2014

A Toolkit for Tracking and Mediating Parametric Objects upon Commodity Mobile Devices

Shantanu Thatte
Louisiana State University and Agricultural and Mechanical College, shantanuthatte@gmail.com

A Toolkit for Tracking and Mediating Parametric Objects upon Commodity Mobile Devices

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Computer Science

by
Shantanu Thatte
B.E., University of Pune, 2012
December 2014

Acknowledgments

I would like to express my deepest gratitude towards Dr. Brygg Ullmer and Dr. Chris Branton for being my advisors, mentors and teachers. Right from when I joined the Tangible Visualization *"TangViz"* group, Dr. Ullmer and Dr. Branton have shown me immense patience and always guided me throughout the research. They have always supported and critiqued me and thus helping me grow.

I would also like to thank Louisiana Biomedical Research Foundation (LBRN) and its support from National Institute of General Medical Sciences (NIGMS) Grant 8P20GM103424 and the Louisiana Board of Regents Support Fund. This research work has also been supported by NSF award number CNS-1126739.

My sincere thanks go to the members of my committee, Dr. Gerald Baumgartner and Dr. Jesse Allison, for their continuous support. Also, I appreciate the assistance provided by staff members of Division of Computer Science and Engineering.

I especially appreciate help and support from my colleagues from "TangViz" lab. Hali for her critique on my designs; Narendra for his assistance with electronics; Ben for bouncing ideas, looking at my code and help in shaping the architecture; Andre and Alan for helping me out with the Sifteos; and everyone else for their timely support.

I also would like to thank the staff at Center for Computation and Technology and IT Support for their continuous assistance.

Last but not the least, I would like to thank my family for their continuous support and guidance throughout my journey. Finally my friends, without their support my stay at LSU would not have been enjoyable and memorable.

Table of Contents

List of Tables

List of Figures

Abstract

The large number of mobile devices introduced in the market in recent years provide numerous interaction opportunities. We used commercially available products to create an interface for interaction with diverse datasets. We extended this scenario to include multiple physical instances, realizing a real-time multi-party interaction session. Tangibles act as tools and containers, in a fashion that requires their physical location and orientation to be evaluated continuously during a session. We present a technique to detect an object using only the touchscreen of a mobile device as an input, by means of touch patterns created on the screen. This helps avoid additional embedded electronics and allows an object to be used as a tangible. We also conducted various other experiments using sensors available on mobile devices to detect objects. We also discuss the architecture and implementation of a low latency backend to relay and propagate events to all participating devices. This includes a low latency memory based key-value store implemented using Redis, and persistent storage using MongoDB. The server side scripts are implemented using the fast non-blocking Node.js Javascript engine to minimize impact on latency imparted during data processing.

Chapter 1
Introduction

1.1 The Current Mobile Ecosystem

There are presently a multitude of mobile devices available for consumers, ranging from smartphones and tablets to specialized controllers. In today's scenario the mobile device market is dominated by smartphones and tablets, which sport a two dimensional screen as the primary user input source. Physical tactile buttons are also a disappearing feature on these devices. The mobile ecosystem is also expanding to incorporate other forms of inputs in form of voice-commands and hand gestures, but these are still in their not fully developed for extensive use and provide limited control over the functionality of applications. The mobile family is also expanding into the realm of wearable computing in the form of smart watches and digital eye-wear. Recent examples of commercial products include the likes of *Motorola 360 watch* and *Google Glass* respectively. Although innovative, these primarily act as extensions of the smartphones or tablets to which they are connected. Even though they offer a new way to interact with the devices, they still require user attention for most of their functionality. For complex actions the user needs to largely detach himself from other interactive devices to work on them.

A major advantage of these mobile devices is consumer uptake. According to the global mobile survey conducted by Deloitte for the year 2013 [**?**], on average a consumer in United States owns 5.2 portable devices, an 18% growth over the last year alone. Specifically, the tablet penetration stands at 31% for the United States [**?**]. This presents a platform already used by large number of potential users. The device themselves have evolved over time and provide better on-board processing power and connectivity over their predecessors.

1.2 Solution

We employ techniques using on-board sensors to identify and track physical objects placed on the screen of tablet devices. These objects or *"tangibles"* act as physically manipulable containers and controls. They also provide haptic feedback to the user, allowing a more natural interface to the underlying application. This expands the interaction scope of these devices, allowing for *graspable* interactive elements. To select data points between a set of values, it is straightforward for users to adjust dials to define the range, as an alternative to GUI controls such as a textbox with range controls.[1] However, this has been developed further by applications targeting mobile devices by creating *"touch-friendly"* controls that have movable elements representing the current value. Considering more complex queries for selecting a class or category, and applying a function on the result might seem suited for a query language (like SQL), but selecting a rotary control to point towards the intended class and selecting the function seems natural [**?**]. *Tangibles* allow interface designers and users to use physical controls on the traditional two dimensional digital surfaces.

Apart from simplifying query creation, such interfaces open the possibility of a real-time interaction with big data, where changing parameters immediately reflect the change on output. Coupling this feature with good network connectivity can lead to multi-party interaction systems. As datasets grow and shift towards distributed computing/storage, efficient simultaneous collaboration on these data sets seem complicated. By maintaining a global interaction state and dedicated communication channels, a interaction system can facilitate many research workflows by reducing the time to create an optimized query and collaborate on same problems.

[1] A *range selector* traditionally has a textbox showing the current numerical value with "up" and "down" buttons next to it.

Chapter 2
Related Work

Mobile devices have brought a shift in various interaction scenarios. Mark Weiser's vision of a ubiquitous computing is as close to reality as it has ever been [**?**]. There are numerous devices available to users that are bridging the gap between desktop and mobile computing. Most of these devices share the same architecture and design, but vary in form factor. The shift from mainstream computation devices (such as workstations, desktops and laptops) towards cloud connected, thin-client mobile devices (such as Google's Chromebooks and Microsoft Surface Pro^{TM}) present opportunities in new interface designs. Along with this many modern net-books and laptops also come with a touchscreen; additionally some of these devices morph from a traditional laptop to a tablet-like device.

There has been tremendous progress in HPC[1] in last decade, the product of which are suites of programs able to crunch huge amounts of data. There are numerous clusters, each with multiple nodes to allow large computations on complex models. What lacks within these suites are better interfaces, as such most of these tools sport a CUI[2]. This sometime presents a hurdle for researchers new to the software and its instructions. The small sub-class of these tools that do offer GUI[3] in the form of WIMP[4] interfaces suffer from lack of natural user interface[**?**].

Tangible interfaces have been spearheaded in recent years by many researchers and their groups. Currently, there is a great momentum towards enabling computational ergonomics for the masses, primarily due to the drop in price per computation. Combining *tangibles*

[1]High Performance Computing

[2]Character User Interface

[3]Graphical User Interface

[4]Windows-Icons-Menus-Pointers

(tangible interfaces) with mobile devices enables the physical world to be associated with the digital counterpart portrayed within the mobile devices. Tangibles have been used in conjunction with other systems to support streamlined workflows, like the G-Nome Surfer[**?**]. Tangible systems can leverage the knowledge of a subject-specific researcher by providing intuitive interaction tools to be used within their research workflows without much disruption.

Of the numerous applications on *tangibles*, a unique application was described by Ullmer, et.al, where they defined a new interaction element, a *"cartouche"*, to provide spacial constraints for *tangibles*. *Cartouches* allow for easy insertion of physical controls within the interaction scenario of mobile devices, where screen space is limited. Extending upon this work, *cartouches* can be easily created by end users, and customized based upon the domain application. The low-cost and ease of mass production of these elements, make it an ideal candidate for our application. Uniting physical tokens within constraints allows for efficient haptic feedback and encourages natural gestures based in daily used controls like the dials, sliders, knobs, etc.

Chapter 3
Hardware Available on Mobile Devices

Modern smartphones, tablets and portable computers provide plurality of sensors for measuring acceleration, rotation, proximity, magnetic field, air-pressure, ambient light and capturing images. These are illustrated in Figure 3.1 which shows the various sensors. These devices also now support substantial computational resources. High-end devices now come equipped with multi-core processors running in GHz range (fastest mobile processor as of mid-2014 is the quad-core Qualcomm Snapdragon 805 at 2.7 GHz), open the possibilities of a multi-threaded applications, which were previously restricted to at least the mainstream desktop computers. The plethora of applications available for users seldom use multiple sensors for interaction and can be re-tasked to support additional gestures. These sensors are highly sensitive and when data from multiple sensors are conglomerated and complex gestures involving various sensors is possible.

Communication capabilities of mobile devices has also increased over past few years. Low-end smartphones have basic *Wi-Fi* connectivity with speeds reaching 100Mbits/second, where recent high-end devices support the latest wireless protocols like 802.11ac (supporting around 500 Mbits/second[1]), along with support for 4G networks, Bluetooth (version 4.0 and low-power Bluetooth) and Near-Field Communication (NFC). These open new viable communication channels to connect to physical objects.

The touchscreens available on the devices has also evolved to support multiple simultaneous touches along with the latest proximity touch detection (as portrayed in Samsung Galaxy S5). These components are used primarily for enabling interaction with the graphical inter-

[1]Specifications state a throughput of 433 Mbits/s for a single link at 80 MHz bandwidth and up to 3.39 Gbit/s for multi-link stations with 4x4 MIMO.

face presented by the application, by directing user's actions to the underlying application. However, they lack the effectiveness and feedback of a tactile button[**?**].

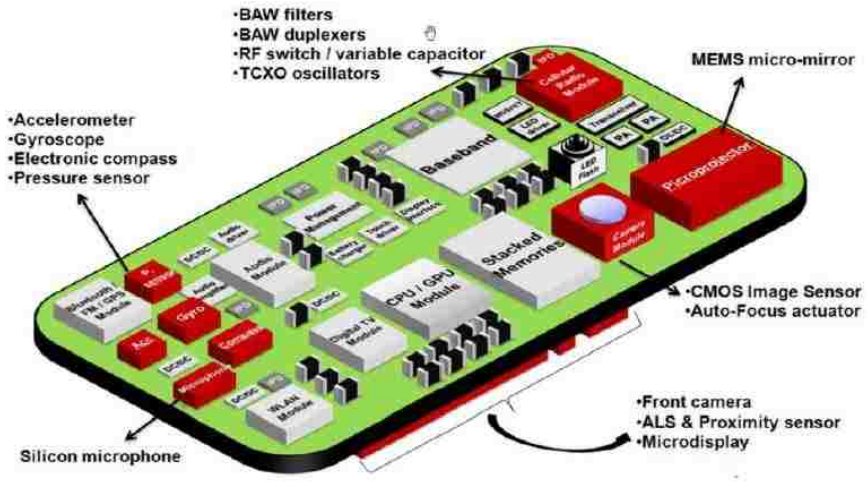

Figure 3.1: Mobile Phone Sensors

Chapter 4
Re-targeting On-board Sensors

4.1 Touchscreen

The touchscreen on mobile devices can track multiple finger touches, up to 10 instances in most devices. Most of the applications present an interface that seldom utilize this capability of the touchscreen. A vast majority of the applications use maximum of two touches within their gestures.

