

2013-05-23

Developing a Usable API for Multi-Surface Systems

Burns, Christopher

Burns, C. (2013). Developing a Usable API for Multi-Surface Systems (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from <https://prism.ucalgary.ca>. doi:10.11575/PRISM/25621
<http://hdl.handle.net/11023/727>

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Developing a Usable API for Multi-Surface Systems

by

Christopher Charles Burns

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

MAY, 2013

© Christopher Charles Burns 2013

Abstract

A multi-surface system brings together a variety of different devices – such as a tabletop, tablet, mobile phone and wall display – into a single cohesive system. This integration allows users to take advantage of the unique capabilities of each device in ways that would not be possible using those devices separately. But creating usable interactions for moving content and control between all these devices has proven a difficult problem. Spatially augmented gestures, which are gestures which incorporate the spatial layout of the room as well as the people and devices in it, might provide a solution to this problem. Building such gestures into a multi-surface systems is difficult and tedious to develop. It represents too large an investment of time and effort for developers to bear. To decrease the cost of developing such systems, we have created an API – called MSE-API – that allows developers to quickly and efficiently add gestural interactions to multi-surface applications. In developing such an API we focused especially on making it usable for developers. Specifically we insured the API was learnable and discoverable for inexperienced developers but still an efficient tool for more experienced developers.

This thesis presents the requirements and structure of an API for developing multi-surface systems with spatially augmented gestures. The result of two case studies, in which the API was used to develop real world multi-surface applications, are also presented.

Acknowledgements

For bringing me out of a tutorial and into the lab, I would like to thank Tedd. His countless hours of editing and revising made sure my papers and this thesis were even readable. It was because of him that I pursued graduate school at all and it was because of him that I finished.

For his mentorship and support I would like to thank Dr. Maurer. He made sure I never got off track. His thoughts, feedback and advice were invaluable in completing this work.

For their humour, support and friendship I would like to thank Tulio and Abhi. Without the support and approval provided by Tulio— especially for MRI Kinect— I doubt this important project would have even been created. Without their presence, the lab would not have had such an atmosphere of tolerance, respect, and professionalism.

For their work on developing MSE-API I would like to thank Arlo, Patrick and Daniel. Without you my work would not have been possible.

For her patience with the long hours and stress, for her love and support I'd like to thank my girlfriend Georgette.

For a four year partnership on classes, papers, research and projects, I'd like to thank Teddy. Our late nights and long hours paid off through two degrees. We will definitely go on to accomplish even more.

For their love and support I'd like to thank my family. For my brother, whose hard work and focus has always been an example. For my Mom, who never let me quit, even when things were hard. If you hadn't put those first books in my hands, I never would have written this one. For my Dad, who set the bar high and always helped me meet it. Our long talks, your guidance and support are visible in all my work and accomplishments.

Dedication

To Mom and Dad

Table of Contents

Abstract	ii
Acknowledgements	iii
Dedication	iv
Table of Contents	vi
List of Tables	x
List of Figures and Illustrations	xi
Epigraph	xiii
 CHAPTER ONE: INTRODUCTION	 1
1.1 Research Questions	2
1.2 Goals	3
 CHAPTER TWO: MULTI-SURFACE SYSTEMS AND APIS	 4
2.1 Multi-surface Systems – Benefits & Overview	4
2.1.1 Effective Use of Device Properties	4
2.1.2 Public & Private Space	5
2.1.3 Improved Collaboration	6
2.1.4 Novel Interactions	8
2.1.4.1 Seismic Slicing	8
2.1.4.2 MRI Kinect	10
2.1.5 Summary of Benefits	11
2.2 Infrastructure to Support Multi-surface Systems – Content and Control Transfer	12
2.2.1 Control & Content Transfer	12
2.2.1.1 Content Transfer	13
2.2.1.2 Control Transfer	14
2.2.2 Spatially Augmented Gestures	15
2.3 An API for Multi-Surface Systems	17
2.3.1 Communication	18
2.3.2 Spatial Tracking	19
2.3.3 Summary of Requirements	20
2.4 Usability of the API	21
2.4.1 API Usability	21
2.4.1.1 Learnability & Discoverability	22
2.4.1.2 Efficiency	23
2.4.1.3 Satisfaction	23
 CHAPTER THREE: RELATED WORK	 24
3.1 Interaction Techniques	24
3.1.1 Graphical Interfaces	25
3.1.1.1 Menu Based	25
3.1.1.2 World in Miniature	26
3.1.2 Physical & Proxemic Approaches	27
3.1.2.1 Docking	27
3.1.2.2 Conduits	28
3.1.3 Gestural Approaches	29

3.1.3.1 With Device Gestures	30
3.1.3.2 Gestures & Proximity	32
3.1.4 Interactions Summary	34
3.2 APIs Supporting Multi-surface Systems	35
3.2.1 Device Communication	36
3.2.1.1 3MF	36
3.2.1.2 Event Heap	37
3.2.1.3 MAGIC Broker	38
3.2.1.4 ROSS	39
3.2.2 Proximity & Location	40
3.2.2.1 Proximity Toolkit	40
3.2.2.2 Easy Living	42
3.2.2.3 NearMe Server	43
3.2.2.4 Shared Substance	43
3.3 API Usability	44
3.3.1 API Usability Issues	44
3.3.1.1 Naming & Concepts	45
3.3.1.2 Design	45
3.3.1.3 Documentation	46
3.3.2 Evaluation Strategies	46
3.3.2.1 User Studies	47
3.3.2.2 Review Processes	48
3.3.2.3 Measurement Methodologies	48
3.4 Conclusion	51
CHAPTER FOUR: MSE-API	52
4.1 Requirements	52
4.1.1 Constraints	52
4.1.1.1 Consumer-Accessible Hardware	53
4.1.1.2 Removing Markers	53
4.1.1.3 Platform Support	54
4.1.2 Functional Requirements	55
4.1.2.1 Inter-Device Communication	55
4.1.2.2 Spatial Information	56
4.1.3 Usability Requirements	57
4.2 API Components	57
4.2.1 Locator	58
4.2.1.1 Locating Devices	60
4.2.1.2 Querying the Locator	61
4.2.1.3 Locator Methods	62
4.2.2 Client Libraries	63
4.2.3 Visualizer	64
4.2.3.1 Visualization of Devices	64
4.2.3.2 Room Configuration	65
4.3 API Usage Example – Pour Gesture	65
4.3.1 Detection & Query	65
4.3.2 Sending & Receiving the Image	66

