluetooth

Core Bluetooth[9] provided the backbone of both the iOS and Mac apps by utilizing a new technology known as Bluetooth Low Energy[10]. Introduced a few years ago Bluetooth LE allows devices to communicate over short distances with minimal power consumption. While Bluetooth has been around for many years in devices such as wireless headsets, speakers, keyboards, and mice, the new low energy implementation operates slightly differently. Rather than having a

continuously open stream of information, Bluetooth LE intermittently sends packets of data from the peripheral device to the central device. Furthermore, rather than the central continually polling the peripheral, the peripheral will push the new data packets to the central device. These two operational changes are the cornerstone of the Bluetooth LE specification. In order to assist third-party developers in implementing Bluetooth LE in their apps and accessories, Apple added Core Bluetooth to their SDK at the same time they released the iPhone 4S, their first Bluetooth LE equipped device. In order to use Bluetooth, it is necessary to have a properly configured hardware and software stack. The software for this stack runs at a very low level, and programming it can be a challenge. Core Bluetooth, however, gives developers a library of functions to incorporate into an app making programming more straightforward. While Core Bluetooth functions that run on both the iOS and Mac sides of the application and are built to communicate together, it is not a trivial task to make everything work properly. Using this framework, however, provided the basis for reliable, short range data transfer and made the rest of the application functions possible.

4.2 Quartz

Quartz is actually made of several technologies packaged into OS X that are responsible for the rendering and displaying of graphics content. Most important to this project is that Quartz is responsible for drawing the on screen content, which includes cursor position. The Magic Launchpad Mac application made ample use of Quart Display Services[11] to adjust the cursor position on screen, which, when paired with the Quartz Even Services[12] allows for mouse clicks as well. While Core Bluetooth formed the basis for app communication, Quartz and its multiple libraries formed the basis for computer navigation.

4.3 IOKit

In contrast to Core Bluetooth, IOKit[13] has been around since the beginning of Apple's SDK. As the name suggest, the framework allows developers to programmatically control device I/O and gives them the ability to manage kernel functions without having to worry about intricate details. The kernel is the piece of the operating system that dictates how all pieces of hardware, internal and external, interact with each other and with any installed software. It is a small, yet remarkably powerful and complex part of the operating system. As is to be expected with such integral program, missteps, whether intentional or accidental, can have catastrophic consequences. To help prevent issues, Apple developed IOKit giving programmers a set of high level functions which in turn properly executes the corresponding low level kernel functions. This implementation introduces another layer of abstraction and therefore another layer of security. While not currently implemented, IOKit is an important framework and will be able to expand the capabilities of the Magic Launchpad by providing access to hardware functions that other pointing devices make use of.

4.4 AppleScript

AppleScript[14] is a programming language created by Apple, specifically for its Macintosh operating systems. The language makes use of inter-application communication through AppleEvents, and is able to automate complex tasks. For this project, the most important aspect

of AppleScript is its ability to execute AppleEvents. Since AppleScript was developed by Apple and is built in to the OS X operating system, scripts are able to be run from inside applications. The Magic Launchpad Mac application can make use of this, and is how it is able to launch other applications and execute functions with the touch of a button.

5. Design

From a design perspective the project was fairly straightforward. Following Dieter Rams' ten principles for good design[15] the aim was to create an interface that was as unobtrusive as possible. Since the functionality of the application resides in the background, there was a minimal amount of visual elements to consider.

In order to reduce the amount of onscreen clutter on the computer, the Mac application was relegated to the menubar in the form of what is known as a menulet. When launched it is a single icon in the right side of the menu bar, next to where the date and time reside. Clicking it presents a drop down menu with the only 3 options necessary: connect, start, and quit.

Figure 3. The only Mac UI element: the menulet

On the iOS side the application interface has more to deal with in the foreground. The app is always on whenever it is open, and off whenever a user returns to the iOS springboard, so the menu needed in the Mac application was unnecessary. Instead, the main view is a simple, grey rectangle used to track gestures and simultaneously transmit them back to the computer for interpretation. In the bottom right hand corner is a small plus button, used to flip the main screen over and reveal the application launcher underneath (it can also be accessed with a 2 finger double-tap). This type of segue is not ideal, nor is the gesture to get to the app launcher, however it was by far the easiest to implement so focus could be placed on making the app launcher work rather than creating a flashy segue.

Figure 4. The Magic Launchpad main screen

6. Implementation

In order for the project to work successfully from a technical perspective, both the iOS device and Mac had to communicate seamlessly; the iOS app needed to transfer data to the computer which, in turn, the computer could manipulate as needed. A first glance at the Apple documentation makes this look like a relatively simple task:

Figure 5. Simple client-server Bluetooth configuration

A simple communication between server and client, also know as the peripheral and central, respectively, needs a lot of different parts working together in the background to be successful. A diagram of the Mac application organization is shown below:

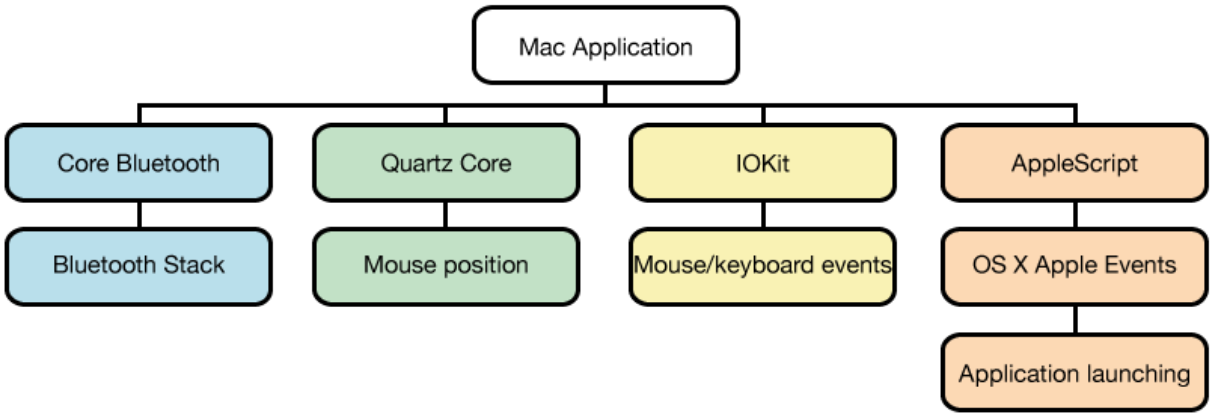

Figure 6. Organization of Magic Launchpad menulet

As the diagram suggests, the four separate technologies work independently of one another, but are all controlled by a central service. It should be noted, however, that the diagram can be misleading. While the frameworks are separate and handle processing their respective data individually, the connecting lines are not one-way data paths as one might assume. Data is constantly being transferred back and forth from the central service to each framework. The main Mac application can be thought of as a hub, shuttling data to the right framework, and then passing on the resulting output to a different framework as necessary.

Since most of the processing and data handling is done on the Mac side, the iOS app is only responsible for gathering and transferring information. As such, the application organization looks much simpler:

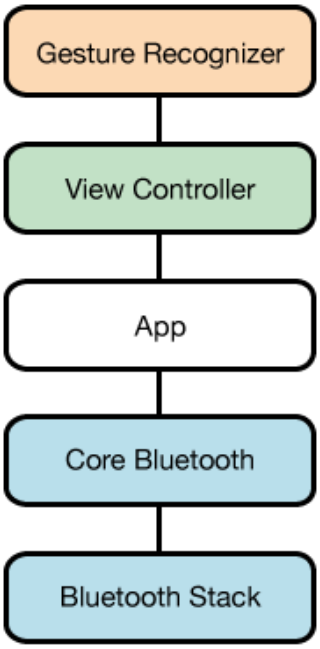

Figure 7. Organization of Magic Launchpad iOS app

The one difference between the two sides, central and peripheral, is that the peripheral is responsible for broadcasting the fact it has data, technically known as advertising its services. In order for the central device to find these services, the peripheral must first build a service tree. This tree, as shown below, can consist of many services with multiple characteristics each.

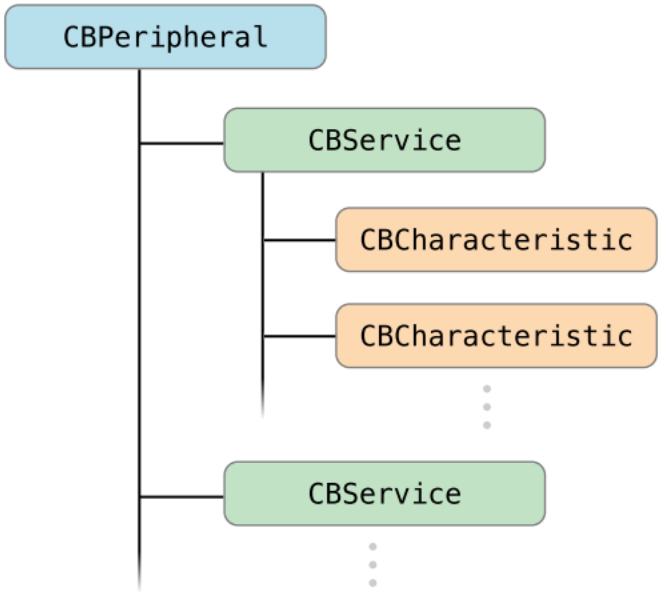

Figure 8. Core Bluetooth service tree

Initially, the Magic Launchpad only has one service providing information on mouse position and mouse clicks. As the app expands more services will need to be added to handle additional functionality such as application launching. The way the service tree is built programmatically, though, will make adding more services and characteristics fairly easy in the future; the biggest issue is setting up the tree initially, adding more services requires fewer lines of code.

```
- (void) peripheralManagerDidUpdateState:(CBPeripheralManager *)peripheral {
    if (peripheral.state == CBPeripheralManagerStatePoweredOn) {

        NSLog(@"self.peripheralManager powered on");

        // set up tree
        self.mousePointCharacteristic = [[CBMutableCharacteristic alloc] initWithType:[CBUUID UUIDWithString:MOUSE_POINT_UUID]
                                                                            properties:CBCharacteristicPropertyNotify
                                                                                 value:nil
                                                                           permissions:CBAttributePermissionsReadable];

        CBMutableService *launchpadMutableService = [[CBMutableService alloc] initWithType:[CBUUID UUIDWithString:LAUNCHPAD_SERVICE_UUID]
                                                                                    primary:YES];

        // Add characteristic to service
        launchpadMutableService.characteristics = @[self.mousePointCharacteristic];

        // publish service
        [self.peripheralManager addService:launchpadMutableService];
        NSLog(@"Service added");

        // begin advertising
        [self.peripheralManager startAdvertising:@{CBAdvertisementDataServiceUUIDsKey:@[launchpadMutableService.UUID],
            CBAdvertisementDataLocalNameKey:@"iOS Magic Launchpad"}];
        NSLog(@"Now advertising as %@", CBAdvertisementDataLocalNameKey);
    }
}
```