There are various applications involving object detection on a screen, where majority of these use optical methods to detect. One of these is the *Microsoft SUR40* which uses a matrix of IR sensors to detect objects placed on the screen. Since it uses IR, the matrix produces a (low -resolution) gray-scale image. By placing *ByteTags*, which are a pattern of white circles on a black background as shown in Figure 4.1. Each ByteTag is uniquely identified by a 8-bit number, which depends upon the presence or absence of a white circle.

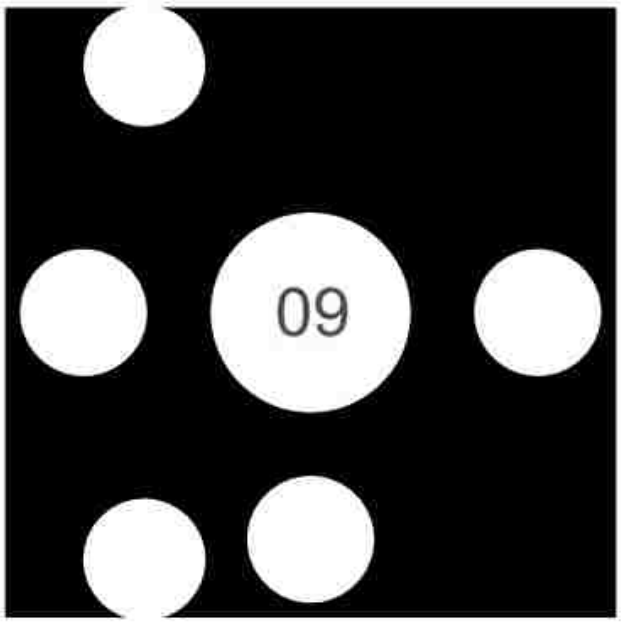

Figure 4.1: *ByteTag* used on a SUR40.

Another method utilized by some manufacturers to provide joystick functionality on a touch-screen involves using a hemi-spherical conductive surface with the joystick on the other end. Upon placing this on a capacitive screen the conductive surface creates a touch point on the screen. When the user tilts the joystick, it causes the surface to roll and causing the touch point to move.

In our application, we exploit this potential by simulating on-screen touch by using conductive elements embedded within our *tangibles*. From the perspective of the touchscreen, it appears just as a normal finger touch, our software on receiving this data identifies patterns that these simulated touches create. The pattern created are unique for each object and are used to identify as well as track them on the screen. This *pseudo-touch* acts as a secondary communication channel between *tangibles* and device.

To create the pads that make contact with the screen, we required the material used to be electrically conductive and soft so as not to create any scratch marks on the glass screen. Using any metal would create scratches on the glass surface. Conductive rubber sheet, which is primarily used in electro-magnetic shielding was used to cut out circular pads to be placed on the objects. These pads also bear a certain degree of elasticity which helped in creating a uniform contact on the screen surface.

4.2 Front-facing Camera

Detecting objects using a camera is one of the easiest method and there are numerous image recognition libraries available which can track the position and orientation of objects. New 3D mapping hardware such as the *Microsoft Kinect* and *Leap Motion Controller* allow for spacial object detection, but are available for external uses and are not embedded within the device. Secondly, they require special software requirements on client side to interface.

We also explored the possibility to use the front facing camera on most of the devices to detect objects placed on the screen. An approach tried was by trying to use fresnel lenses on top of the camera sensor to allow it focus on close objects. Due to the small aperture of the camera elements, a fresnel lens with sufficiently small concentric section is not feasible.

Another method attempted was based on color detection, by using light pipes and optical fibers, to channel certain areas of the screen which would be obscured by the objects placed on them to the camera. The software could then detect the object placed by calculating the color viewed by the camera. An early prototype is shown in Figure 4.2 where., placing a cartouche over a red square would result in the red light being *"piped"* to the camera via the optical fibers. This approach allows a fine granularity in the object detection by changing the color that is collected, but the additional hardware required to channel light from multiple areas exceeded a physical space constraint to be practically applicable. Even by using the thinner fibers, the field-of-view of the camera also limited the scope of this approach.

We also briefly examined the idea of using prisms on top of the front facing camera to *"see"* the space above the screen or by using a second device (such as mobile phone in conjunction with a tablet) to record the objects. This approach caused any objects behind another to be obscured and not being detected. The higher the camera from the screen would result in better detection.

4.3 Magnetometer and Digital Compass

Most of the mid-range devices offer a digital compass as a part of the sensor assemblage and many high-end devices also present a separate three-axis magnetometer, capable of reading magnetic field strengths. These sensors are extremely sensitive and have a wide range of measurement. In our tests, keeping a strong neodymium magnet directly above the sensor could only result in reaching the upper limit of the sensor on a Nexus 10 tablet. These sensors are generally coupled with the gyroscope to measure device's orientation and rotation.

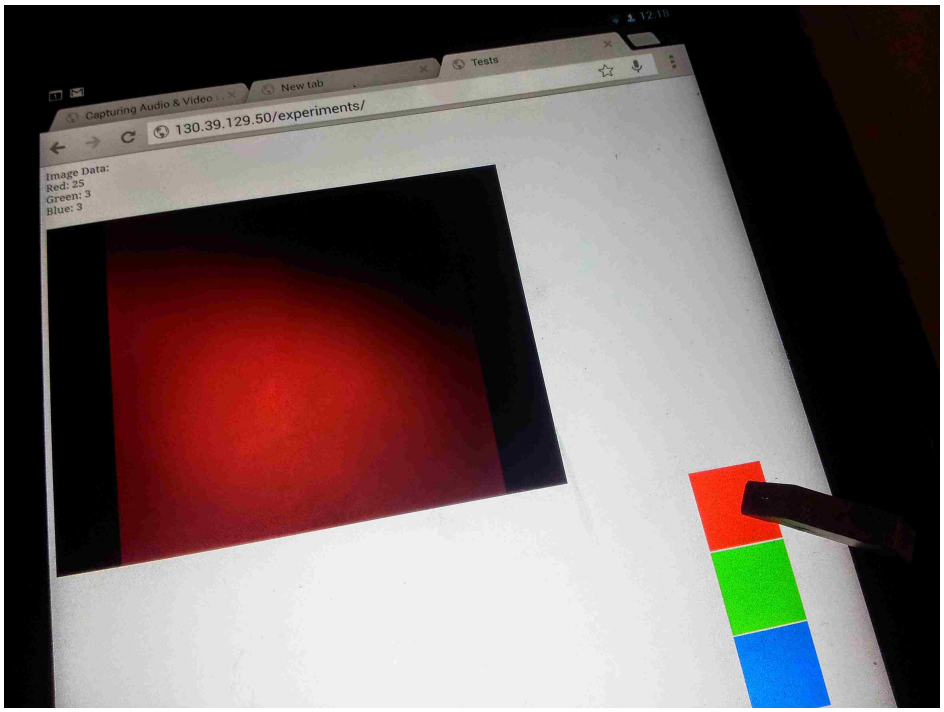

Figure 4.2: Color detection using front camera

We examined this possibility by creating *cartouches* with small neodymium magnets within them to create a unique magnetic signature as displayed in Figure 4.3. Each device on being placed would induce a different magnetic field across the sensor and could be detected. By monitoring the field changes, we could also deduce to a certain extend the movement of the object. We chose four different positions along the edge if the screen, to occupy the space used by bezel of the screen. A sample of various configurations are presented in Table 4.1. Each position could result in three possibilities: North pole facing down; South pole facing down and; magnet is absent. Upon trying various configurations and recording the magnetic field strengths, the result was inconclusive. The positions of the magnets were chosen such that when all magnets are placed in same orientation, the resultant magnetic field strength would be just shy of the sensor's maximum. Upon changing the orientation of one of the magnets, so as to create a sequence of *North-South-North-South* would result in least magnetic field strength. In this configuration the effect of the farthest magnet on the

sensor was least and could not be reliably detected. A possible solution for this was to limit the number magnets and arrange them equidistant from the sensor. This layout would have the magnets over the screen which would interfere with the on-screen UI.

Table 4.1: Sample positions of magnets

Pos. 1	Pos. 2	Pos. 3	Pos. 4
N	N	N	N
N	N	N	S
N	N	N	NULL
N	N	S	N
N	N	S	S
N	N	S	NULL
N	N	NULL	N
N	N	NULL	S
N	N	NULL	NULL
N	S	N	N
N	S	N	S
N	S	N	NULL

One of the factors affecting its use is the inherent magnetic influence of the earth's magnetic field. Changes in orientation of the device would result in significant variations in the induced magnetic field. We also attempted to overcome this by placing stronger magnets but resulted in the sensor being maxed sometimes, also persistent exposure to strong magnetic field is not recommended for such electronic devices. The effect of earth's magnetic field could be nulled out by pattern detection algorithms, but the computational requirements of such algorithms was beyond the scope of the on-board computational resources of the device. Remote computation is an option, but the delay introduced by the overhead of communication would be large enough to hamper smooth interaction.