4.3.3 Sequence of Messages	66
4.3.4 Summary	67
4.4 Feature Comparison	68
4.4.1 Communication Features	68
4.4.2 Spatial Locator Feature	69
4.5 Conclusion	70
CHAPTER SIX: SKYHUNTER CASE STUDY	71
6.1 Skyhunter & Data Problems in Oil & Gas Exploration	71
6.1.1 Types of Data	72
6.1.1.1 Microseeps	72
6.1.1.2 Subsurface Data	72
6.1.1.3 Well Data	73
6.1.2 Roles in the System	74
6.1.3 Issues & Difficulties	75
6.1.3.1 Data Entry and Exit	75
6.1.3.2 Fluid Data Transfer	75
6.2 Skyhunter MSS Application	75
6.2.1 System Components	76
6.2.2 Data Transfer Features	77
6.2.2.1 Tablet to Tabletop	77
6.2.2.2 Tablet to Tablet	77
6.2.2.3 Tabletop to Tablet	79
6.2.3 Issues Addressed	79
6.2.3.1 Supporting Data Entry & Exit	79
6.2.3.2 Fluid Data Movement	82
6.3 Study Results	82
6.3.1 Results	82
6.3.2 Discussion	84
6.3.2.1 Time to Completion	84
6.3.2.2 Other Lessons Learned	84
6.3.3 Threats to Validity	85
6.4 Conclusion	85
CHAPTER SEVEN: C4I CASE STUDY	87
7.1 C4I & Emergency Planning	87
7.1.1 Emergency Planning & Simulation	88
7.1.2 Roles & Content	88
7.2 C4I MSS Application	89
7.2.1 Structure of the System	89
7.2.2 Transfer Features	90
7.3 Study	92
7.4 Results	94
7.4.1 Time Logs	94
7.4.2 Code Analysis	95
7.4.3 Questionnaire	98
7.4.4 Discussion	100

7.4.5 Threats to Validity	100
7.5 Conclusion	101
CHAPTER EIGHT: CONCLUSIONS	102
8.1 Thesis Contributions	102
8.2 Future Work	103
8.2.1 Improving the Spatial Engine	103
8.2.2 Data Fusion	104
8.2.3 Further Evaluation of the API	104
REFERENCES	105

List of Tables

Table 1: Summary of Interactions.....	34
Table 2: Routes Provided by the Locator	62
Table 3: Comparison of Existing APIs on Communication Features.....	69
Table 4: Comparison of Existing APIs on Spatial Location Features	69

List of Figures and Illustrations

Figure 1: iPad Seismic Slicing.....	9
Figure 2: MRI Table Kinect Slicing	11
Figure 3: Proposed Gestures from Elicitation Study [9].....	16
Figure 4: Menu Based Approach for Device Selection. [16].....	26
Figure 5: World in Miniature Approach for Device Selection. [8].....	27
Figure 6: Stitching as a Method for Content Transfer. [20]	28
Figure 7: Physical Bridging as a Method for Content Transfer. [25]	29
Figure 8: Throwing Gesture as a Method of Content Transfer. [26]	31
Figure 9: Flicking Gesture as a Method of Content Transfer. [30]	32
Figure 10: Pointing Gesture as a Method of Content Transfer. [30]	33
Figure 11: MSE-API Components in an MSS.....	58
Figure 12: Simplified Architecture of MSE-API Locator	59
Figure 13: Pairing Gesture Performing With Device	61
Figure 14: Visualizer displaying a paired person and device	64
Figure 15: Detecting Pour Gesture & Querying Locator.....	66
Figure 16: Sending & Receiving an Image.....	67
Figure 17: Sequence of Pairing & Orientation Updates	67
Figure 18: Sequence of Locator Query & Content Transfer.....	68
Figure 19: Flight Grid Pattern.....	73
Figure 20: Subsurface Contours.....	74
Figure 21: Role Selection in Skyhunter MSS	76
Figure 22: Pour Gesture to Transfer Layers	78
Figure 23: Flick Gesture to Transfer Layers.....	80
Figure 24: Camera Gesture to Transfer Layers.....	81

Figure 25: Development Time (Person-Hours) by Category.....	83
Figure 26: Examining Visible Layers Using a Web Browser.....	85
Figure 27: Pour Gesture to Transfer Annotations.....	91
Figure 28: Button Press to Capture Entities.....	93
Figure 29: Experience Level of Participants.....	94
Figure 30: Time Spent on Tasks	95
Figure 31: Initializing the API	96
Figure 32: Sending & Receiving Extents.....	97
Figure 33: Requesting Entities.....	98

Epigraph

In academia, the number one sin is plagiarism, not triviality. So much of the innovation is esoteric and not at all useful.

Peter Thiel

Chapter One: Introduction

For many years, users interacted with a computer with a mouse and keyboard. They worked on applications which ran on that computer or web-based systems through a web browser. Their files and applications lived on that computer alone and if they needed to be moved they were sent over a network or stored on physical media. This situation has changed drastically in the last decade; users today likely have a smartphone and/or a tablet in addition to a traditional computer. Their files live across these devices and are commonly synchronized between them. Users might now have access to large format wall displays for collaborative work and even newer technologies, such as digital tabletops, or position tracking systems. All these devices are rapidly reaching consumer level prices (i.e. measured in hundreds of dollars).

In this new computing environment, the old reality of *one application, one device* is rapidly fading away and does not meet the needs of users anymore. In the old paradigm an application “lives” on a single computer but in an environment with numerous connected devices, this paradigm is rapidly fading away. Such connected devices might include a mobile phones, tablets, large format wall display and even digital tabletops. Traditionally each of these devices was used separately with each application and its associated data being independent from other systems. As their use increases, users want to be able to use these devices in an integrated environment. But in order to support such an environment it is necessary to be able to move content between a device – a common task for users – which, in existing systems, is clumsy and difficult. Controlling one device from a more convenient device, another important capability, is rarely even supported. Consider the task of moving a document from a wall display to a mobile device during a presentation. Using the traditional paradigm this would typically involve sending the file over email or transferring it with a USB. While these information exchanges are

commonplace and occur many times in a given work session, the interactions for accomplishing them are poorly-supported, often take place outside the context of the application in which they're required, and are time-consuming. To deal with these usability issues, designers need to reconsider the fundamental paradigm of applications which run on single device and adapt to multi-device computing. Developers need to adapt to building applications which are, by design, *spread across* multiple devices. By this we mean applications where the movement of content and control between devices is so smooth and fluid that the application no longer appears to be localized to a specific device.

While developers in industry have been slow to consider the new situation of multi-device systems, researchers working in the field of human-computer interaction have been investigating this paradigm for over two decades. Research has emphasized several important benefits which are provided by multi-display environments (MDEs) – environments containing multiple displays which may or may not be interactive. For our purposes, we define a multi-surface system (MSS) to be a system composed of multiple, tightly integrated, interactive devices. In applications running on an MSS users can switch seamlessly between the devices and the application appears to spread across the devices and is not localized to a single device. Under this definition therefore MSSs are a distinct subset of MDEs. Detailed definitions of terms are provided in Chapter Two.

1.1 Research Questions

This thesis proposes the creation of an API which supports gestural interactions in a multi-surface environment. In addition to providing useful functionality to developers, the API is intended to be usable. This leads to the two major research question of this work:

- 1) Can an API for constructing multi-surface environments be built which uses only consumer level hardware?
- 2) How learnable and discoverable can the API be made while still preserving efficiency?

In answering these questions it will be necessary to confirm that the API provides all the features necessary to construct a multi-surface system with spatially augmented gestures. To do this it's necessary to show that such a system can be built and all the requirements discussed in Section 1.3 have been met. To evaluate its usability it will be necessary to show that the API is learnable, discoverable and productive when used by developers and that they are satisfied with the API.

This will be measured by a case study and a self-evaluation of the API. We will measure whether developers can learn to use the API in a reasonable time frame, how productive they are in building an MSS, and how satisfied with the API they were during various stages of their project.