Figure 9. Setting up the Magic Launchpad service tree

Correctly configuring Core Bluetooth was by far the biggest challenge in creating the Magic Launchpad application. Getting the two devices to successfully communicate and transfer data was quite time consuming, but once done, the rest was simpler to implement. The pseudocode for both the Mac and iOS apps illustrate why it was so critical to get Core Bluetooth working before other features were implemented.

```
CONNECT BUTTON PRESSED
        INITIALIZE BLUETOOTH CENTRAL MANAGER

CENTRAL MANAGER INITIALIZED
        SCAN FOR AVAILABLE PERIPHERALS

IF ACCEPTABLE PERIPHERAL IS DISCOVERED
        CONNECT TO PERIPHERAL
        DISCOVER PERIPHERAL SERVICES

IF DATA IS RECEIVED
        IF MOUSE CURSOR POSITION DATA
                MOVE MOUSE TO POSITION
        IF CLICK DATA
                CREATE MOUSE CLICK EVENT

QUIT BUTTON PRESSED
        DISCONNECT PERIPHERAL DEVICES
        TERMINATE APPLICATION
```

Figure 10. Mac application pseudo code

```
APP LAUNCHED
        INITIALIZE BLUETOOTH PERIPHERAL MANAGER

PERIPHERAL MANAGER INITIALIZED
        BEGIN ADVERTISING SERVICES

IF CENTRAL CONNECTS
        IF TOUCH DETECTED
                SEND MOUSE POSITION DATA
        IF TAP DETECTED
                SEND MOUSE CLICK DATA
```

Figure 11. iOS app pseudo code

The other major task the iOS app is responsible for is tracking touches and gestures while simultaneously sending them to the central. Each screen in an iOS app is managed by a view controller, and in order to track touches, each view controller had a gesture recognizer[16] bound to it. The gesture recognizers respond to very specific touch patterns, so multiple were bound to each view in order to handle the different gestures that might occur. Figure 12 below illustrates how a gesture recognizer is bound to a view and in turn passes information on to the view controller. Coupled with the pseudo code in figure 11, one can see how the iOS side operates by first pairing with the Mac, then monitoring for touch data and sending that data on to the computer. The data itself is converted to a string, since Core Bluetooth only supports sending certain data types, then once on the computer the application converts it from a string back into the appropriate type to process.

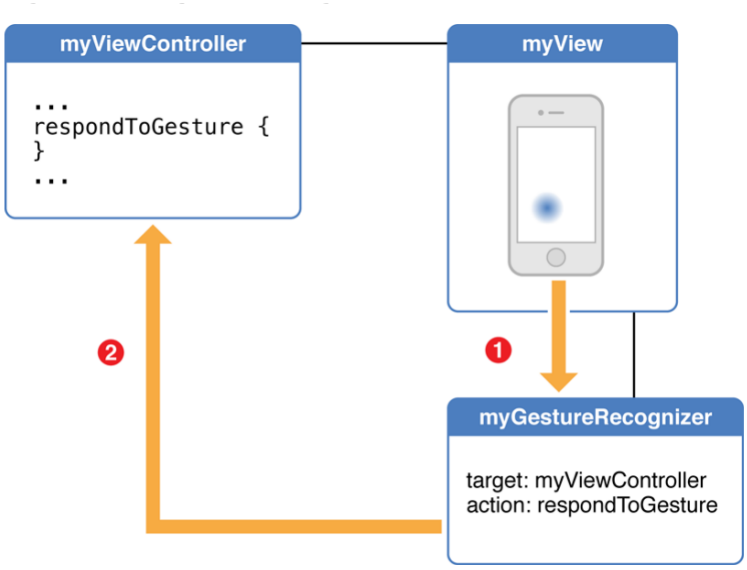

Figure 12. View - Gesture Recognizer - View Controller feedback loop

Gesture	UIKit class
Tapping (any number of taps)	`UITapGestureRecognizer`
Pinching in and out (for zooming a view)	`UIPinchGestureRecognizer`
Panning or dragging	`UIPanGestureRecognizer`
Swiping (in any direction)	`UISwipeGestureRecognizer`
Rotating (fingers moving in opposite directions)	`UIRotationGestureRecognizer`
Long press (also known as “touch and hold”)	`UILongPressGestureRecognizer`

Figure 13. Different types of gesture recognizers and their uses

The implementation chosen does present drawbacks. Needing two applications running simultaneously is clearly a limitation, however, more research may yield a solution to this issue. The second limitation of using Core Bluetooth is that it will only work between Apple devices, and furthermore only recent devices since Bluetooth LE is so new. While the goal of this project is to create a new interface that will work with all major platforms, the design chosen will ultimately limit the amount of devices Magic Launchpad will work with. However, for testing, validation, and proof-of-concept purposes, the fact that it will initially only work with Apple products is not an issue. The major benefit of it only being able to run on Apple hardware is that it can be optimized for each device it is running on. The Magic Launchpad provides a positive environmental impact as well by using existing devices rather than requiring mass production of new hardware. The last benefit of implementing it as an app is that functionality will be easy to expand in the future simply by adding new Bluetooth services. As mentioned previously, the biggest challenge was getting the two devices to communicate and initialize the service tree, since that is already done extension and expansion will be fairly painless. It will require modification of code on both platforms, the iOS side to send new data types, and the Mac side to correctly handle those data types, but that requires a much smaller amount of work than was necessary to get the app up and running.