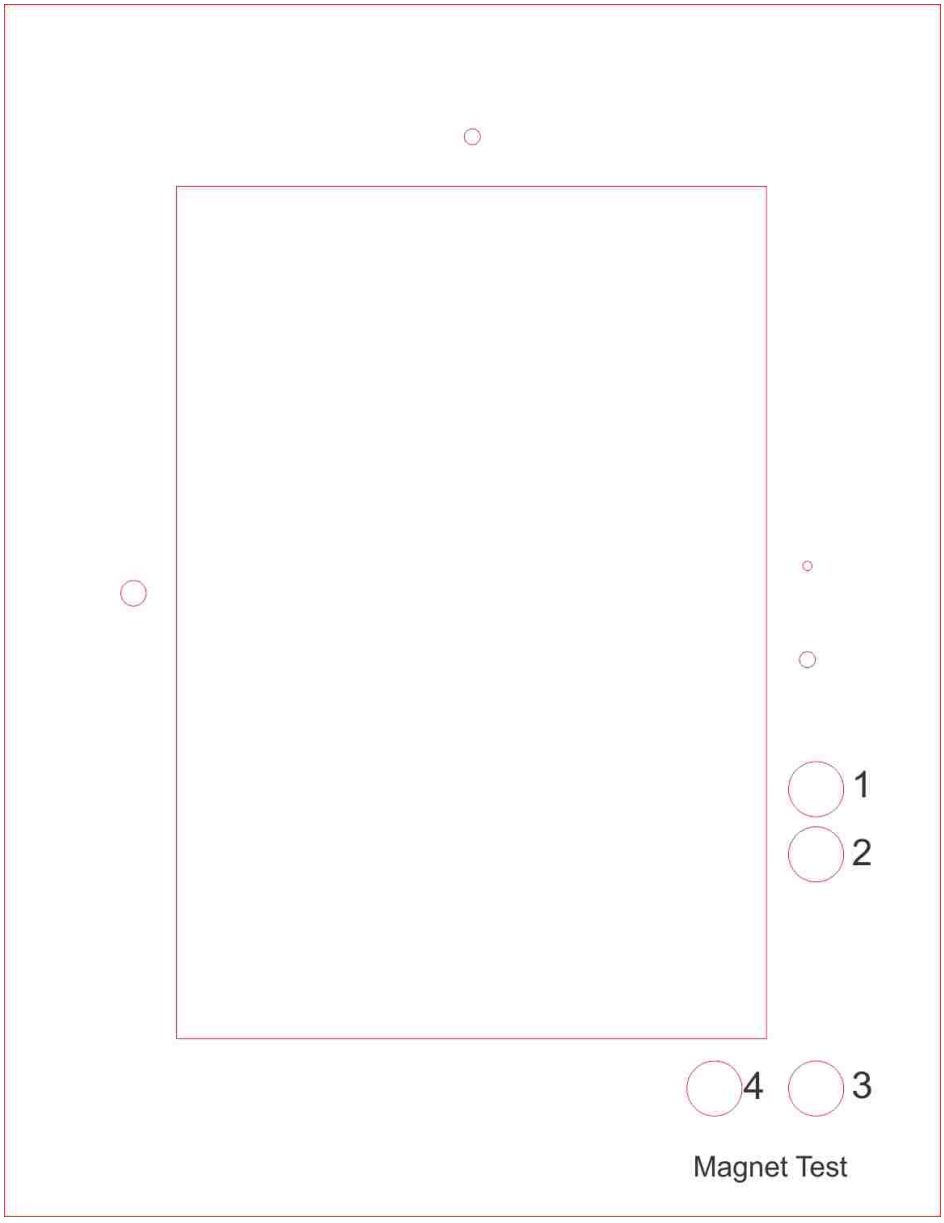

Figure 4.3: Test cartouche with various magnet positions

Chapter 5
Device-side Implementation

Since most the devices allow a seamless network connectivity and diversity of the platforms provided, we implemented the application based on HTML5 and the latest mobile browsers. Our requirements for the device side software was it to be easily deployable and provide maximum compatibility with the large platform differences in the mobile device market. In 2012 W3C designated HTML5 as the candidate recommendation for the World Wide Web standard [**?**]. It contains various new features added after the previous version viz. HTML4 such as native support for audio, video and the canvas elements. Among the set of new features, support for WebSockets was highly beneficial for us. WebSockets is a full-duplex communication protocol standardized by IETF. This was also one of the features that was incorporated early by many browsers and their mobile variants.

The device side implementation is in essence a HTML5 application running on the device and communicating with the server using WebSockets. The application is divided into two parts: Object detection section and display feedback section. All device side code has been implemented using Javascript and the jQuery Javascript library [**?**]. Each module has its own separate WebSocket object, that allows them to communicate independently. We also used the Socket.IO library so as to provide a more manageable WebSocket implementation [**?**].

In designing the client side interface we chose to continue with the idea of *cartouches* as a constraint based interaction system, originally proposed by Ullmer, et.al. [**?**]. The interface has cutouts for displaying a section of the screen and separate cutouts to act as wells for the object to be placed in. Figure 5.1 shows the design of the cartouche we used. Our target was to enable detection and communication with Sifteo Cubes which are great for tangible interface applications.

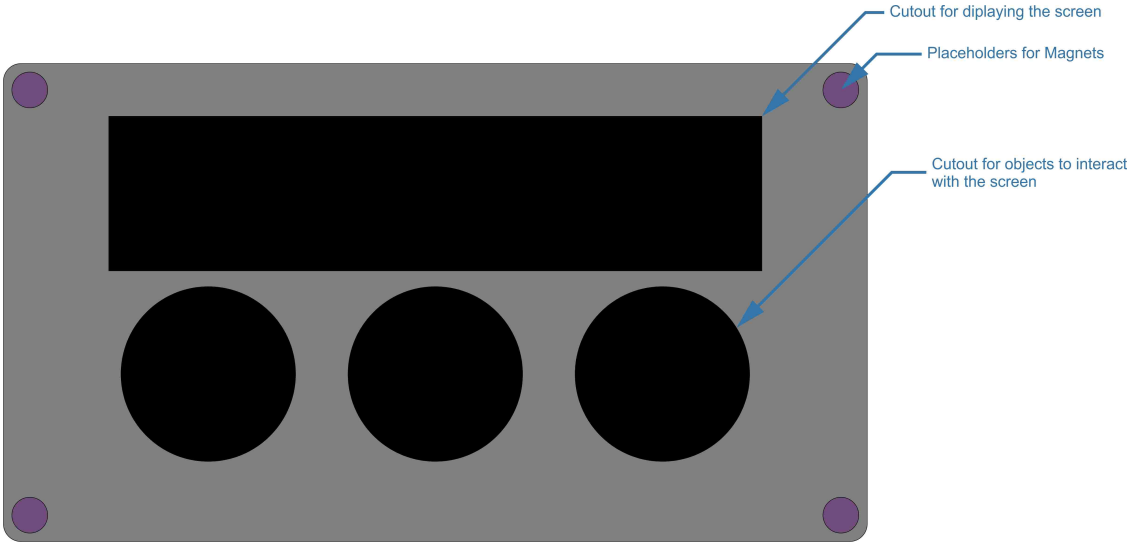

Figure 5.1: Cartouche used for the interaction scenario

As illustrated in Figure 5.2, each of these battery-powered cubes have a small $128px \times 128px$ screen capable of detecting only a touch and not the location of the touch. These cubes also have accelerometer to measure tilt and wireless connectivity with the Sifteo base unit. An unique feature of these cubes is that they have coils on the four sides. These coils get magnetically linked to another coil from a neighboring cube, thus allowing the system to detect which side of the cube is next to another cube and its side. The Sifteo base is a standalone battery-powered unit, on which the actual program is executed. It then streams the changes on the screen for each cube and receives back the data from the cube's sensors. To load programs, it connects to a PC/Mac via the USB port. The manufacturer, Sifteo Inc. provides a SDK[1] to allow custom applications to be built. To achieve communication during runtime a developmental version of the SDK was used, which allowed bi-directional USB communication.

Figure 5.2: A Sifteo Cube © Sifteo Inc.

5.1 Object Detection using Touchscreen

We used conductive rubber pads underneath the objects to create virtual touches on the touchscreen. Drawing inspiration from the ByteTags used on Microsoft's SUR40 and our application goal of using the Sifteo cubes on the screen by rotating them, we selected a circular form factor. We aimed to design a electronically-passive object, within which a Sifteo cube could reside. We call this as a *"receptacle"*.

Figure 5.3 illustrates the possible pas positions within the circular design of the receptacle. The square inscribed within denotes the position of the Sifteo cube. In our initial prototype we explored the possibilities of patterns within a single quadrant of the receptacle. This allowed us to create separate patterns for 38 individual receptacles to be detected. These are described in Table 5.1. The patterns created have three attributes which are used to

[1]Software Development Kit

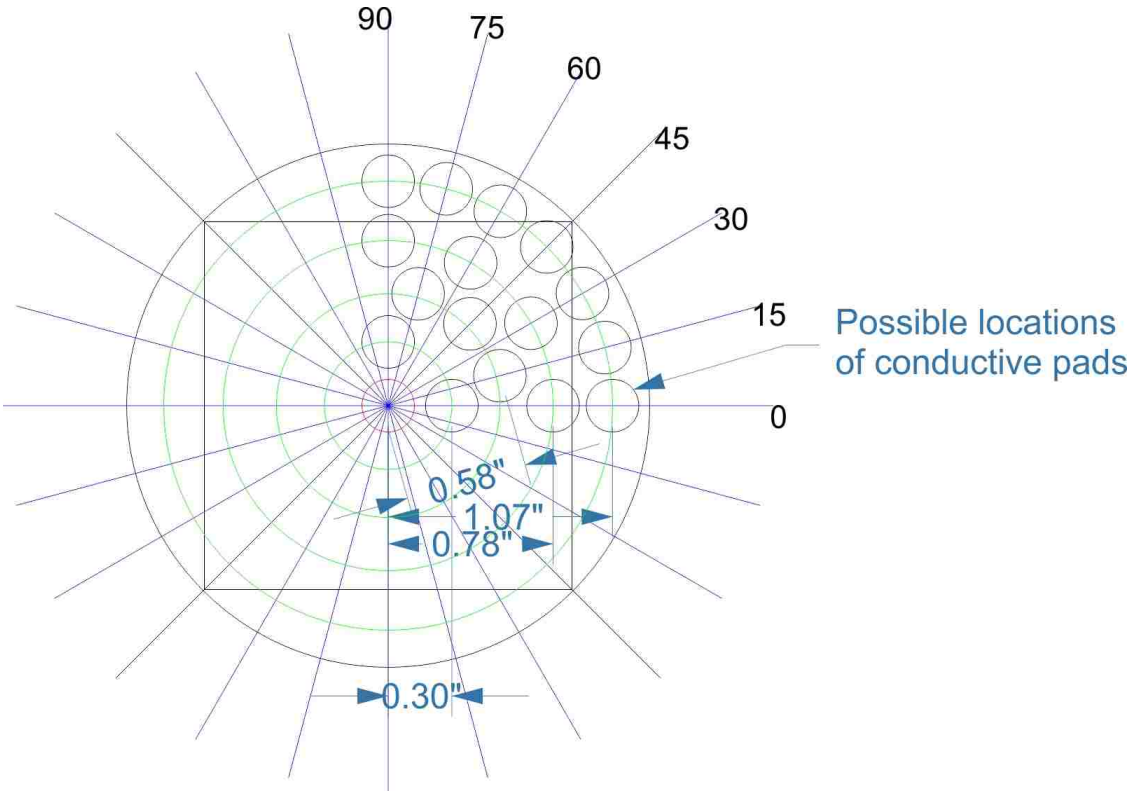

Figure 5.3: Possible locations of conductive pads underneath the objects

uniquely identify individually; major axis, minor axis and the angle. Each pattern has a static center and two other pads. The major axis is the maximum distance from the center pad to either of the two other pads and minor axis is the other distance. The third attribute is the angle inscribed by the the three pads with the center pad as the vertex. Varying these three attributes in incremental steps allowed us to create multiple unique patterns (Table 5.1).