1.2 Goals

This thesis has two main goals: to develop and evaluate the usability of an API which supports spatially augmented gestural interactions in a multi-surface system. In addition to supporting the creation of such systems, we intend to answer the previous research question. That is, we wish to determine if such an API can be built in a way such that it is usable for developers.

Chapter Two: Multi-Surface Systems and APIs

2.1 Multi-surface Systems – Benefits & Overview

An MSS allows users to take better advantage of the properties of its component devices. It allows users private space through the use of devices whose affordances – such as a small size or mobility – allow for work to be done discretely. It allows for the creation of public information radiators by incorporating large and highly visible devices. It also allows for the creation of specific and novel interactions which inherently involve the tight integration of several devices. These benefits are discussed in detail in the following sections.

2.1.1 Effective Use of Device Properties

One of the principle advantages of an MSS is that it can be made up of different types of devices, each of which can have properties that are useful for certain tasks. In a traditional application, a designer would have to compromise and choose the most appropriate device for the majority of tasks but which would not be ideal for other tasks. In an MSS, a designer can simply use the device best suited for whatever task or subtask is required. This added flexibility allows designers to build systems where users can switch between devices depending on the tasks and the properties required.

Selecting the device with properties which best support a subtask is often a source of tension for designers. Because a traditional application only runs on a specific single device, a single device must be chosen even if it's non-optimal for some of the subtasks, preventing designers from building an optimal application. In an MSS, designers do not have this constraint, so a different device can be used for a separate subtask. This allows designers to choose a set of devices which are most appropriate for all the subtasks in an application.

Consider a system for presenting three dimensional models to a group. The most optimal device for displaying the models would be a large format, high resolution display placed at the front of a room. Controlling and exploring the model with such a display might, however, be awkward for the presenter. In an MSS, the designers could move the subtask of controlling the model to a more appropriate device such as a tablet while still maintaining the effectiveness of the large display for the presentation task. As an MSS can incorporate various devices; it's possible for system designers to assign devices which are more convenient for a specific task to the device which is most appropriate for that task.

2.1.2 Public & Private Space

Consider a user who is in the process of working out an incomplete idea or has only partially completed a solution to a particular problem. This user might prefer not to have this partial work accessible to other members of his team or available to a wider group. This desire might depend on a variety of factors but designers need to consider how the usability of their system would be affected by users who feel the lack of private space to work. A device, therefore, can provide a private space if a user can have a reasonable feeling of privacy while working on it.

Alternatively, situations occur where users would prefer to have information available to their co-workers and group members. In such a situation it would be useful for users to be able to present information in a public space. This might occur in a task where some shared information needs to be available to different users, for example, a shared background map for some analysis task.

In a traditional system, where applications run on devices which are isolated from one another, it would be difficult to choose a device which could provide both a public and a private space. A designer would therefore have to emphasize one type of space and this could cause usability

issues when users really desired the opposite type of space, such when doing preliminary work on a public display (if the public aspect was emphasize) or find it inconvenient to share information on a private laptop or tablet (if the private aspect was emphasized).

An MSS can resolve this difficulty by dividing an application over devices which are appropriate for private work (e.g. a tablet) and those with public visibility (e.g. a large display). Several systems have demonstrated this technique in which balance public and private space by using multiple devices. In one such system, users could browse web content using a PDA (which provided private space) and dispatch content to a shared wall display (which provided public space) [1]. Another application area where private and public spaces are both needed is in card games. In such games users must keep some cards private while others are made public, such as in the game Rummy. Researchers created a digital version of Rummy where users have an iPhone which displays their private cards and while an iPad is placed in the center of users to show “runs” and placed down cards [2].

These examples illustrate that applications sometimes require a private space for certain tasks and a public display for others. It is difficult to reconcile these two goals in any system which must run on a single device. With an MSS, a system designer can simply use the device most appropriate to the task needed by users.

2.1.3 Improved Collaboration

In systems where multiple users will be using the system together, one of the goals of designers is to support collaboration among those users. Users collaborate in different ways and, and the use of an MSS can impact collaboration. The integration between devices in an MSS can support collaboration by improving communication, awareness, and coordination among users while they completed a specific task [3]. An MSS assists users by incorporating devices which could

improve awareness – such as a highly visible wall display. It supports communication by making it straightforward to share artifacts among group members, and it supports coordination by providing workspaces on which multiple group members complete and can integrate work products.

One study has shown how an MDE, which involves passive displays as well as interactive devices, can impact communication, awareness and coordination [3]. In the study, participants used an MDE to build a schedule and each user was provided with a private tablet and could view a public display. It was found that using an MDE gave users a more sheltered interface which is less visually distracting. This shows how device choice impacts awareness. In a longer term study focusing on software development, researchers found that developers used an MDE in an opportunistic way to quickly share a specific problem or feature they were working on with their wider group of co-workers [4]. In the study users could quickly replicate a view from their own workstation to a larger shared display. The availability of a large display, which was well integrated with their existing system, directly supported communication between team members. This task is specifically interesting because collaboration was done opportunistically, that is, quickly transitioning between a collaborative task done with other users and then back towards an individual task.

Another study extended these results to an MSS, considering what collaborative benefits would exist in a system composed of a multi-touch tabletop and personal tablets for each user [5]. The researchers found that when using such a system, the users fell into distinct strategies ranging from team-up (where the tablets were used to compare aspects of documents related to the task while the team members worked together) to divide-and-combine (where team members worked separately and in parallel on some task and then combined their progress afterwards). The

authors suggest that an MSS gives flexibility for a wide variety of collaborative strategies. These research studies suggest that an MSS can improve collaboration for software development and schedule making tasks. The MSS allowed designers to choose devices, such as a large format display, which improved communication and awareness and which in turn supported a variety of collaboration strategies which could not necessarily be supported by a traditional system.

2.1.4 Novel Interactions

The benefits discussed in the previous sections all improve usability by matching the appropriate device to the appropriate task. But an MSS can improve usability in other ways: it can create entirely new interactions by using devices in novel and interesting ways. Since these interactions involve several closely integrated devices, they could not be replicated by traditional applications running on isolated devices. We discuss two examples of such systems from our own research in the following sections.

2.1.4.1 Seismic Slicing

In oil and gas exploration, a common task for geophysicists is to analyze the area below the earth's surface to determine if the area may contain a reservoir. Analyzing this area is typically done by multiple different users with different backgrounds and skills. The analysis is typically supported by a visualization of the subsurface called seismic. Seismic is often organized into a three dimensional volume, the volume contains information about the structure of the subsurface. Visualizing this data is often done using a traditional application on a single desktop computer. A user selects a specific plane of the data (called a cutting plane), to reduce the three dimension volume to a two dimensional structure, which is more amenable to analysis. This process can be difficult for users who are not experts with the software and the process is often limited by the awkwardness of using a desktop computer with a large group of users.

An MSS application for visualizing this seismic data addresses these issues by making the visualization task more intuitive and natural. In this application the area of interest is presented using a map of the surface displayed on a digital tabletop. To explore the subsurface, a user manipulates a tablet by placing it vertically on the tabletop. When the tablet is placed down on the tabletop a slice of the seismic data, corresponding to the area under the tablet, is generated and then displayed on the tablet itself (see Figure 1) [6]. As a user rotates and manipulates the iPad, new slices are created and displayed. With this system, users can navigate the entire three dimensional structure in a natural way wherein the slice is presented in its original vertical dimension. In this system, the tablet device becomes a kind of physical slicer which a user can manipulate to explore the subsurface.