7. Analysis

Usability studies revolve around a principle known as Fitts's Law[17]. Proposed by Paul Fitts in 1954, it predicts that the time required to rapidly move to a target area is a function of the distance to the target and the size of the target itself. This model has been used to analyze pointing, either in the physical world, or more commonly, on digital elements. In practice, measured data has a straight line fit to the graph of Fitts's Law with a correlation coefficient of 0.95 or higher, making the mathematical model an incredibly accurate representation. Without having equipment able to measure reaction time in milliseconds or a substantially large data set, however, it is impossible to obtain quantitative data with statistical significance. Instead, analysis was subjective, done by using volunteers to determine if the controller makes a difference in interacting with a computer. The test involved different types of input devices for users to test, along with different tasks to perform on each. Some of these tasks were the same to directly compare the devices, while others were input device specific. In order to minimize the effects of muscle memory, task order was shuffled so that the user was unable to predict, react, and therefore complete tasks faster on later devices.

Testing feedback on the application was mostly positive. The most frequent comment was regarding how the device implements tracking and cursor movement. Standard trackpads use relative positioning, meaning that the computer ignores the exact XY position of a finger on the trackpad, and instead looks at the direction in which the finger is moving in order update cursor position. The Magic Launchpad implementation, however, uses absolute positioning, which means that a finger press in the top left corner of the trackpad will immediately place the cursor into the top left corner of the computer screen, regardless of where the cursor was located previous to said finger press. Absolute positioning makes moving large distances faster, however, almost all input devices are based on relative positioning so it does take some time to adjust. Going into the tests it was assumed the absolute positioning would give an advantage to the Magic Launchpad application. In reality, however, performance between the Magic Launchpad and a standard trackpad was negligible when evaluating point and click operations. Users were able to get the cursor close to the target instantly, but it took more time to land directly on the target resulting in similar performance between the Magic Launchpad and the built in laptop trackpad. Overall, volunteers still chose the mouse as their go-to pointing device, however the promise of the Magic Launchpad was appealing and they would be interested in revisiting it when it is more polished.

8. Societal Impacts

The Magic Launchpad has the potential to make a large impact, and not just from an interaction standpoint. Businesses are always striving for efficiency and productivity. As discussed earlier, the ramifications of second screens have been studied, and even the small amount of space the Magic Launchpad contributes would make a big difference. Furthermore, customizable software and the core of the project allows users to create their own optimal setup. Not everything is a one-size-fits-all solution, and when it comes to usability, preferences vary greatly. Along with being able to customize it for personal preferences, it would also allow customization for those with special needs. Accessibility needs are a major area of concentration for all types of software, but hardware has yet to follow suit. For those who struggle with motor skills, the typical mouse and keyboard setup can present problems. Having a solution that can be customized to the needs of its users — be it employees looking for an optimal work solution, or ordinary consumers with accessibility issues — is needed in the marketplace and will have a large impact on computer interactions.

9. Related Work

For the most part, the Magic Launchpad is a unique device. However, gaming company Razer recently introduced their SwitchBlade interface[18]. Known for creating gaming keyboards and mice, the SwitchBlade interface is a touchscreen controller built into they keyboard. This is a similar concept to the Magic Launchpad as they both have small touch screens at finger's length. The two differ, however, in their purpose. The SwitchBlade is designed to have a flexible and customizable interface bringing various hotkey functions to games. On top of that, the SwitchBlade UI is only made to work with Windows, while the Magic Launchpad is aimed at Mac users. The Magic Launchpad, aims to be a standalone peripheral combining a pointing device, application launcher, and app platform. Beyond the SwitchBlade, however, there are no others product that provides similar capabilities as the Magic Launchpad, or one that uses a touchscreen as an external controller.

10. Conclusion

As with any business venture and new product, time is always a major issue. With only two quarters of senior project and a summer in between, there is not enough time to produce fully polished, professional level product. While it gives ample time to design and create a rudimentary Magic Launchpad app, there is plenty that could be done with more time. One of the biggest issues with the app as it stands now is the lag between swiping on the iOS screen and the cursor moving on the computer screen. In order to reduce this lag, more research needs to be conducted on utilizing multiple threads simultaneously. Currently, tracking touches and gestures, converting them into the correct data type for sending, and then transmitting them over bluetooth is all handled by the main thread, meaning each action is executed one at a time and nothing happens consecutively. iPhone and iPad processors are remarkably fast, but utilizing concurrency and multithreading will increase performance. Beyond that there are plenty more features that can be built in by harnessing the power and flexibility of IOKit and AppleScript. With further time and development the Magic Launchpad can evolve into a polished and fully featured application.

Overall the Magic Launchpad was a successful project and proved there is room for human interface devices to grow and evolve. The Magic Launchpad, in either software or hardware form, might not reach large scale popularity, but its warm reception is encouraging and certainly warrants further development.