HTML5 introduced the new *touch-events* API, that created a separate space for handling touch based interactions, typical of a mobile device [**?**]. This API also has the capabilities to maintain a set multiple touches currently on the screen along with their coordinates. It also generates events whenever a touch starts, stops or moves. This allows for a continuous event based tracking, instead of repeatedly polling for available touches.

Table 5.1: List of possible unique patterns

ID	Major	Minor	Angle	Ratio	ID	Major	Minor	Angle	Ratio
A	1.07	1.07	90	1	T	1.07	0.3	60	3.5667
B	1.07	1.07	75	1	U	1.07	0.3	45	3.5667
C	1.07	1.07	60	1	V	1.07	0.3	30	3.5667
D	1.07	1.07	45	1	X	1.07	0.3	15	3.5667
E	1.07	1.07	30	1	Y	0.78	0.78	90	1
F	1.07	1.07	15	1	Z	0.78	0.78	60	1
G	1.07	0.78	90	1.3718	AA	0.78	0.78	30	1
H	1.07	0.78	75	1.3718	AB	0.78	0.58	75	1.3448
I	1.07	0.78	60	1.3718	AC	0.78	0.58	45	1.3448
J	1.07	0.78	45	1.3718	AD	0.78	0.58	15	1.3448
K	1.07	0.78	30	1.3718	AE	0.78	0.3	90	2.6
L	1.07	0.78	15	1.3718	AF	0.78	0.3	60	2.6
M	1.07	0.58	75	1.8448	AG	0.78	0.3	30	2.6
N	1.07	0.58	60	1.8448	AH	0.58	0.58	60	1
O	1.07	0.58	45	1.8448	AI	0.58	0.58	30	1
P	1.07	0.58	30	1.8448	AJ	0.58	0.3	75	1.9333
Q	1.07	0.58	15	1.8448	AK	0.58	0.3	45	1.9333
R	1.07	0.3	90	3.5667	AL	0.58	0.3	15	1.9333
S	1.07	0.3	75	3.5667	AM	0.3	0.3	90	1

The first version of the HTML page contained three canvas elements for each of the object wells in the cartouche as shown in Figure 5.1. Later versions combined the three wells within the same canvas element. The elements are so positioned to be directly beneath the cutouts. The algorithm constantly maintains and updates a list of all touches on the screen in an array. The algorithm proceeds as follows:

1. Identify the touch points within the object area by locating touches inside the bounds of the marked area.

2. Identify the center pad by calculating the distance of each touch to the center of the area. The touch with the smallest distance is then selected as the center pad.

3. If the number of touches in an area is three, then an object is placed

(a) Mark point A as the center point and B and C as the other two points.

(b) Calculate the distances from each of the points to the other, viz. AB, AC, BC.

(c) Angle is calculated as $\arccos\left(\frac{AB^2 + AC^2 - BC^2}{2 \times AB \times AC}\right) \times \left(\frac{360}{2\pi}\right)$.

(d) The ratio to the major to minor axis is calculated as $\frac{\max(AB, AC)}{\min(AB, AC)}$

(e) The angle at which the object is currently rotated to is also calculated using similar calculations.

4. The computed *angle* and *ratio* are then compared with a list designating a pattern to an object. A 5° threshold for angle and *0.08* variance for ratio is considered before selecting the detected pattern.

5. Upon selection, the detected object along with the well it is placed is communicated to the server via WebSockets to store the current state of the interface.

6. The entire sequence is repeated for any touch events that are triggered.

WebSockets work over a TCP connection as a part of their protocol, which is a connection-oriented protocol [**?**]. HTML5 also specifies WebRTC which is intended for real-time communication and transfers data over UDP [**?**]. Unfortunately, implementation of WebRTC is currently in development stage and is not available widely on mobile platforms. But with TCP based WebSocket connections the average round trip time was calculated to be around $40ms$ for wired network and approximately $200ms$ using 3G network. The additional delay of 3G network was caused due to packets being routed from Baton Rouge, Louisiana to California then to the server in Georgia, Atlanta, whereas for wired network they were routed directly to Georgia from Baton Rouge. The test server was located at a distance of 500 miles from the device in Atlanta, Georgia. Processing overhead at server side was maximum of 1 *ms*, which started from the time packet was received and ended when a return packet was sent.

5.2 Display Feedback

This section of the device side scripts is rather simple and acts as a frontend to the values stored at the server. The complete UI with various elements is shown in Figure 5.4. The first version maintained a separate WebSocket connection to the server and receives the updated values from the server. Later versions utilized a single WebSocket connection to server for receiving values as well as sending the rotational values of receptacles.

Figure 5.4: Final prototype including UI, cartouche and a receptacle (with a Sifteo Cube)

During the initialization the script identifies the device ID that it is running on and the server then selects the values to be displayed on the device. The server upon a data change event pushes the related values to the device, which is then updated in the GUI by this section. It is also possible to show graphics such as charts and graphs related to the operations being performed locally or remotely.

Additional, interaction of dynamically binding a Sifteo cube + receptacle combination to a parameter is also implemented. To (re)bind cube, the selected parameter is touched on the screen and then the screen of target cube is touched. The actual binding takes place on the server side, and the device only sends the touch events.

5.3 Sifteo Communication

Since we required the data displayed on screen of the Sifteo cubes to change depending on events of other devices, the Sifteo base unit was connected over USB. We used the 2nd generation Microsoft Surface tablets as our target device for Sifteo communication. This was partially influenced by availability of Windows operating system and a dedicated USB port. We originally used Python 2 for interacting with the Sifteo base. But due to issues with network communication over WebSocket and lack of native asynchronous event support for USB events, we transitioned to using Node.js application running on the tablet itself.

The Node.js application uses the *usb* package to communicate with the Sifteo base. It establishes a WebSocket connection to the server, separate from the web-page. Its sole aim is to act as a relay between the Sifteo base and the server. It listens for all parameters and sends the values to the Sifteo base to display. Upon receiving events from the base, it converts into the desired format and sends it to server. The use of Node.js allows for this application to be easily deployed on any Windows, Linux or Mac OS based machines. There is also an implementation of Node.js for Android devices, but it is a developmental release with low functionality. The application on the Sifteo base has the following functions:

1. Upon initialization send the unique hardware IDs of connected Sifteo cubes.
2. Receive the values for parameter to be displayed on the cube.
3. Send touch and shake events to the Node.js application.
4. Manage graphics displayed on cubes.

Chapter 6
Fabrication Process

The fabrication of hardware consisted of three components: the cartouche, the case for tablet, and the receptacle itself. During the entire process we aimed at using environmental friendly material and easily replicable designs. We also intended to have these objects mass-produced easily, hence, used readily available materials for construction.

The most common material used was wood as it is easy to work with. The cartouches are made from matboard, which provide sufficient strength with minimum thickness. We primarily used a laser cutter to cut various sections. For three dimensional objects such as the receptacle, the design was made in multiple layers, which were cut and glued on top of one another. The laser cutter also allowed etching on various surfaces to allow visual identification marks on the objects.

6.1 Cartouche

Our initial prototypes for testing was made out of single walled cardboard, which allowed multiple iterations to be made quickly. Once the layout of the cartouche was completed, we created multiple copies using matboard. During the evolution of the design, we found that cardboard is an excellent choice for cartouches which will be used once. Cardboard and matboard are easily recyclable and cost effective when comparing them with plastics and metal.

We initially started with creating a generic design that could fit multiple tablets, but due to different screen dimensions, the effective screen area that would overlap on all devices decreased. We then focused on creating cartouches for the Microsoft Surface Pro tablets which were closer to the standard letter size dimensions of $8.5in \times 11in$. This allowed for standard letter sized paper/matboard which is easily available. The final version of

the cartouche is a double layered matboard with wells and the windows laser cut. The final design used is also illustrated in Figure 6.1. The bottom layer has circular voids for magnets. Both layers are glued together using paper glue and the neodymium magnet using super glue. While assembly we made sure that the polarity of the magnets matched those embedded within the case.

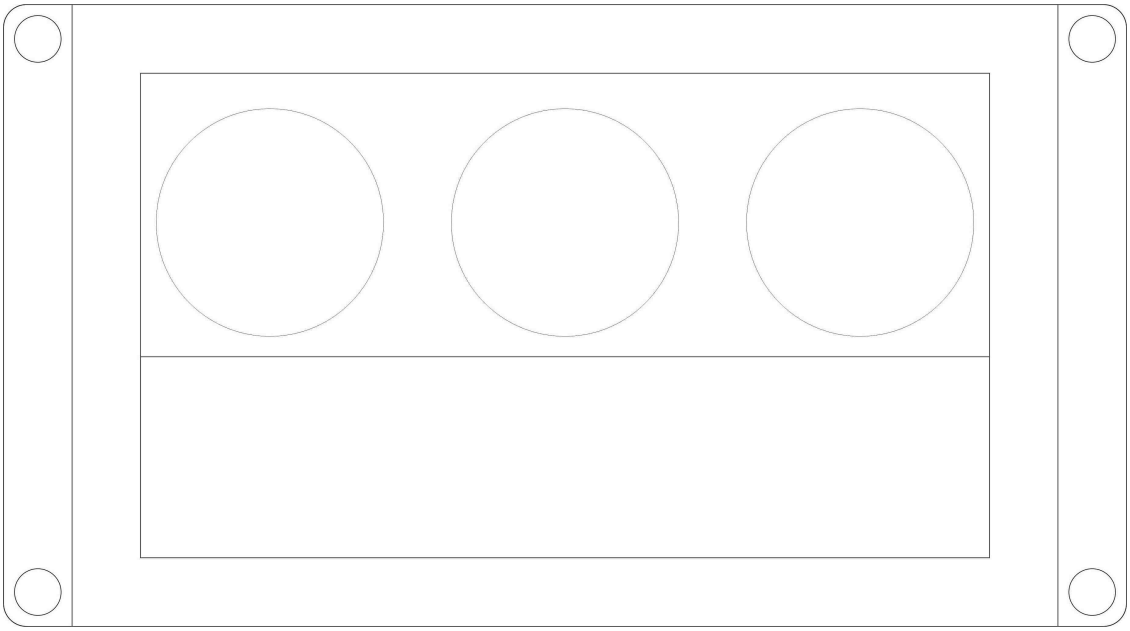

Figure 6.1: Case and cartouche design

6.2 Case for Tablet

While designing the cartouches, we were facing the issue of stability of the cartouche on the screen. Certain tablets have inbuilt magnets that allow flip covers to stick to the screen when closed. A case for the tablet would allow the table to be stationary with respect to the cartouche. We placed strong neodymium magnets in the four corners of the case and replicated those in the cartouche. Figure 6.2 shows the various components that were laser cut and made the case. The case also served as a possible location for embedded electronics and other hardware during our initial evaluation of designs.