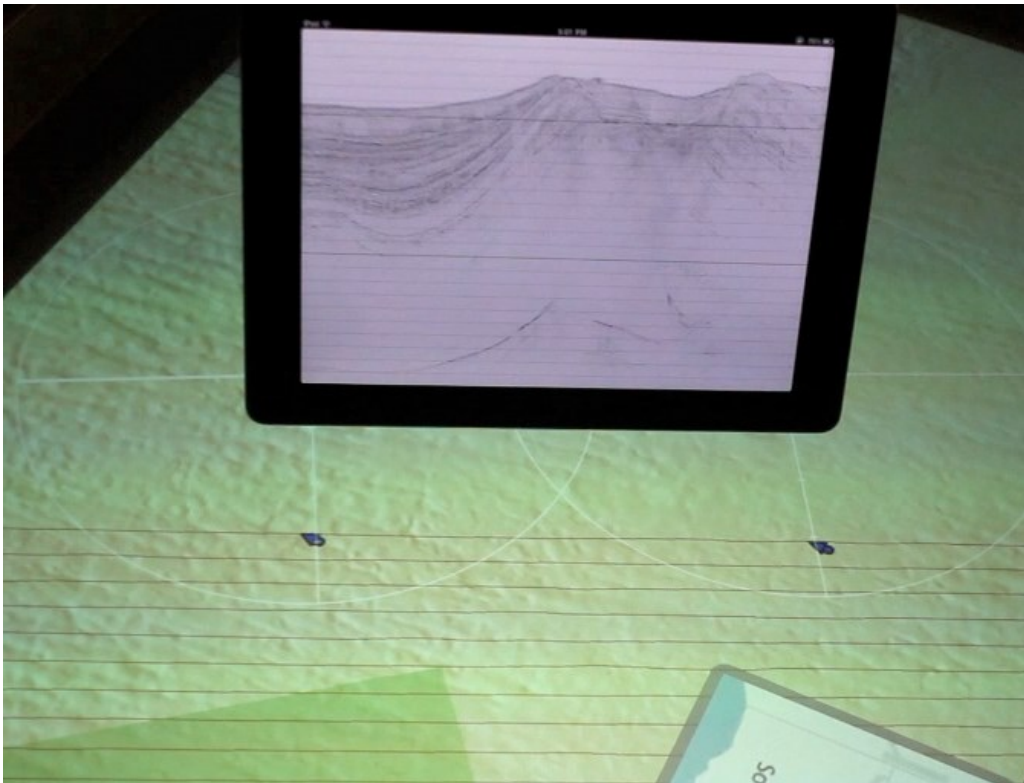


Figure 1: iPad Seismic Slicing

This interaction improves on existing techniques of seismic navigation, because it allows users to navigate the subsurface in an intuitive way. Because it incorporates a digital tabletop, the exploration is visible to a wide group of people who can work around the tabletop. In this way the addition of an MSS can create an entirely new interaction, which is novel, and comes as a direct consequence of the paradigm of dividing an application between multiple devices.

2.1.4.2 MRI Kinect

Another system where an interesting and novel interaction is made possible by an MSS is the *MRI Kinect* application [7]. Medical volumetric data is a three dimensional model of the body, derived from medical imaging techniques. Visualizing this data is an important practice in medicine. Typically this exploration is done using traditional applications isolated to a single device. Users explore the medical volume by selecting planes to isolate a specific two-dimensional slice from the overall three-dimensional model. While this task is often done by imaging experts, it can also be done by students studying anatomy or by clinicians preparing for surgery.

An MSS application for exploring volumetric imaging data provides a more intuitive interaction. A tabletop display is used to provide a picture of a human body. This body acts as a reference to assist users in selecting a slice in their particular area of interest. To select a particular slice from the volumetric data, a user simply moves their tablet over the part of the body they wish to create a slice in. The appropriate slice is then created based on the position of the tablet and the two dimensional slice is displayed to the user (see Figure 2). As the user moves the iPad up and down the reference body, a slice corresponding to the appropriate area of the body is displayed on the tablet. This technique is interesting because it gives the user a sense of exploring, in

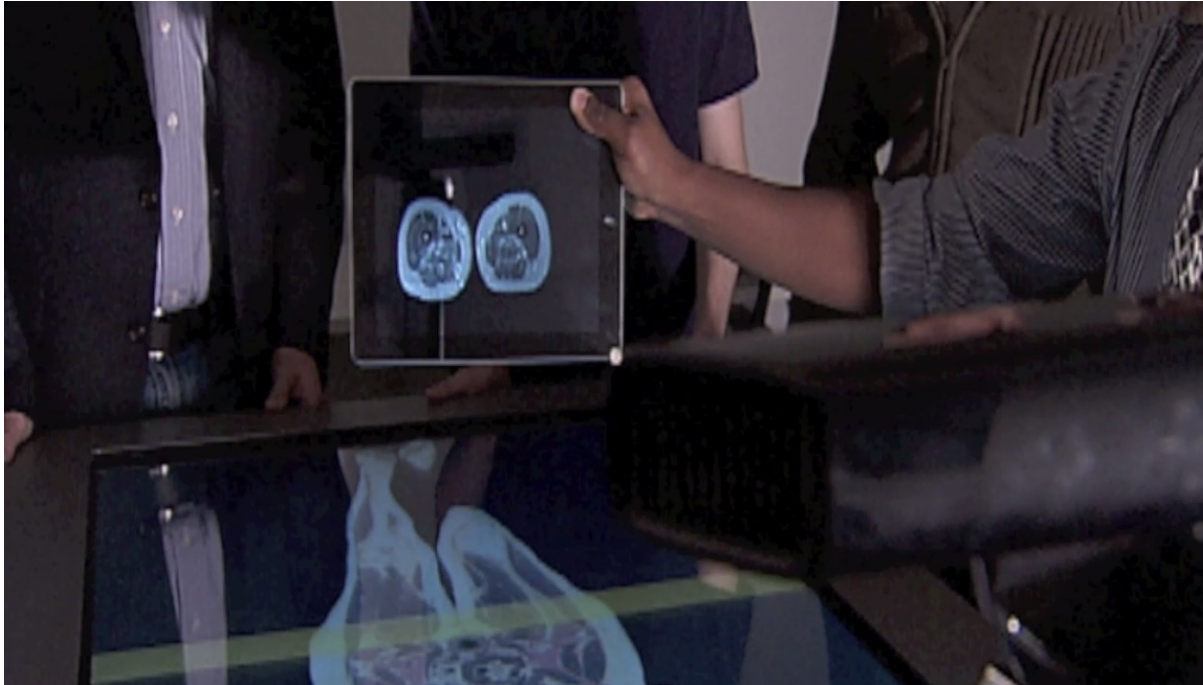


Figure 2: MRI Table Kinect Slicing

physical space, the three-dimensional model of the human body. The interaction technique is novel and takes advantage of several different devices, integrated into single application.

2.1.5 Summary of Benefits

As more devices become available to users, new applications and interactions will become possible. The multi-surface paradigm allows designers to match devices of different sizes, displays, and input mechanisms to the tasks which are most appropriate to them. This provides several benefits, such as the ability to take advantage of specific properties of different devices for increased usability, the distinction between public and private space, and the improvement to collaboration that such systems provide and several examples of entirely new and novel interactions which couldn't exist in the single device paradigm.