References

1. Encyclopedia of Interaction Design: http://interaction-design.org/encyclopedia/interaction_design.html
2. Bill Verplank homepage: http://www.billverplank.com/professional.html
3. A History of the GUI: http://arstechnica.com/features/2005/05/gui/
4. MagicPrefs: http://magicprefs.com/
5. Better Touch Tool: http://www.boastr.net/
6. Apple iOS Developer Center: https://developer.apple.com/devcenter/ios/index.action
7. The Virtues of a Second Screen: http://www.nytimes.com/2006/04/20/technology/20basics.html?_r=3&
8. Two Screens Are Better Than One: http://research.microsoft.com/en-us/news/features/vibe.aspx
9. Apple Developer Center Core Bluetooth Library: https://developer.apple.com/library/ios/navigation/#section=Frameworks&topic=CoreBluetooth
10. A Look at the Basics of Bluetooth Wireless Technology: http://www.bluetooth.com/Pages/Basics.aspx
11. Apple Developer Center Quartz Display Services Reference: https://developer.apple.com/library/mac/documentation/GraphicsImaging/Reference/Quartz_Services_Ref/Reference/reference.html
12. Apple Developer Center Quartz Event Services Reference: https://developer.apple.com/library/mac/documentation/Carbon/Reference/QuartzEventServicesRef/Reference/reference.html
13. Apple Developer Center Introduction to I/O Kit Fundamentals: https://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html#//apple_ref/doc/uid/TP0000011
14. Apple Developer Center Technical Note TN2084: Using AppleScript in Cocoa Applications: https://developer.apple.com/library/mac/technotes/tn2084/_index.html
15. Dieter Rams: ten principles for good design: https://www.vitsoe.com/us/about/good-design
16. Apple Developer Center Event Handling Guide for iOS: Gesture Recognizers: https://developer.apple.com/library/IOS/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizer_basics/GestureRecognizer_basics.html#//apple_ref/doc/uid/TP40009541-CH2-SW14
17. Fitts's Law: http://www.interaction-design.org/encyclopedia/fitts_law.html
18. Razer SwitchBlade UI: http://www.razerzone.com/switchblade-ui

Appendix A - iOS Application Implementation File for Main View Controller

```
//
//  MainViewController.m
//  Magic Launchpad
//
//  Created by Tyler Durkin on 12/3/13.
//  Copyright (c) 2013 Tyler Durkin. All rights reserved.
//

#import "MainViewController.h"
#import <CoreBluetooth/CoreBluetooth.h>
#import "TransferService.h"

@interface MainViewController () <CBPeripheralManagerDelegate>

@property (strong, nonatomic) CBPeripheralManager       *peripheralManager;
@property (strong, nonatomic) CBMutableCharacteristic   *mousePointCharacteristic;
@property (strong, nonatomic) NSData                    *dataToSend;

@property (strong, nonatomic) IBOutlet UIPanGestureRecognizer *panRecognizer;

@end

@implementation MainViewController


#pragma mark - View Lifecycle


- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do additional setup after loading the view
    _peripheralManager = [[CBPeripheralManager alloc] initWithDelegate:self queue:nil options:nil];
    NSLog(@"Peripheral manager allocated");
}


- (void)viewWillDisappear:(BOOL)animated
{
    [self.peripheralManager stopAdvertising];
    [super viewWillDisappear:animated];
}


#pragma mark - Peripheral Methods


// returns error if unable to publish
- (void) peripheralManager:(CBPeripheralManager *)peripheral didAddService:(CBService *)service error:(NSError *)error {
    if (error) {
        NSLog(@"Error publishing service: %@", [error localizedDescription]);
    }
}


- (void) peripheralManagerDidUpdateState:(CBPeripheralManager *)peripheral {
    // Opt out from any other state
    if (peripheral.state == CBPeripheralManagerStatePoweredOn) {

        NSLog(@"self.peripheralManager powered on");

        // set up tree
        self.mousePointCharacteristic = [[CBMutableCharacteristic alloc] initWithType:[CBUUID UUIDWithString:MOUSE_POINT_UUID]
                                                                            properties:CBCharacteristicPropertyNotify
                                                                                 value:nil
                                                                           permissions:CBAttributePermissionsReadable];

        CBMutableService *launchpadMutableService = [[CBMutableService alloc] initWithType:[CBUUID UUIDWithString:LAUNCHPAD_SERVICE_UUID]
                                                                                    primary:YES];


        // Add characteristic to service
        launchpadMutableService.characteristics = @[self.mousePointCharacteristic];

        // publish service
        [self.peripheralManager addService:launchpadMutableService];
        NSLog(@"Service added");

        // begin advertising
        [self.peripheralManager startAdvertising:@{CBAdvertisementDataServiceUUIDsKey:@[launchpadMutableService.UUID],
            CBAdvertisementDataLocalNameKey:@"iOS Magic Launchpad"}];
        NSLog(@"Now advertising as %@", CBAdvertisementDataLocalNameKey);
    }
}


// Notify when a Central subscribes
- (void) peripheralManager:(CBPeripheralManager *)peripheral central:(CBCentral *)central didSubscribeToCharacteristic:(CBCharacteristic *)
    characteristic
{
    NSLog(@"Central subscribed to characteristic");
}


// recognize when central unsubscribes
- (void) peripheralManager:(CBPeripheralManager *)peripheral central:(CBCentral *)central didUnsubscribeFromCharacteristic:(CBCharacteristic *)
    characteristic
{
    NSLog(@"Central unsubscribed from characteristic");
}