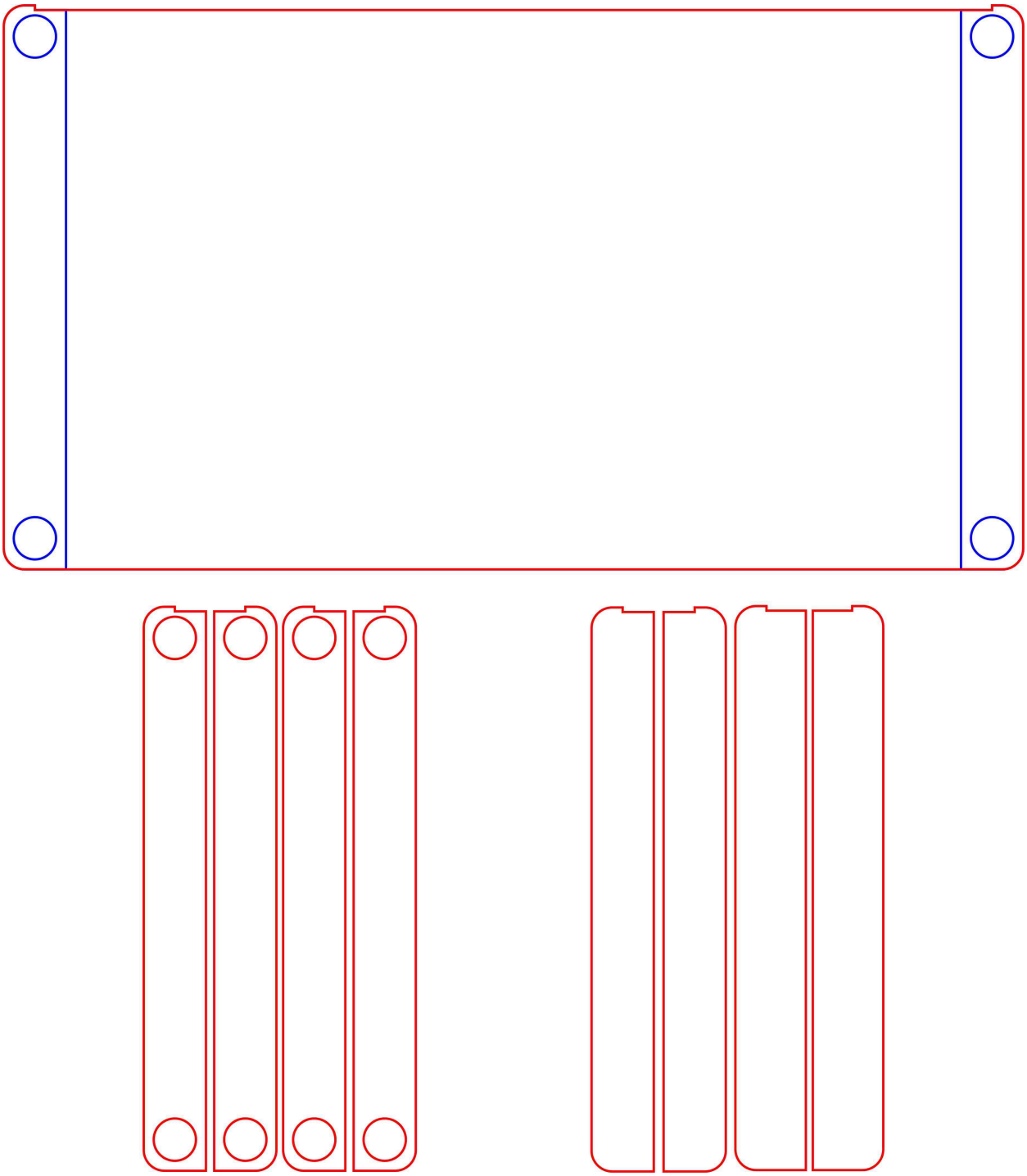

Figure 6.2: Components of the case

The construction of the case is in three layers, each layer is a laser cut from different types of wood. Figure 6.3 shows the various layers of wood when the casier is viewed from the side. The bottom most layer covers the entire area of the case and the tablet sits on top of this. We chose $0.25in$ thick hardboard (or High Density Fiberboard) for the bottom layer as it provided with sufficient strength without adding much weight to the case.

The next layer was cut from $0.5in$ Balsa wood. It also has voids for placing the magnets that hold the cartouche in place. Balsa wood is extremely light weight and slightly deformable. The ability of Balsa wood to deform allows for the edges of the tablet to slightly cut into them and provide cushioning effect, while holding it in place firmly.

The top layer was same in design as the middle layer but without the voids for magnets. This was made from Bass wood, which has more strength than Balsa. The thickness of this layer was $0.125in$, which did not add bulk to the design. This layer prevented the magnets from being ripped out from the case and it creates a slight lip inwards towards the tablet. It also prevents the tablet from popping out and keeping it in place. Due to this design, the tablet could only be slid in from the side.

Figure 6.3: Layers in the tablet case

6.3 Receptacle

The receptacle is a electronically passive element which allows for an object to placed within it and be detected on a touchscreen. Detection on a touchscreen is done by using conductive rubber pads at the bottom in a pattern, which are electrically connected to the copper tape surrounding the receptacle. When an user holds the receptacle, there will be a electrical path from the pads to the user, allowing the capacitive effect of the body to simulate touch points at pads. The receptacles are made purely out of Bass wood with additional metal parts in the second version.

6.3.1 First Version

The first version of the receptacles made to house the Sifteo Cube inside while providing conductive pads at bottom. The conductive rubber pads were stamped out of a sheet $0.125in$ thick. They were then sewed in place using conductive thread while maintaining contact with thin copper tape. This copper tape and one side of the pads is sandwiched between 2 layers of bass wood 1/16"in thick. The thin copper tape was then connected to the broader tape which is covering the side of the receptacle. These features are visible in Figure 6.4 which shows a first version receptacle.

These were light weight and fit in the wells created on the cartouche. The pads worked on different capacitive touchscreens (of various devices) when an user touches the copper tape on side. The initial batch of these were used to test with the detection algorithm running on the tablet.

6.3.1.1 Drawbacks

These first versions had their shortcomings in terms of usability. The major problem was that users had to press down with slight force when operating these on touchscreens as all pads would not make even contact with screen. Another issue that rose was asymmetric

Figure 6.4: First version of the receptacle

distribution of the pads made the receptacles wobble on the screen and loose contact mid-interaction. And when the pads would make contact again, the value registered would jump to the new value.

6.3.2 Second Version

The second version of the receptacles developed upon the problems faced in the earlier versions. To add weight, a $0.125in$ thick circular washer with $1in$ diameter was added beneath the Sifteo cube. To create better electric conductivity, $0.25in$ aluminum tubing was used to house the conductive pads at bottom. This aluminum tubing was in contact of the washer on the side. Additional white plastic dowels were inserted at opposite locations to the

conductive pads to balance them out. Once assembled, the bottom was sanded down to level the pads and the dowels. A engraved colored plastic face was attached to each receptacle to distinguish between two sets of receptacle for two users. Figure 6.5 shows the second version of the receptacles.

Figure 6.5: Second version of the receptacle

Chapter 7
Server-side Application

The server-side application is entirely developed using Javascript and Node.js which is an asynchronous event based, non-blocking I/O based platform that aims in maximizing throughput and efficiency [**?**]. The event based nature of Node.js is highly useful for us as each event results in an asynchronous function call. It also uses Google's V8 Javascript engine to execute code, which is one of the fastest Javascript execution engines. The server-side application also uses multiple worker threads to distribute the load evenly over the multiple processing cores. Each worker thread maintains a separate connection to the Redis based key-value store and MongoDB so as to minimize impact of long data queries on other connections.

The server-side application is distributed into three modules:

- *Communication module:* It maintains the server-side of the WebSocket connections using the Socket.IO library.
- *Redis Store:* This is used to maintain the current state of the entire system by storing variables associated with each device in the system memory.
- *MongoDB database:* All past events, large data chunks of data are stored in this file system based database.

7.1 Communication Module

Most of the clients connecting to the server are HTML5 based Socket.IO connections as discussed in section 5. The Socket.IO library is highly versatile in maintaining numerous simultaneous connections from multiple clients. It also supports Redis based storage for synchronization between separate worker threads along with the ability to create virtual

rooms and add connections to a room. This allows to send a single message to a subset of te connected devices. Along with Socket.IO this module also runs a separate UDP server for non-HTML5 clients to connect.

This module acts as the network facing part of the server application and interprets all incoming messages. All messages received either through WebSockets are of the form *KEYWORD* {*variables*}. Based on the *keyword* it executes queries on the Redis server or MongoDB database. Any incoming data of an UI event is interpreted and the corresponding variables updated in the Redis server and MongoDB asynchronously. During completion of update in the Redis server it broadcasts this change to the devices depending upon the particular variable changes. This is primarily achieved by maintaining multiple rooms of WebSocket connections and broadcasting in the intended rooms. The broadcast message contains the data present in variables of the application itself and not retrieved from Redis and avoiding a redundant data retrieval. A device can be a part of multiple rooms, thus enabling selective UI event propagation. Since, Redis is a memory based key-value store, operations performed on it are extremely fast and the broadcast to other devices occurs with minimum delay. The asynchronous call to MongoDB prevents blocking and delay caused by file based databases. The most updated state is always maintained in the Redis server. The data flow through this module is illustrated in Figures 7.1 and 7.2.

For devices that request a specific data or event, the communication module can handle in two different ways:

1. For recent or the most updated state of the system, the values are retrieved from the Redis server. These type of requests are extremely fast when compared to MongoDB based requests.

2. When a device requests past events or large chunks of data, this is often stored in MongoDB and is slower compared to Redis server.