Creating such applications, however, will engender additional costs: increased programmer effort will be required to deal with distributed interfaces, synchronization, and other tasks. Since some

of the tasks are common to a wide variety of MDEs and MSSs, it's possible that tool support could be created to assist developers. The core tasks of such systems are discussed in the next section.

2.2 Infrastructure to Support Multi-surface Systems – Content and Control Transfer

While an MSS provides a variety of benefits, such systems are not commonly found either in consumer applications or in industry. We propose that this lack of support in industry is due to the additional complexity that is involved in building an MSS. This complexity is translated into additional development costs which make an MSS expensive to build and maintain. Before tools can be developed to mitigate these additional costs, a clear understanding of the core common problems related to building an MSS must be developed. Based on a review of several MSS applications (see Chapter 2) and experience in developing prototype multi-surface systems, we identified these core problems to be *control* and *content* transfer and the details of both are discussed in the following sections.

2.2.1 Control & Content Transfer

The first obvious difference between an MSS and a traditional single device application is the presence of additional devices. Creating an application which spans all these different devices presents several difficulties compared with an application which must run on only a single device. An MSS application requires developers to rethink how interfaces are designed, where computation is performed, where data is stored, which interaction techniques to keep from traditional applications and which to ignore etc. But while these problems might be solved in different ways depending on the specific application, two problems specifically stand out as being common to nearly every MSS. These common problems are the content and control

transfer tasks. These are both central to an MSS because they pertain to how an application is spanned across multiple devices.

These problems are broader than simply a networking issue. They have several separate dimensions depending on the perspective of the people involved in the MSS. From a user's perspective, the tasks must have an interaction or interface associated with them, to (a) initiate control over another device or their own device and (b) initiate or accept content transfers from other devices. We then discuss the types of interactions which, research suggests, might be most usable. From a developer's perspective the tasks involve how they design their applications to be (a) controlled or commanded from other devices, and (b) how they can send and receive specific content from other devices. Before interactions are proposed for users or tool support is discussed for developers, we will review precisely what is meant by each task.

2.2.1.1 Content Transfer

In an MSS, it is possible that multiple users will be working simultaneously on different devices in the system. In fact, this support for collaboration is one of the major benefits of an MSS over a traditional software system. In such a situation it is likely that users will create or access different content – by which we mean digital artifacts like images, documents, etc. – on different devices. For example, in an MSS designed to visualize data related to oil and gas exploration, the content might be a three-dimensional model of subsurface information, while in an MSS designed to share web pages, the content might be a link to a website. As users work with this content they will often want to transfer it to other devices as part of the task they are trying to complete. This transfer might be to another mobile device in order to capture the data on a personal device, to take advantage of the mobility of the device. The transfer might send the content to the device of another user, to support collaborative work with that user. Or the transfer might to a large wall

display in order to take advantage of the displays size. Content transfer can also have two directions: a user can send content from their device to another or they can retrieve content from another device to their own device. From this we can see the content transfer is an integral task to several desirable features of an MSS, such as making effective use of devices and collaborating with other users.

Without the ability to move content – which is necessary to complete tasks – the full usefulness of having multiple devices in the system is not realized. Content transfer must also be different in an MSS from content transfer systems which merely synchronize data automatically. Triggering content transfer, when used as part of a MSS to complete a task, should be a conscious decision made by a user. This is especially so in an MSS where multiple users will bring in personal devices as well as interact with public devices. To automatically synchronize all this data would create issues around privacy and control.

While users and not developers will actually perform the content transfer, supporting content transfer in an application can be made easier. Tools can be provided to support adding this feature to an MSS.

2.2.1.2 Control Transfer

Similar to transferring content, a user might want to control other devices in an MSS. By control transfer we mean remote application control – the control of one device by another device . This is important in several scenarios such as controlling a device with an inconvenient input mechanism or controlling applications on the device of a user who you are collaborating on a task with.

The ability to pass commands between devices could be useful whenever a device's available interaction modes are less optimal than those available on another alternative device. Consider a

large wall display which – because of its large size – might be difficult to use with a mouse and keyboard. A user might want to send instructions or command from their tablet (such as move up or move down) rather than control the interface with the keyboard. In this way control is passed from one device to another.

Consider an application which normally runs on a mobile phone. When the user is working within the MSS he would like to be able to use a larger interface such as a digital tabletop. When the user places his phone down on the tabletop, control is moved from the phone to the tabletop. Interface and other controls on the tabletop could dispatch commands to the phone even while computation and storage remained on the phone itself. .

The transfer of control is an important task in an MSS because it allows users to take full advantage of their devices. Previous research has developed functionality which allows control to be passed using screen replication approaches, but this has difficulties if the input mechanism, screen size or resolution is different between devices. To support this task we believe that commands should be passed between devices. Developers should write software to respond to the commands in the manner which is most appropriate for the specific application keeping in mind issues of security, privacy, and social conventions.

2.2.2 Spatially Augmented Gestures

In the previous sections, the content and control transfer task were presented as the core tasks associated with an MSS. Developers could provide many different types of interactions and interfaces to accomplish these tasks. But previous research has established that interaction techniques which incorporate and leverage the spatial layout of component devices of an MDE are more usable for the task of selecting devices [8]. This is important because selection of a target device is part of both the content and control transfer tasks which have been identified as

important for an MSS. in the following, spatial gestures and the implications for implementation of an MSS are discussed.

Studies which elicited gestures for content and control transfer tasks have identified a set of gestures which users would like to use for content and control transfer [9]. In these studies, a wide variety of gestural interactions were performed, but all incorporated spatial position (see Figure 3). For example, to send a picture from an iPad to a wall display, users proposed a gesture where they performed a flick on their iPad while it was facing the targeted wall display. From this research we can conclude that these gestures could be an important and usable interaction for accomplishing both the content and control transfer tasks.

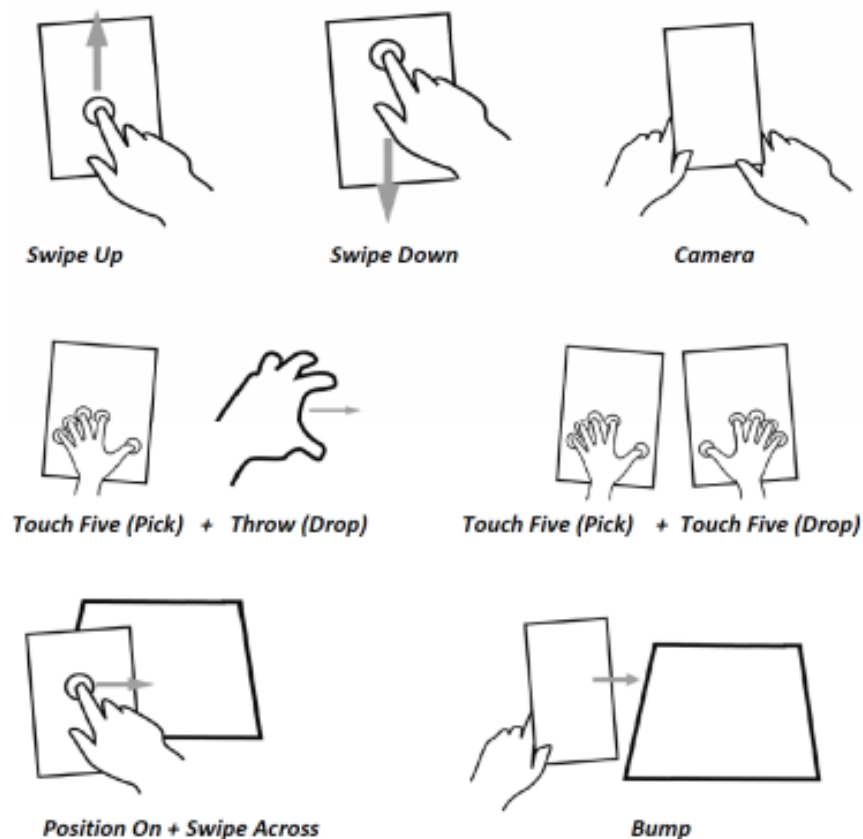


Figure 3: Proposed Gestures from Elicitation Study [9]