```

```
#pragma mark - UIPanGesture Methods


- (IBAction)methodForPanRecognizer:(UIPanGestureRecognizer *)sender
{
    touchPoint = [sender locationInView:self.view];

    // gather first touch information
    if ([sender state] == UIGestureRecognizerStateBegan) {
        NSLog(@"First touch at: %@", NSStringFromCGPoint(touchPoint));
    }

    // recognize when it's a continuing touch
    else NSLog(@"Touch at: %@", NSStringFromCGPoint(touchPoint));

    NSString *stringPoint = NSStringFromCGPoint(touchPoint);

    // we have the new data string so let's send it
    BOOL didSend = [self.peripheralManager updateValue:[stringPoint dataUsingEncoding:NSASCIIStringEncoding] forCharacteristic:self.
        mousePointCharacteristic onSubscribedCentrals:nil];

    if (didSend) {
        NSLog(@"Touch sent");
        didSend = FALSE;
    }

}


#pragma mark - Default Methods


- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Appendix B - Mac Application App Delegate Implementation File

```objc
/*
 //  File: LaunchpadClientAppDelegate.m
 //  Version: 1.0
 //  Created by Tyler Durkin on 12/3/13.
 //  Copyright (c) 2013 Tyler Durkin. All rights reserved.
 */

#import "LaunchpadClientAppDelegate.h"
#import "TransferService.h"

@implementation LaunchpadClientAppDelegate

@synthesize statusMenu;
@synthesize window;
@synthesize deviceSheet;
@synthesize deviceName;
@synthesize connectStatus;
@synthesize mousePointCharacteristic;
@synthesize launchpadDevices;
@synthesize arrayController;


- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // start menulet
    [self activateStatusMenu];

    // setup launchpad device list and create CoreBluetooth central
    self.launchpadDevices = [NSMutableArray array];
    manager = [[CBCentralManager alloc] initWithDelegate:self queue:nil];
    NSLog(@"Bluetooth central manager allocated");

    // Gather info about the active screen
    [self gatherScreenInfo];
}


// create the status bar menulet
- (void)activateStatusMenu
{
    NSStatusBar *bar = [NSStatusBar systemStatusBar];
    statusItem = [bar statusItemWithLength:NSVariableStatusItemLength];

    [statusItem setMenu:statusMenu];
    [statusItem setImage:[NSImage imageNamed:@"menuIcon.png"]];
    [statusItem setHighlightMode:YES];
    NSLog(@"Status bar created and app launched");
}


- (void)gatherScreenInfo
{
    // get active screen
    NSScreen *mainScreen = [NSScreen mainScreen];
    NSRect screenRect = [mainScreen visibleFrame];

    // determine screen size
    NSSize screenSize = screenRect.size;

    screenWidth = screenSize.width;
    screenHeight = screenSize.height;
    NSLog(@"Screen resolution is: %f by %f", screenWidth, screenHeight);
}


- (void) dealloc
{
    [self stopScan];
    [launchpadPeripheral setDelegate:nil];
}


// Disconnect peripheral when application terminate

- (void) applicationWillTerminate:(NSNotification *)notification
{
    if(launchpadPeripheral) [manager cancelPeripheralConnection:launchpadPeripheral];
}


#pragma mark - Scan sheet methods


// Open scan sheet to discover launchpad peripherals if it is LE capable hardware
- (IBAction)openScanSheet:(id)sender
{
    if( [self isLECapableHardware] )
    {
        NSLog(@"Hardware is OK");
        autoConnect = FALSE;
        [arrayController removeObjects:launchpadDevices];
        [NSApp beginSheet:self.deviceSheet modalForWindow:[self window] modalDelegate:self
           didEndSelector:@selector(sheetDidEnd:returnCode:contextInfo:)
              contextInfo:nil];
        [self startScan];
    }
}

```

```objc

// Close scan sheet once device is selected
- (IBAction)closeScanSheet:(id)sender
{
    [NSApp endSheet:self.deviceSheet returnCode:NSAlertDefaultReturn];
    [self.deviceSheet orderOut:self];
}


// Close scan sheet without choosing any device
- (IBAction)cancelScanSheet:(id)sender
{
    [NSApp endSheet:self.deviceSheet returnCode:NSAlertAlternateReturn];
    [self.deviceSheet orderOut:self];
}


// This method is called when Scan sheet is closed. Initiate connection to selected launchpad peripheral
- (void)sheetDidEnd:(NSWindow *)sheet returnCode:(NSInteger)returnCode contextInfo:(void *)contextInfo
{
    [self stopScan];
    if(returnCode == NSAlertDefaultReturn)
    {
        NSIndexSet *indexes = [self.arrayController selectionIndexes];
        if ([indexes count] != 0)
        {
            NSUInteger anIndex = [indexes firstIndex];
            launchpadPeripheral = [self.launchpadDevices objectAtIndex:anIndex];
            [manager connectPeripheral:launchpadPeripheral options:nil];
        }
    }
}


#pragma mark - Connect Button


// This method is called when connect button pressed and it takes appropriate actions depending on device connection state
- (IBAction)connectButtonPressed:(id)sender
{
    if(launchpadPeripheral && ([launchpadPeripheral isConnected]))
    {
        /* Disconnect peripheral if its already connected */
        [manager cancelPeripheralConnection:launchpadPeripheral];
    }
    else if (launchpadPeripheral)
    {
        /* Device is not connected, cancel pending connection */
        [manager cancelPeripheralConnection:launchpadPeripheral];
        [self openScanSheet:nil];
    }
    else
    {
        /* No outstanding connection, open scan sheet */
        [self openScanSheet:nil];
    }
}