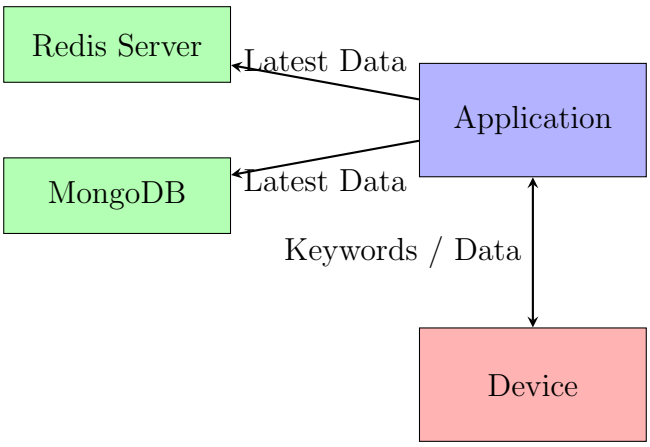

Figure 7.1: Data-flow for incoming data

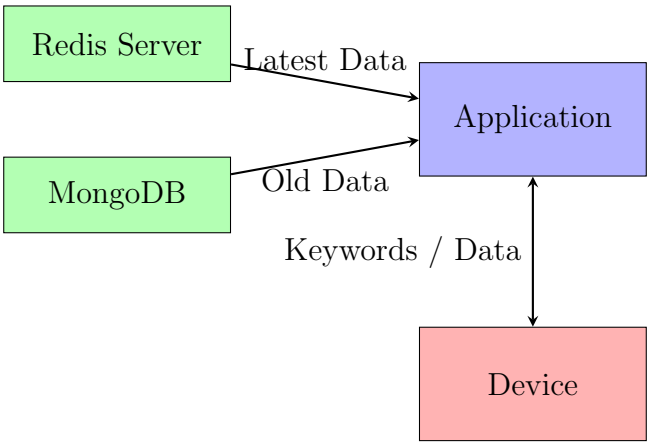

Figure 7.2: Data-flow for outgoing data

Responses to such requests are only communicated back to the requesting device thus allowing a late arriving device to synchronize itself with the current ongoing interaction scenario, or to retrieve the results of an query executed beyond the scope of this application.

We describe the detail functionality of the server in the following sections. There are two types of variables used within the scope of the application:

tokens : They are the digital representation of all interactive elements of the entire system, such as Sifteo cubes, mobile devices, UI elements, etc.

parameters : They are abstraction of tokens and represent the logical variables within the current context. They can depend on values of multiple tokes.

For example an *Area*-parameter will be derived from X-token and Y-token.

7.1.1 Handshake Process

Any connecting client will need to perform the handshake with the server before it is allowed any further actions. Figure 7.3 shows the sequence of packets during a handshake.

1. First packet is an incoming packet from server as, with event `WELCOME` and data as {`question: "who"`}.

2. This needs to be replied with `SETID` event and data containing two fields, {`id: "MyID", type: "MyType"`}. Valid types are `"display"`, `"tablet"`, `"sifteo"`, `"mobile"` and `"query"`. Future versions may introduce a multi-role model. Protected strings that cannot be used for `id` are:

   - `uuid`

3. Server will reply with event `OKSETID` and data as:

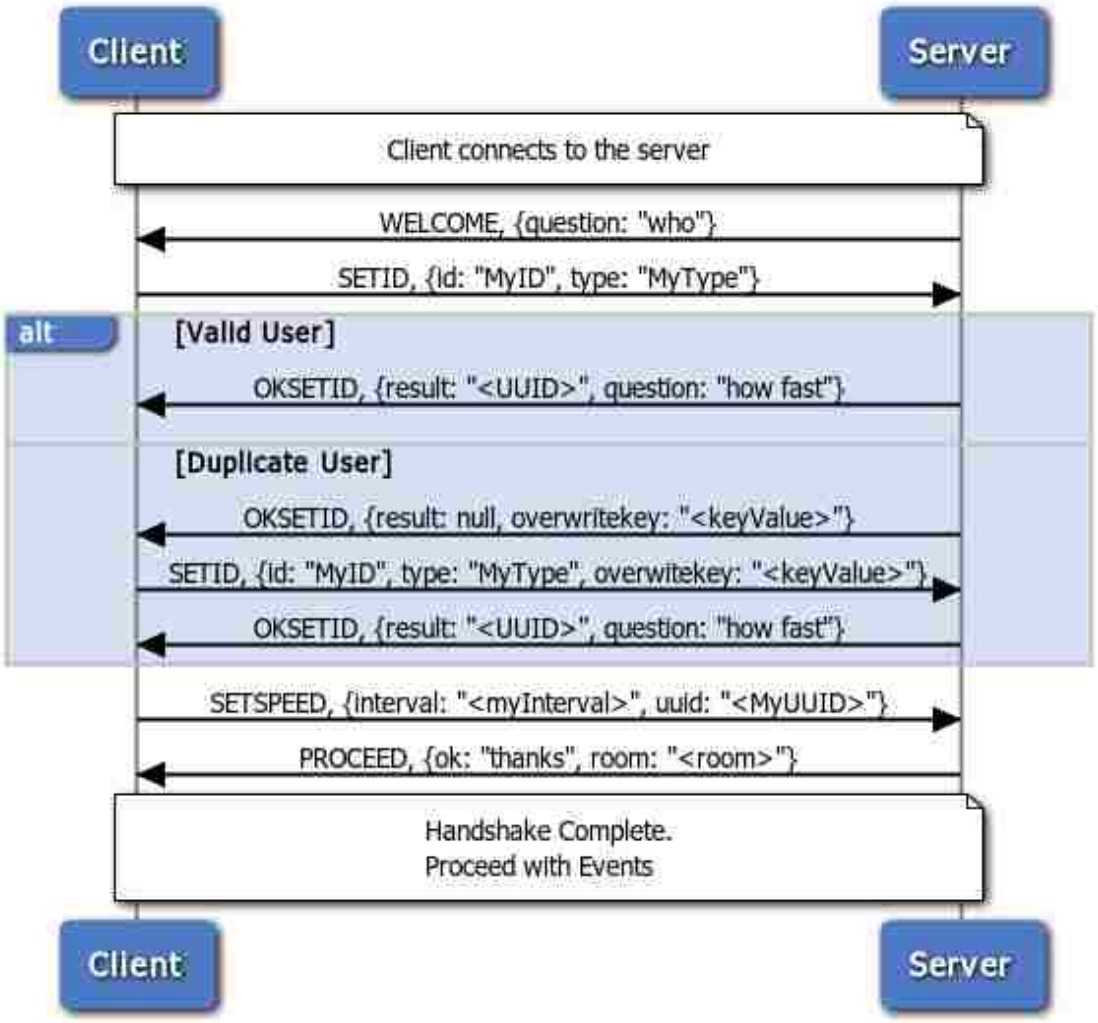

Figure 7.3: Sequence diagram of a connecting client and the server during the handshake.

`result` : This value is either a UUID for the client's session and needs to be provided in subsequent requests, or `null` meaning an active user with the same name already exists. If so, the client needs to restart the handshake from Step 2. above. This will also result in an additional `ERROR` event to be generated with type `"moderate"`.

`question` : This will only be provided when the client gets a valid UUID and the client's ID is unique in the current system. The value will always be `'how fast'`.

`overwritekey` : This parameter is available only when an existing session with the same username and type exist. An additional `SETID` event needs to be sent with this key to overwrite the existing one.

If the previous `SETID` failed due to existing session and it is desired to overwrite the existing one, an overwritekey with new `SETID` event must be sent with an additional parameter `overwritekey` (received from `OKSETID`) to overwrite the last session.

4. After this the client needs to send a `SETSPEED` event with its desired interval of update as {`interval: "<myInterval>", uuid: "<MyUUID>"`}. Valid options are `"instant"`, `"00ms"`, `"00s"` and `"00m"`, where `'ms'`, `'s'`, `'m'` mean millisecond, second, minute respectively; replacing `'00'` with the desired number. The client will get updates of the latest valid changes after every interval specified, with the exception of `"instant"` which will send events as they happen. Restrictions on valid intervals can be implemented.

7.1.2 Sending Events

All event data must be sent with the `EVENT` event. Format for the data to be sent are:

`uuid` : This contains the client's UUID that was obtained earlier during the handshake.

`values` : This is a JSON[1] object, which contains the tokens that were registered. Each key within this object could contain any value as long as the resultant `value` is JSON parsable. Examples can be:

[1]JSON: JavaScript Object Notation

- token-1: {another: "JSON Object"}
- token-2: "Some string"
- token-3: 43 //An int
- token-4: Buffer(binaryData)

Hence resulting data can look like:

uuid : <MyUUID>

values :

- token-1: {another: "JSON Object"}
- token-2: "Some string"
- token-3: 43 //An int
- token-4: Buffer(binaryData)

In the above examples '1', '2', '3' and '4' need to be replaced by the token ID (8-character) string that was received with a OKTOKEN event, like token-jd4us8se. If the client is a type: "query" device, then to send updates for a parameter the client needs to send a DATA event rather than an EVENT event.

7.1.3 Receiving Events

If the client has subscribed to events from any parameter or token the client will receive EVENT events for each subscribed parameter/token separately. The frequency of updates is reflected by the interval defined in SETSPEED. A EVENT event may look like:

- id: "token-wef84m4s" OR id: "param-wef84m4s"
- user: "<originalUser>"
- name: "<MyName>"
- description: "Some description"
- value: {json: "value", as: "object"}

7.1.4 Registering Tokens

Any interaction elements that receive user's inputs are considered as tokens. Every such element must be registered with the system, before any `EVENT` events generated by it are processed. `EVENT` generated by unregistered tokens are ignored. Tokens can be registered by issuing a `REGTOKEN` event with following parameters:

`uuid` : This contains the client's UUID that was obtained earlier.

`name` : An easy to remember name for the token.

`description` : *(Optional)* A string that describes the token.

`ref` : *(Optional)* A string for internal reference for the client side use (Max 32 characters long). This will be returned with the `OKTOKEN` event.

On sending a `REGTOKEN` event, the client will receive a `OKTOKEN` event with the following parameters:

`result` : This value is either a unique 8-character string (ID) for the token and needs to be provided in subsequent `EVENT` events by prepending it to the string `token-`, example: `token-8dj47dh4`. Tokens expire if the user's session expire. Upon re-connection, if a new UUID is established the client will need to re-register the tokens.

`name` : An easy to remember name for the token the client provided in `REGTOKEN` event.

`description` : The string that was provided earlier as the `description`.

`ref` : A string for internal reference for the client's use that was provided with the `REGTOKEN` event.

The difference between `description` and `ref` is that, `description` is visible publicly and can be searched for, whereas `ref` is only returned once with the `OKTOKEN` event and is provided to allow the client to track pending `REGTOKEN` requests asynchronously.