We define this types of gestures as spatially augmented gestures, these are gestures which incorporate the spatial layout of the system itself. Supporting spatially augmented gestures as a mechanism of interfaces requires even larger development effort than a standard MSS alone. In order to implement gestural interactions for content and control transfer, a developer would have to maintain location information for every device in the system including mobile devices. Since gestural interactions could dramatically improve the usability of an MSS it is necessary to support developers in providing gestural interactions which incorporate spatial elements.

2.3 An API for Multi-Surface Systems

In Section 1.1, we have reviewed the benefits which an MSS can provide over more traditional single device applications. Two general tasks – which every MSS must solve – were also highlighted: transfer of control and content. Such tasks should be accomplished, we believe, by providing gestural interactions which incorporate spatial information such as location and orientation information. Implementing such a system, however, would require a developer with a wide variety of skills a great deal of development effort. For example, to implement content transfer a developer would need to have an understanding of networking protocols and data encoding. Worse still, the developer would need to be able to implement this functionality across a variety of platforms in order to support different devices (e.g. tabletop displays, small and medium sized tablets, laptops, etc.). Multiplatform support is necessary because not every type of device necessarily exists on the same platform and further still we would like to support a wide range of devices within a specific type. However, the tasks being described are not specific to a particular application but are common to every MSS and could be solved in a reusable way. This approach could then be incorporated by other developers into their applications and a substantial amount of developer work would be avoided. The API could then reduce the

developer effort required to build an MSS and potentially increase their prevalence. In this section the high level requirements of the API will be discussed.

2.3.1 Communication

In order to implement content and control transfer it is necessary for the proposed API to assist developers in supporting communication between devices. This communication must be simple for developers to setup but at the same time extensible. Communication in an MSS must include device discovery so that devices can find each other as well as message passing between devices. Before communication can be made between devices, it is necessary for the devices to be visible to one another. It must be possible for developers to programmatically discover devices which are in the environment. Practically this is implemented by scanning for devices on a common shared network. This is as a reasonable assumption as an MSS is expected to be contained within a room.. Device discovery should, therefore, be a feature provided by the API. This will allow developers to build an MSS in a flexible way, without having to hard-code specific names or device types.

Once a device has been identified, it should be possible for developers to send a message to it in a straightforward way. Likewise, it should be simple for developers to setup methods which are called when a specific message is received. Many different formats for exchanging messages in a system are available, but a protocol should be chosen which is familiar to developers. The API should specifically choose a protocol because, once chosen, it would allow for tighter integration with the other features provided by the API.

As content transfer is an important task, methods for sending specific types of data should be provided directly by the API Common data types might include images, dictionary and binary files. Developers should also be able to specify code to respond to these messages. Additional

metadata, such as the device that sent the message and what type of interaction was used to trigger it, should be provided to the receiving application when a message is received.

2.3.2 Spatial Tracking

While communication alone would allow a developer to construct a basic MSS, research suggests that spatial information can inform the design of these systems and improve their usability [8]. This task can also be handled by the API so that developers do not need to implement this functionality independently.

The gestures proposed by the elicitation study (see Figure 3) all require location and orientation information to function properly. Consider a *swipe-up* gesture performed by a user to transfer an image from a tablet to another tablet held by a user facing them (see Figure 4). In this gesture the user is making a selection of which device they wish to send the image to by the orientation of their device. If the user had wanted to transfer the image to another device, for example a wall display, they would have pointed the device in that direction. In order to programmatically determine which device a user is facing, it is necessary to know three pieces of proxemic or spatial information, the location and orientation of the user's device and the location of the target device. Without this information a selection cannot be accurately made.

To address this, the API should keep track of the positions of all the devices which make up the MSS. This includes the positions of the fixed devices such as a wall display or tabletop. Some mechanism should also be provided for users and administrators to alter these positions in the event that the layout of the room changes. Because devices in an MSS could be mobile (such as phones or tablets) it will be necessary for some sensor to track the location of these devices and add this information to the spatial layout of the room. To improve the usability of the system, the

API should also provide developers with a visual layout of the room which can they can incorporate into the interface of their application.

It is also necessary that the API expose this spatial layout so that it can be queried by developers. These queries should allow developers to quickly determine which devices in the room the user is facing (to support some of the gestures found in previous elicitation studies [9]) and queries based on which devices are within a specified range. Methods for querying the server should be provided on platforms common to typical devices (such as tabletops and tablets) specifically .NET and iOS , but the server (where this information is stored) should expose its information in manner that is consistent with web standards.

2.3.3 Summary of Requirements

In this thesis we propose the creation of an API to support developers in the creation of multi-surface systems. Consistent with research that has found that spatial information provides an interface which is potentially better than standard GUI based approaches, we have proposed that the API should support the use of gestural interactions as an interface for accomplishing those tasks.

The API must provide straightforward communication between devices, allowing developers to discover devices, send messages, and define behavior when messages have been received. To enable spatial tracking, the API must provide location and orientation information for devices in the MSS. These requirements define the features which the API must have in order to be effective in helping developers to build an MSS. In the next section the usability of this API is discussed.

2.4 Usability of the API

Aside from meeting the requirements discussed in the previous section, what else is necessary for the API? One important aspect is how usable the API is for developers. Just as the usability of an interface can be evaluated from the perspective of a user, the usability of an API can be evaluated from the perspective of a developer. This is an important distinction because it forces API designers to consider the experience that developers will have when working with an API. In assisting developers in building an MSS it is desirable that the API that is created is usable. A more usable API widens the set of developers which is capable of using it and reduces the frustrations and difficulties experienced by those developers. This in turn might make the development of multi-surface systems more prevalent. In this section the definition and criteria of usability which will be applied to the API will be discussed.

2.4.1 API Usability

In an early article on API Usability, Steven Pemberton asked API designers to “imagine hypothetically, just for a moment, that programmers are humans” and “that their chief method of communicating and interacting with computers was with programming languages” [10]. This concern for developers using an API leads the authors to describe several aspects of API usability – such as learnability, efficiency, memorability and misconceptions generated from the API. They also make a close connection between the usability of an API and its documentation. This is an important consideration which later studies could validate. In his canonical work on API Usability, Jeffrey Stylos makes a clear distinction between the *usability* of an API and the *power* of an API [11]. He further describes aspects of usability to include productivity, error prevention, simplicity, consistency and conceptual integrity. These aspects mirror the initial

usability concepts proposed for user interfaces by Jacob Nielsen which are learnability, efficiency, memorability, errors, and satisfaction [12].