#pragma mark - Start/Stop Scan methods


// Request CBCentralManager to scan for launchpad peripherals
- (void)startScan
{
    NSLog(@"Initiating scanning");
    [manager scanForPeripheralsWithServices:nil options:nil];
    NSLog(@"Now scanning...");
}


// Request CBCentralManager to stop scanning for launchpad peripherals
- (void)stopScan
{
    [manager stopScan];
}


#pragma mark - Start/Stop notification/indication


// Start or stop receiving notification or indication on interested characteristics
- (IBAction)startButtonPressed:(id)sender
{
    if(self.mousePointCharacteristic)
    {
        // Set indication on mouse characteristic
        [launchpadPeripheral setNotifyValue:YES forCharacteristic:self.mousePointCharacteristic];
    }
}


#pragma mark - LE Capable Platform/Hardware check


```

```objc
// Uses CBCentralManager to check whether the current platform/hardware supports Bluetooth LE. An alert is raised if Bluetooth LE is not enabled or is not supported.
- (BOOL) isLECapableHardware
{
    NSString * state = nil;

    switch ([manager state])
    {
        case CBCentralManagerStateUnsupported:
            state = @"The platform/hardware doesn't support Bluetooth Low Energy.";
            break;
        case CBCentralManagerStateUnauthorized:
            state = @"The app is not authorized to use Bluetooth Low Energy.";
            break;
        case CBCentralManagerStatePoweredOff:
            state = @"Bluetooth is currently powered off.";
            break;
        case CBCentralManagerStatePoweredOn:
            return TRUE;
        case CBCentralManagerStateUnknown:
        default:
            return FALSE;

    }

    NSLog(@"Central manager state: %@", state);

    [self cancelScanSheet:nil];

    NSAlert *alert = [[NSAlert alloc] init];
    [alert setMessageText:state];
    [alert addButtonWithTitle:@"OK"];
    [alert beginSheetModalForWindow:[self window] modalDelegate:self didEndSelector:nil contextInfo:nil];
    return FALSE;
}


#pragma mark - CBManagerDelegate methods


// Invoked whenever the central manager's state is updated.
- (void)centralManagerDidUpdateState:(CBCentralManager *)central
{
    [self isLECapableHardware];
}


// Invoked when the central discovers launchpad peripheral while scanning.
- (void)centralManager:(CBCentralManager *)central didDiscoverPeripheral:(CBPeripheral *)peripheral advertisementData:(NSDictionary *)advertisementData RSSI:(NSNumber *)RSSI
{
    NSLog(@"Did discover peripheral. peripheral: %@ rssi: %@, UUID: %@ advertisementData: %@ ", peripheral, RSSI, peripheral.UUID, advertisementData);

    NSMutableArray *peripherals = [self mutableArrayValueForKey:@"launchpadDevices"];
    if( ![self.launchpadDevices containsObject:peripheral] )
        [peripherals addObject:peripheral];

    /* Retreive already known devices */
    if(autoConnect)
    {
        [manager retrievePeripherals:[NSArray arrayWithObject:(id)peripheral.UUID]];
    }
}


// Invoked when the central manager retrieves the list of known peripherals.
// Automatically connect to first known peripheral
- (void)centralManager:(CBCentralManager *)central didRetrievePeripherals:(NSArray *)peripherals
{
    NSLog(@"Retrieved peripheral: %lu - %@", [peripherals count], peripherals);

    [self stopScan];

    /* If there are any known devices, automatically connect to it.*/
    if([peripherals count] >=1)
    {
        launchpadPeripheral = [peripherals objectAtIndex:0];
        [manager connectPeripheral:launchpadPeripheral options:[NSDictionary dictionaryWithObject:[NSNumber numberWithBool:YES] forKey:CBConnectPeripheralOptionNotifyOnDisconnectionKey]];
    }
}


// Invoked whenever a connection is succesfully created with the peripheral.
// Discover available services on the peripheral
- (void)centralManager:(CBCentralManager *)central didConnectPeripheral:(CBPeripheral *)peripheral
{
    NSLog(@"Did connect to peripheral: %@", peripheral);

    self.connectStatus = @"Connected";
    [peripheral setDelegate:self];
    [peripheral discoverServices:nil];
}
```

```objc
// Invoked whenever an existing connection with the peripheral is torn down.
// Reset local variables
- (void)centralManager:(CBCentralManager *)central didDisconnectPeripheral:(CBPeripheral *)peripheral error:(NSError *)error
{
    NSLog(@"Did Disconnect to peripheral: %@ with error = %@", peripheral, [error localizedDescription]);
    self.connectStatus = @"Not Connected";
    self.deviceName = @"";
    if( launchpadPeripheral )
    {
        [launchpadPeripheral setDelegate:nil];
        launchpadPeripheral = nil;
    }
}


// Invoked whenever the central manager fails to create a connection with the peripheral.
- (void)centralManager:(CBCentralManager *)central didFailToConnectPeripheral:(CBPeripheral *)peripheral error:(NSError *)error
{
    NSLog(@"Fail to connect to peripheral: %@ with error = %@", peripheral, [error localizedDescription]);
    if( launchpadPeripheral )
    {
        [launchpadPeripheral setDelegate:nil];
        launchpadPeripheral = nil;
    }
}