Only the device that registered the token is allowed to send `EVENT` events for that token. Since tokens expire after a user is disconnected, there is no need to delete tokens. A mapping that uses the token also expires when the token is deleted, this is in place to prevent data errors when the token is removed from the context. A new mapping needs to be created upon re-registration of a token.

7.1.5 Registering Parameters

Parameters are values that bounded to tokens. However, tokens can be re-bounded to parameters during runtime, allowing different tokens to control different parameters at various times. Parameters can be registered by issuing a `REGPARAM` event, which is similar to `REGTOKEN` event. It takes the following parameters:

`uuid` : This contains the client's UUID that was obtained earlier.

`name` : An easy to remember name for the token. However, contrasting to tokens, names of parameters should be globally unique.

`description` : *(Optional)* A string that describes the parameter.

`ref` : *(Optional)* A string for internal reference for the client side use (Max 32 characters long). This will be returned along with the `OKTOKEN` event.

Due to the fact that the name of a parameter should be globally unique, the resultant `OKPARAM` event sent after receiving a `REGPARAM` is slightly different.

`result` : This value is either a unique 8-character string (ID) for the token and needs to be provided in subsequent `DATA` events by prepending it to the string `param-`, example: `param-76djlq3b`. However, parameters are not removed from the context if the user who created the parameter disconnects. If this value is `null`, it means that the name provided is already taken. This will also result in an additional `ERROR` event to be generated with type `"moderate"`. In order to overwrite this use the `overwritekey` value also returned with the `OKPARAM` event.

`name` : An easy to remember name for the token the client provided in `REGPARAM` event.

`description` : The string that was provided earlier as the `description`.

`ref` : A string for internal reference for the client's use that was provided with the `REGPARAM` event.

`overwritekey` : This parameter is available only when an existing parameter with the same name exists in the current context. An additional `REGPARAM` event needs to be sent with this key to overwrite the existing one. This is similar to the usage of `overwritekey` in `SETID` event.

By default, every client that registers a parameter is eventually subscribed to its changes (Which depend on the value of interval set using `SETSPEED` event).

7.1.6 Retrieving and Subscribing to Parameters/Tokens

Existing parameters that are registered within the current context can be retrieved by issuing a `GETPARAM` event with the client's `uuid` as the data. The server will then reply with `CURPARAM` with the `result` containing an array of current parameters. Additionally, filters can be applied on `id`, `name` and `description` by providing a regex filter string or an array of strings.

A `GETPARAM` event can look like:

`uuid` : The client's UUID.

`id` : Filter string for id.

`name` : Filter string for name.

`type` : Any of `"any"`, `"display"`, `"mobile"`, `"tablet"`, `"sifteo"` or `"query"`.

`description` : Filter for description. Also, if the *regex* used is too general, the request may take a long time to execute. This will severely impede the performance of the server.

In response, a `CURPARAM` event can look like:

`result` : An array of objects, like: `[param-1, param-2, ..., pram-n]`. Where each of the object can have the structure as:

- `id: "param-mo83m9i2"`
- `name: "Foo"`
- `description: "Bar"`
- `value: <currentValue>`

The client can also use `GETTOKEN` event in a similar manner, with the addition of `user` available as a parameter with the event. The `user` parameter will allow filtering by the user the token belongs to. The server will respond with a `CURTOKEN` event with similar data; an additional field, `user` will be included for each returned token within the result.

`GETPARAM` and `GETTOKEN` are provided as a means to list the current parameters within the system. The value returned may not be up-to-date. This should not be used to retrieve the values of parameters or tokens.

To receive updates to a parameter the client will need to subscribe to changes that affects the parameter. To subscribe the client needs to send a `SUBPARAM` event with the client's `uuid` and `id` of the parameter the client wants to subscribe to. Whenever the value of the intended parameter changes, a `DATA` event will be sent with the updated value. The frequency of updates is determined by the `interval` set during the `SETSPEED` event. Upon a successful subscription the server will send a `OKSUBPARAM` event with `message: "ok subscribed"` and `id` of the parameter subscribed to.

A sample `SUBPARAM` event's data may look like:

- `uuid: <MyUUID>`
- `id: param-rtm58dh1`

And the resulting `OKSUBPARAM` will look like:

- `message: 'ok subscribed'`
- `id: param-rtm58dh1`

Similarly, a client can subscribe to tokens also by sending a `SUBTOKEN` event which would take `id` of the token and the client's UUID in `uuid` field.

7.1.7 Binding Tokens and Parameters

Bindings are relations between tokens and parameters (a parameter-to-parameter binding is also possible, but not parameter-to-token). Bindings contain a set of input tokens (or parameters) and a set of output parameters and a map function. The mapping function can be a simple mathematical function taking in n tokens or parameters and returning m parameters.

There are two ways to create a mapping:

1. Providing a Javascript function.
2. Creating a separate client that runs long computations.

These are detailed in the following sections. In the following sections the term token and parameter can be used interchangeably, except that a mapping to a token is not allowed.

7.1.7.1 Using Javascript Functions to create Mapping

The client can provide a Javascript function that calculates the output parameters and is executed within the environment of the server itself. For this user-defined function the server application passes the previous value and the current value of the token (that triggered the mapping function). Along with the input tokens, and array of output tokens is also passed as `outputTokens` for manipulation. The client should modify this, and return it back from the function. This method of mapping is faster than the `query` type, as the function is executed within the server during an `EVENT` event.

This function should take three parameters: `inputTokens`, `prevInputTokens` and `outputTokens`. And it should return the modified `outputTokens`. Each of the three variables is an array of token objects. Each token/parameter object contains:

- `id: "token-k283hdw8"` OR `id: "param-k283hdw8"`
- `user: "<originalUser>"`
- `name: "<MyName>"`
- `description: "Some description"`
- `value: {json: "value", as: "object"}`

An example of a mapping function that does 1-to-1 mapping by averaging can look like:

```
function (inputTokens, prevInputTokens, outputTokens){
//inputTokens and prevInput is an array of subscribed tokens/parameters
  for(var index = 0; index < inputToken.length; index++){
    outputToken[index].value =
    ((prevInputTokens[index].value + inputTokens[index].value) / 2);
  }
return outputTokens;
}
```

It must be noted that for all cases of input values the function should not run for more than 100ms. The client that created the mapping will also get run times for the function. The client will get an `ERROR` event with a warning if the runtime exceeds 100ms. Any functions that run for more than 500ms are automatically killed and `ERROR` event generated with `type: "moderate"`. For longer computations it is advised to use the `query` type device.

To create a mapping between n tokens/parameters and m parameters, you will need to send a `REGMAP` event. The data to send with this event is as follows:

- `uuid: <MyUUID>`
- `name: "<MyMappingName>"`
- `description: "<MyMappingDescription">`
- `type: "function"`
- `inputs: [<Array of token/param ids, like "token-s89k9d84">]`
- `triggers: [<Array of token/param ids>]`
- `outputs: [<Array of token/param ids>]`
- `map: <JSON object>`
- `ref: <internal ID>`

Mapping function is invoked when any of the objects in `triggers` are present within an `EVENT` event, and not the `inputs`.

The value of `map` above should be a JSON object with a function called `mapper` available. This function should take three inputs, `inputTokens`, `prevInputTokens` and `outputTokens`, while returning an array of `outputTokens`. Secondly, this function is executed within a sandbox, hence, no external variables will be accessible from within the function.

Upon creating a mapping, the server would send a `OKREGMAP` event with the following data:

`result` : This is the id of the mapping created. The client will need this to delete the mapping if required. The value is an 8-character string and should be prepended with "map-", like `text`. If this value is `null` then creation of mapping failed. If so, then the client will also receive an `ERROR` event with the detailed message.

`type` : This value will be `"function"` if the client is sending a function in the `"map"` variable.

`inputs` : The same array that was provided with `REGMAP`, but with full token objects instead of string of ids.

`triggers` : The same array that was provided as triggers, but with full token objects.

`outputs` : The same array that was provided, but with full token objects.

`ref` : This will be the same value that was passed earlier.

There can be only one mapping between any pair of token/parameter (that is, the sets of inputs and outputs should be unique), even if the mapping is a many-to-many. This is checked within the code and the client will receive a `null` value in `result` if this rule violated along with an `ERROR` event with `type` as `"moderate"`. A mapping is disabled when a token disappears from the context, if the token with the same id re-appears the mapping

is re-activated. Mappings that are disabled for more than 30 minutes are removed and new mappings must be created. The client can get current mappings by sending a `GETMAP` event. One can also create multilevel mappings by creating parameters to serve as intermediate variable, hence they act as output of one mapping and as inputs to another.

7.1.7.2 Using `query` Devices to create Mapping

If the operation that the clients desire are long running or require the use of executing queries on a separate database, they should create a separate client (to the server) that does this. While sending the `SETID` event, the device `type` should be set to `query`. `query` type clients off-load the computational task of mapping from the server. They act as normal clients, but have the special privilege of sending `DATA` event for parameters instead of tokens. These clients are not allowed to register tokens. They can however register parameters. The process of setting up a `query` client is simple:

1. Perform handshake (with `type: "query"`) with the server and obtain the UUID.
2. Subscribe to `EVENT` events of the tokens/parameters that the client is interested in.
3. Upon completion of computation, the client can send a `DATA` event, but instead of using `token-` in `values`, the client will need to use `param-`.

Even though parameters are updated using a `DATA` event, subscribers will still receive it as an `EVENT` event.

There is no limit on the time it takes for a computation to complete. The client can send `DATA` event any time they require. The clients can also choose to ignore subsequent `EVENT` events if the computation is still in process, if the interaction model allows. The clients should also retrieve current mappings regularly to check if they are not creating a duplicate mapping. Also, if it is possible, a `REGMAP` event should be sent with `type: "query"` and the `map` parameter should be ignored. This will help others in finding suitable mappings.

7.1.7.3 Managing Mappings

Any device can retrieve mappings within the system by sending a `GETMAP` event. The data to sent with this event is:

`uuid` : The client's UUID

`inputs` : A search (regex) string to filter the inputs. Also, this can be an array of strings with the ids of tokens/parameters.

`triggers` : A search (regex) string to filter triggers. An array strings can also be provided.

`outputs` : A filter string or an array of strings.

`name` : A (regex) string or an array to filter user who created the map.

`description` : A (regex) string or array to filter by.