On the other hand, it must also be recognized that optimizing certain usability aspects might have a negative impact on other aspects or on other goals such as extensibility. Because of this, API designers must make a trade-off between certain usability aspects. For our API, we have decided to optimized learnability and discoverability while still maintaining efficiency. This is because developers building an API must be aware of a variety of specific domains within software development, such as networking, distributed systems and interactions with sensors. It would be ambitious to assume that developers wishing to build an MSS will be experts in all these areas. The entire set of usability goals for the API is outlined in the following sections.

2.4.1.1 Learnability & Discoverability

For an API to usable it must also be straightforward for a developer to learn and discover how to use it properly. Learnability is a measure of how easy it is for a developer to learn the features and functionality provided by an API. This aspect can be affected by design issues. For example, if a class was poorly named, it would be difficult for developers to discover its purpose and understand how to accomplish tasks which involve that class. Likewise, poor documentation could impact learnability because too few examples were provided or poor explanations of classes were given.

To assess the learnability of an API, researchers conduct user studies [13]. In these studies developers (who don't have previous experience with the API) are asked to perform a specific task. Researchers then track whether or not the task was completed and what parts of the API confused developers. We conduct this evaluation with a case study, which involves use over a

longer period of time. This provides a more realistic picture of the learnability and the discoverability of our API.

2.4.1.2 Efficiency

Another important aspect of usability is how productive a developer is while using the API. By productivity we mean how quickly a developer can accomplish the useful work that they desire using the API [12]. An API might provide a large number of features but if it is time consuming for a developer to complete a specific task, then it is less usable. To measure the productivity of developers using our API a case study will be conducted with experienced developers. During the study period, we will measure how long it takes developers to complete tasks.

2.4.1.3 Satisfaction

A final factor in the usability of the API is how much developers enjoy using it. This satisfaction might be considered for the whole API or to some of its parts. While this aspect of usability is qualitative, it is still important. It can indicate to designers if using the API is tedious or if they have made a conceptual mismatch between the API and the mental model of developers. Based on this indication, designers can follow up their inquiries to find the ultimate source of the usability issues. Assessment of satisfaction should be performed with developers who have had experience using the API over a longer period of time. Using our API, several developers will attempt to build a multi-surface application during a period of several weeks. After each the completion of the project, the developers will be asked to comment on their experiences using the API.

Chapter Three: Related Work

As my work involves the construction and evaluation of an API for supporting multi-surface systems augmented with gestural interactions, three general types of previous work must be reviewed. First, research into interactions in multi-surface systems, specifically what interactions researchers have proposed to solve the problem of (a) content transfer and (b) control transfer. A summary of these interaction techniques, along with canonical examples, is provided in Table 1. Second, existing APIs which relate to multi-surface systems – such as those providing device communication and proximity information – are discussed in detail. Finally, work related to the usability of APIs is reviewed, specifically the approaches used for evaluating an API and the recommendations provided to API designers. These areas are presented respectively in Section 3.1, Section 3.2 and Section 3.3.

3.1 Interaction Techniques

Since an MSS or MDE is composed of several distinct devices, the core design challenge for such systems is how to divide a single application across these displays. One paper specifies the problem as how to create “interaction [which] spans input and output devices and can be performed by several users simultaneously” [14]. Designing such interactions, which span across multiple devices, is a difficult problem. Designers have narrowed their focus to provide solutions for two tasks (a) transferring content – documents, images, etc. – between devices of the system and (b) transferring control among devices. Numerous interactions have been proposed for addressing these tasks during more than thirty years of research into MDEs. Since it would not be feasible to review each of these systems, interaction categories are illustrated by specific canonical examples. The general categories are as follows: a graphical interface (Section 3.1.1),

the use of proxemic and physical components (Section 3.1.2), and finally the use of gestural interactions (Section 3.1.3).

3.1.1 Graphical Interfaces

Designing an interface for an MDE using the traditional GUI paradigm is the approach most likely to be consistent with interfaces that users have previous experience with. These were among the first approaches described for MDEs. Using such an approach, different devices can be interacted with via a menu or list (Section 3.1.1.1) or the menu can be organized to include a spatial layout or design of the room (Section 3.1.1.2).

3.1.1.1 Menu Based

A menu-based approach is a straightforward way to support both content transfer and control transfer in an MDE. Users can select which device they want to interact with from a list, where the device is represented by a name or icon. In one system, user laptops and public displays composed the devices in an MDE. Users could create a *binding* between their laptop and one of the displays co-located in the room [15]. A user initiated this binding to a display by selecting it from a contextual menu or chose it from a list of icons. Once the binding was created, a user could control the connected display. Menu based systems have also been created for an MSS and address the issue of content transfer. In another system, a user could select a file or digital object and drag it to an area or icon on the screen representing the other devices (see Figure 4) [16]. These menu-based approaches stick closely to the interface patterns users employ in other GUI-based applications and would be familiar to most users. However, since each of the devices in the system must be represented by a name or icon, distinguishing easily between a large number of devices or between different types of devices might become difficult for users.



Figure 4: Menu Based Approach for Device Selection. [16]

3.1.1.2 World in Miniature

To allow users to distinguish devices without the use of names and icons, some proposed interactions have incorporated the spatial layout of a system into the interface. Usually this is done in the form of a small map describing the layout of the system called a world in miniature. The devices in an MSS have a spatial arrangement and users are familiar with this arrangement from working within the system. Incorporating that layout into the system has been shown to improve the ability of users to identify and select their desired device (see Figure 5) [8]. One system with this interface presents the spatial layout in a simple diagram, users can select specific commands to execute on those devices [17]. The spatial layout can be augmented to show the applications currently running on each of the component devices [18] or with some type of iconic design to help users identify it [19]. Often in these systems, after a user has

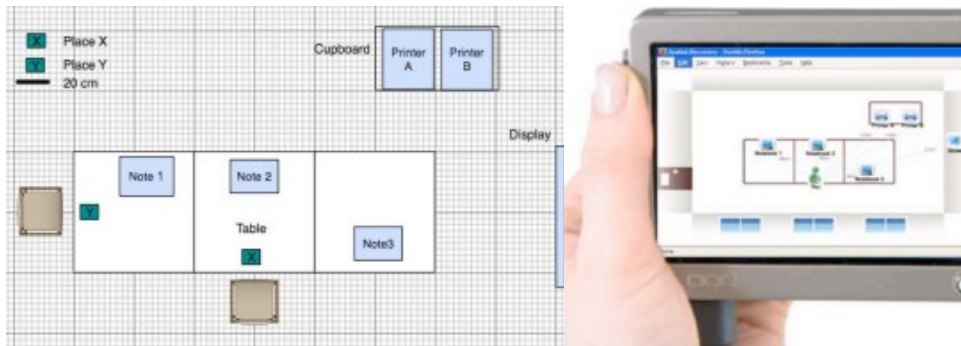


Figure 5: World in Miniature Approach for Device Selection. [8]

selected the component device they wish to use, they can control that device using a keyboard and mouse.