#pragma mark - CBPeripheralDelegate methods


// Invoked upon completion of a -[discoverServices:] request.
- (void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:(NSError *)error
{
    if (error)
    {
        NSLog(@"Discovered services for %@ with error: %@", peripheral.name, [error localizedDescription]);
        return;
    }
    for (CBService * service in peripheral.services)
    {
        NSLog(@"Service found with UUID: %@", service.UUID);

        if([service.UUID isEqual:[CBUUID UUIDWithString:LAUNCHPAD_SERVICE_UUID]])
        {
            /* Launchpad Service - discover mouse characteristics */
            [launchpadPeripheral discoverCharacteristics:[NSArray arrayWithObjects:[CBUUID UUIDWithString:MOUSE_POINT_UUID], nil] forService:
                service];
        }
        else if ( [service.UUID isEqual:[CBUUID UUIDWithString:CBUUIDGenericAccessProfileString]] )
        {
            /* GAP (Generic Access Profile) - discover device name characteristic */
            [launchpadPeripheral discoverCharacteristics:[NSArray arrayWithObject:[CBUUID UUIDWithString:CBUUIDDeviceNameString]]  forService:
                service];
        }
    }
}


// Invoked upon completion of a -[discoverCharacteristics:forService:] request.
- (void)peripheral:(CBPeripheral *)peripheral didDiscoverCharacteristicsForService:(CBService *)service error:(NSError *)error
{
    if (error)
    {
        NSLog(@"Discovered characteristics for %@ with error: %@", service.UUID, [error localizedDescription]);
        return;
    }

    if([service.UUID isEqual:[CBUUID UUIDWithString:LAUNCHPAD_SERVICE_UUID]])
    {
        for (CBCharacteristic * characteristic in service.characteristics)
        {
            /* Set indication on mouse movement */
            if([characteristic.UUID isEqual:[CBUUID UUIDWithString:MOUSE_POINT_UUID]])
            {
                self.mousePointCharacteristic = characteristic;
                NSLog(@"Found a mouse characteristic");
                [peripheral setNotifyValue:YES forCharacteristic:characteristic];
                NSLog(@"Subscribed to mouse characteristic");
            }
        }
    }

    if ( [service.UUID isEqual:[CBUUID UUIDWithString:CBUUIDGenericAccessProfileString]] )
    {
        for (CBCharacteristic *characteristic in service.characteristics)
        {
            /* Read device name */
            if([characteristic.UUID isEqual:[CBUUID UUIDWithString:CBUUIDDeviceNameString]])
            {
                [launchpadPeripheral readValueForCharacteristic:characteristic];
                NSLog(@"Found a Device Name Characteristic - Read device name");
            }
        }
    }
}

```

```objc
// Invoked upon completion of a -[readValueForCharacteristic:] request or on the reception of a notification/indication.
- (void)peripheral:(CBPeripheral *)peripheral didUpdateValueForCharacteristic:(CBCharacteristic *)characteristic error:(NSError *)error
{
    if (error)
    {
        NSLog(@"Error updating value for characteristic %@ error: %@", characteristic.UUID, [error localizedDescription]);
        return;
    }

    // Updated value for mouse point received
    if([characteristic.UUID isEqual:[CBUUID UUIDWithString:MOUSE_POINT_UUID]])
    {
        NSString *stringFromData = [[NSString alloc] initWithData:characteristic.value encoding:NSASCIIStringEncoding];
        NSPoint convertedPoint = NSPointFromString(stringFromData);
        CGPoint newPoint = NSPointToCGPoint(convertedPoint);

        NSLog(@"Received CGPoint: %@", NSStringFromPoint(newPoint));

        // send received point data to warp method
        [self warpMouseCursorPosition:&newPoint];
    }

}


// Move position of mouse cursor to properly scaled point on screen
- (void)warpMouseCursorPosition:(CGPoint *)warpPoint
{
    NSLog(@"Warping mouse position to: %@", NSStringFromPoint(*warpPoint));

    // Determine necessary scaling
    float scaleX = screenWidth/564.5; // currently only using iPhone 5 screen, will need variable for release
    float scaleY = (screenHeight + 26)/310;

    CGFloat scaledX = (warpPoint->x - 1.5)* scaleX;
    NSLog(@"Scaling X by: %f", scaleX);
    CGFloat scaledY = warpPoint->y * scaleY;
    NSLog(@"Scaling Y by: %f", scaleY);

    CGPoint newWarpingPoint = CGPointMake(scaledX, scaledY);

    CGWarpMouseCursorPosition(newWarpingPoint);

    NSLog(@"Mouse warped");
}


// Invoked upon completion of a -[writeValue:forCharacteristic:] request.
- (void)peripheral:(CBPeripheral *)peripheral didWriteValueForCharacteristic:(CBCharacteristic *)characteristic error:(NSError *)error
{
    if (error)
    {
        NSLog(@"Error writing value for characteristic %@ error: %@", characteristic.UUID, [error localizedDescription]);
        return;
    }
}


// Invoked upon completion of a -[setNotifyValue:forCharacteristic:] request.
- (void)peripheral:(CBPeripheral *)peripheral didUpdateNotificationStateForCharacteristic:(CBCharacteristic *)characteristic error:(NSError *)
    error
{
    if (error)
    {
        NSLog(@"Error updating notification state for characteristic %@ error: %@", characteristic.UUID, [error localizedDescription]);
        return;
    }

    NSLog(@"Updated notification state for characteristic %@ (newState:%@)", characteristic.UUID, [characteristic isNotifying] ? @"Notifying" :
        @"Not Notifying");
}

@end
```