Out of these parameters `uuid` is required and at least one of `inputs`, `triggers`, `outputs`, `name` or `description`. On receiving a valid `GETMAP` event the server will respond with a `CURMAP` event with the following data, containing only the `result` variable, which is an array of mappings. The mapping objects within this array can look like:

- `name: "<MyMappingName>"`
- `description: "<MyMappingDescription">`
- `type: "function"` or `type: "query"`
- `inputs: [<Array of token/param ids>]`
- `triggers: [<Array of token/param ids>]`
- `outputs: [<Array of token/param ids>]`
- `user: "<createdBy>"`

The actual mapping function is not returned with the mapping object when searching. In order to remove a mapping, a `DELMAP` event can be sent. The required parameters for this event are:

`uuid` : The client's UUID.

`mapping` : This can be a object, like {`id: "map-8fheu26m"`}" or an array of valid objects.

In response to this the server will send a `OKDELMAP` event. This will have a single `result` variable, which will be always an array of objects, with each object having the following parameters:

`id` : This is the id of the mapping to be deleted.

`result` : This will be `true` or `false`. A `true` means that the mapping was successfully deleted, whereas a `false` will mean that the user who issued the `DELMAP` event is not the owner of the mapping.

`overwritekey` : This will only be sent if the result was `false`, as the user who is trying to delete the mapping is not the owner of the mapping. This needs to be sent back to confirm the delete.

If in case that the mapping that is being deleted was created by another user, then the `result` will be `false`, additionally the server will also send the `overwritekey` with the `result`. To confirm the delete, the client will need to send another `DELMAP` event, but this time, within the `mapping` parameter containing the mapping object; the object should contain the `overwritekey`. For example, a mapping object within the `mapping` can look like, {`id: "map-wyt5h38v", overwritekey: "<recievedOverwritekey>"`}". The client can add multiple of such objects with the `DELMAP` event or send a separate one for each mapping.

7.1.8 Special Handling of Sifteos

Due to the technology present in Sifteo cubes, it is not possible for events related to a Sifteo cube, originate from a single source. There is generally a device that is connected to the Sifteo Base (via USB) and a different device which would sense the physical location and orientation of the cubes. Due to this the current model would have multiple tokens for the same physical cube. This is unfavorable for topology of the entire system, as only the device that registers the token can send `EVENT` event to manipulate it.

An approach to solve this could include a middle-man that aggregates the data from the two sources and then connect to server as a device. This would add latency for the events actually happening and them being transmitted. Another approach keeping the current system model in place would include creating a parameter, which would be the aggregation point of both the sources, by creating a 2-to-1 (two tokens to parameter) mapping and then using this intermediate parameter to be further linked. This increases the number of processing nodes, hence introducing delay.

To solve this problem, the simplest solution is to make tokens registered by the `type: "sifteo"` devices, writable by others. However, this implementation could make *sifteo tokens*[2] globally writable, and anyone searching for a similar name could potentially overwrite it.

Special rules that apply to `type: "sifteo"` devices are:

1. All *sifteo tokens* contain an additional parameter `sifteo-key`, which either can be defined during token registration or dynamically generated during registration. It will always be present in the `OKTOKEN` event.

[2]Tokens generated by devices with `type: "sifteo"`

2. Value of `sifteo-key` remains unchanged during the lifetime of the token.

3. Only `type: "sifteo"` devices can write to *sifteo tokens*. If writing to tokens of other device, include the `sifteo-key` for that token.

4. Only when a `type: "sifteo"` device sends a `GETTOKEN` event, any *sifteo tokens* returned will contain the `sifteo-key` for that token.

It is advised to create a separate client with `type: "sifteo"` on a device that is responsible for supplement data of Sifteo cubes, as a single physical device can have multiple clients with different `type` values.

7.1.9 Errors

Any and all errors will result in an `ERROR` event being sent. The data for this event is as follows:

`message` : This will contain the message for the user, detailing about the error.

`input` : This will contain the event name that was received that caused the error.

`type` : This contains the values for various types of errors. The possible values are:

- `"fatal"` : This generally means an exception was thrown within the code and the program broke its normal execution. This can also mean that the program may have survived the crash. A connection reset is advised.
- `"moderate"` : The command that the tried earlier could not succeed, generally due to incorrect data being passed or the received data would result in an invalid state. The last command should not be retried before checking the status and data.

`"warning"` : This means that the last command was executed successfully but might have resulted in unstable state of variables/status which could result in synchronization errors or might have overwritten a value, without considering the previous value.

7.1.10 Keep-Alive

Socket.IO maintains a socket alive for 60 seconds before killing it. The client protocol sends a `ping` packet ever 25 seconds. The authorization keys in *Redis* will expire after 60 seconds if no packets are received. Hence, this should not be worried about if using a standard socket.io client. For custom clients, at least a `ping` packet is expected within 60 seconds.

7.2 Key-Value Store using Redis

The main concern for handling UI based events is the latency introduced due to computation performed based on the event data. A larger delay in the feedback loop from the action to output results in dis-coordination and disruption of user's interaction work-flow. Using asynchronous calls and non-blocking I/O we aim to minimize this delay. Storing variables in system memory always provides the least latency, but overheads caused by this generally exceed the platform restrictions on a server. Using a key-value based store is also beneficial for fast retrieval, as there are no tables for the server to operate upon. Redis is based upon this ideology of a fast memory based key-value store [**?**].

Our application utilizes this feature of Redis to store the current state of the entire system, including the associations, variables, positions and states of all interactive parts. This allows for a speedy update and retrieval. Each of the device (that has previously participated or is actively participating) is assigned an address space in Redis such as {*physical-location*}:{*logical-location*}:{*device*}:{*property*}.

The execution speed of Redis commands minimizes the chances of a value being overwritten. To further minimize this effect all commands executed atomically on Redis. Due to the nature of an interaction scenario, such event would be nullified upon update from an UI event.

7.3 Persistent Storage using MongoDB

For long-term storage and archival purposes, we favored document-oriented, No-SQL based, MongoDB implementation versus a traditional table-based database systems [**?**]. MongoDB is used to store JSON-style (called as BSON, derived from Binary JSON) documents. This allows a low overhead when retrieving data to Javascript environment of Node.js. All events that occur are stored in each device's individual document, to create a timeline of the events. By using ad-hoc queries of MongoDB any frame of a device can be re-constructed for future devices that depend on actions that took place on other device. Following a similar addressing scheme of Redis, the MongoDB contains documents at root-level denoting the various physical locations participating. Within each root-document is an array of references to various logical locations. The devices are further listed within these logical location documents. This organization is illustrated in Figure 7.4.

Computed data from various analysis tools are also stored in a similar architecture incorporating the logical location of the inputs for the data.

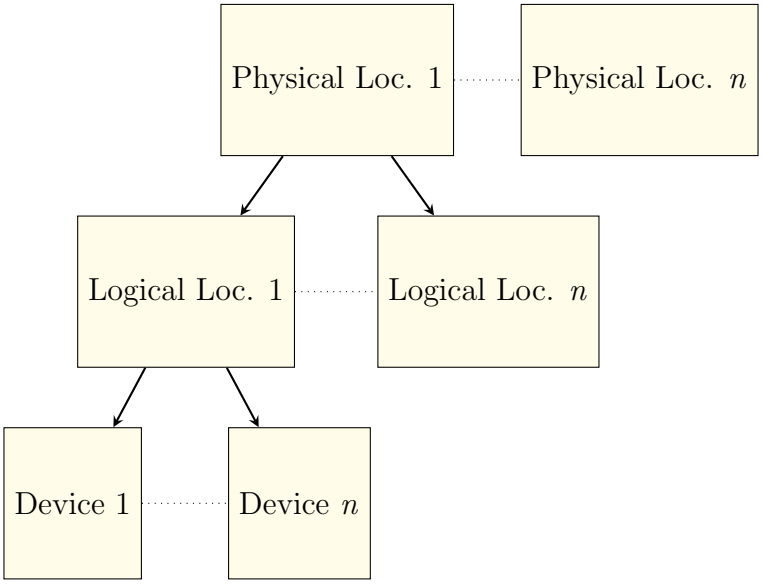

Figure 7.4: Document organization in MongoDB

Chapter 8
Future Work

This system provides ample of scope for expansion to incorporate various applications and parameters incorporated in many research workflows It can be expanded to include other devices, platforms and controls. Integration of research tools, such as Matlab, R, python, etc would create a single system capable of creating and executing queries. Additionally, generating events based on sequence of actions would allow for complex gestures to be defined.

From communication perspective, there are two useful expansions possible. The first would be to create connection-less communication channel between connected devices. This would allow for a lower network overhead and would be suited for devices with flaky-connection. Second expansion could incorporate intra-device communication to provide redundancy in case of server or network failure. Intra-device communication would allow for subsets of connected devices to communicate and ongoing but restricted interactions. Further expansion of communication channels to include non TCP/IP based protocols (such as USB, Serial, Bluetooth, etc.) would allow to include additional devices within the interaction scenario.

Additionally, it is also possible to serialize the packet being send in binary format by incorporating libraries such as *msgpack*, but the effect of encoding and decoding on processing time needs to be studied. For significantly small datasets (or subsets of a large datasets) it is possible to store the dataset at the client side by using *IndexedDB* or *WebSQL* for local querying.

Chapter 9
Conclusion

The system described above is well suited for multi-location interaction scenario incorporating multitude of mobile devices. The UI requirements for these devices include a network connection and a modern HTML5 browser. Current browser support includes Internet Explorer 10+, Firefox 6+, Chrome 14+, and others for Windows, Linux and Mac; iOS Safari 6+, Chrome for Android 33+, Firefox for Mobile 26+ for mobile operating systems.

Maximum memory usage for Redis store was $2.2MB$ during our tests which included 10 devices and a single server. The average latency was $40ms$, which is comfortable for human interaction. Total load on server was minimal even when events were being generated at a minimum interval of $20ms$. Continuous Redis polling at 500Hz and more also resulted in no performance degradation of the Redis Server. Simultaneous value update by two different devices updating same parameter also resulted in zero errors. Event propagation between devices was also with minimal latency and was hardly perceived by the users.

For the receptacle's conductive pads the surface should be flat, since the effectiveness of the virtual touch depends upon the pressure and the area in contact. A flat surface ensures larger contiguous area touching the screen and providing a stable touch point to track. Use of wood also eliminates the RF interference of metal with on-board wireless transceivers. Also, the signal strength of the connected network severely affects the latency. At low signal strengths this effect is pronounced and causes degradation of user experience.

Vita

Shantanu Thatte was born in Jabalpur, India in November 1989. He completed his high-school from Choithram School, Indore. He received his undergraduate degree of Bachelor of Engineering in Information Technology from University of Pune, India in 2012. He joined Louisiana State University, Baton Rouge to pursue Masters of Science degree in Systems Science in January 2013. He expects to successfully graduate from Louisiana State University in December 2014.