3.1.2 Physical & Proxemic Approaches

Extending the concept of using a spatial layout as part of the interface, designers have proposed interactions that incorporate the physical room and its spatial relationships. The physical parts of the system – fixed devices, mobile devices and people – are brought into to interactions for transferring content and control.

3.1.2.1 Docking

Outside of interface design, contact is an important factor in human communication. This factor can be mimicked in interaction design, so that contact (or docking) of two devices can be used as a trigger to accomplish tasks in an MSS. To implement this interaction, however, it is necessary to consistently detect when two objects are in contact. This poses some technical difficulties, such as computing the location of two devices, determining whether they are touching, or tracking their location; implementing this last part is often technically difficult and time consuming. However, contact can also be inferred using proxy measures, for example when a user performs an action simultaneously on two devices. Once the detection of contact is made, content and control transfer can be initiated.

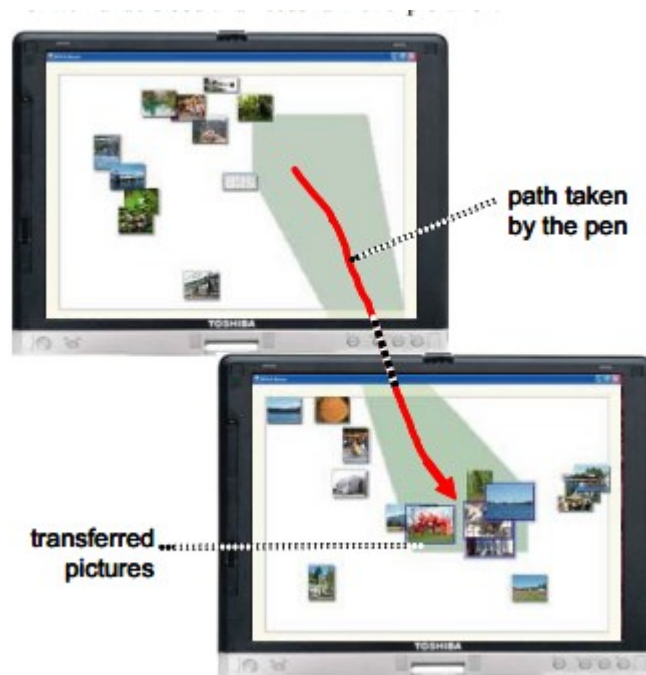


Figure 6: Stitching as a Method for Content Transfer. [20]

One method for detecting contact is for users to perform a continuous swipe on two touch devices which are placed side by side (see Figure 6) [20]. Another system detects contact when two devices are bumped together [21]. In these two systems once two devices are known to be physically in contact or docked together, they form a shared display where content can be dragged to either display. Docking has also been conceived as an interaction in systems equipped with medium-sized tabletops [22]. These devices are mobile enough to be physically moved across a room. When they are placed in such a way that the edges of their screens are together, a customized sensor system detects the contact and creates a continuous shared display.

3.1.2.2 Conduits

Users have experience moving and handling physical objects in their normal experience. System designers have proposed interactions based on this experience to accomplish content transfer.

These use the same actions that users are familiar with for transferring real world physical

objects. An early system allowed users to link a unique physical object to a data file [23]. The object was identified by its weight and when it was physically moved to another device the linked file was transferred to that device. In another early system a stylus could be used to select or “pick” some content item and then, on a different device, transfer or “drop” that content [24]. Users themselves can be used as the conduit for content transfer. This is typically accomplished by combining a MSS with a location tracking system that is able to track a user’s hand and arms. One interaction creates this conduit by having a user place one hand on a large display and then another down on a tabletop to create a “bridge” across these devices (see Figure 7) [25]. The content selected by the user in the first half of the interaction is then transferred from the wall display to the tabletop. In another interaction provided by the same system, a user selects content on a tabletop and carries that content to another tabletop. During this interaction the content itself is displayed constantly on the user’s hands, making it appear that the user is “carrying” the content across the room.

3.1.3 Gestural Approaches

The physical interactions described previously all provide interactions for the content and control transfer tasks. Since they mimic physical actions, they all inherently take advantage of the

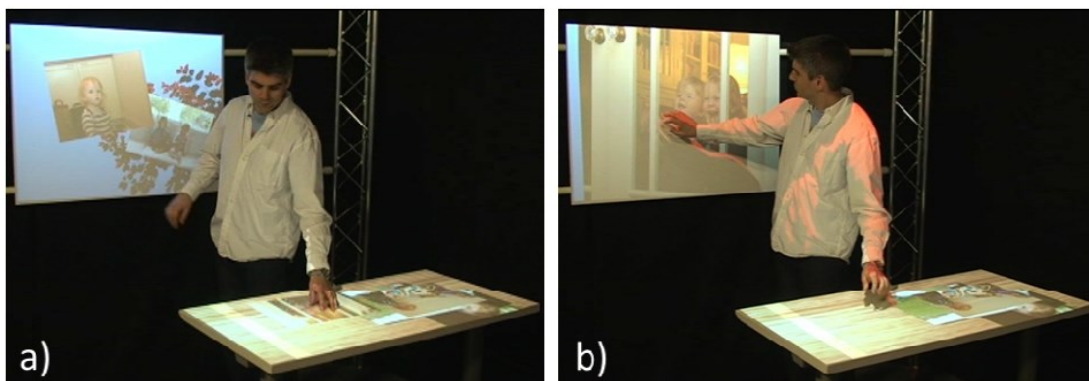


Figure 7: Physical Bridging as a Method for Content Transfer. [25]

physical layout of the room. But it's possible that some tasks would become tedious if constantly performed in this way; consider the effort involved to physically carry a series of pictures from one tabletop device to another. Gestural interactions also mimic physical interactions that users would be familiar with from experience and involve the spatial layout of a room, but may provide a more convenient interaction.

These interactions often involve spatial information, whether complete in the form of tracked locations of all users and devices, or partially in the form of orientation information. Gestural interactions often act as triggers to initiate control or content transfer. The direction of the gesture can be used to indicate selection (i.e. which device is the intended target) while the performed motion can be used to indicate the direction of transfer (i.e. whether the user wishes to send or retrieve). For example, if a user performs a *throw* gesture towards another user, the direction indicates they are selecting the user's device as their target and they intend to send content to that device.

3.1.3.1 With Device Gestures

Gestures can be performed by having a user physically manipulate the device itself. This is appropriate for mobile devices because they are lightweight and manoeuvrable.

Many mobile devices are now equipped with an accelerometer or gyroscope, so it is possible to get detailed information about the direction, intensity and structure of device motion. These gestures are defined as *with-device* gestures.

A *throw* gesture is one type of with-device gesture that mimics the action of throwing a physical object. This gesture has been proposed as a technique for initiating transfer between a mobile device and a wall display (see Figure 8) [26]. This gesture reappears in other systems where it is called the *chuck* gesture [27]. A different gesture for content transfer, which is appropriate for the physical layout of a digital tabletop, is called the *pour* gesture. It is performed by rotating a mobile device over top of a tabletop similar to pouring out a cup and then making contact with the tabletop [28].

In the above described gestures, the location of the user is not known, which means the gesture cannot be used to indicate selection. Without knowing the location of the originating device it's not possible to determine the device that the user is gesturing towards. But if these gestures were implemented in a system with location information, it could be used both to select the device that is the target and to initiate transfer to that device.

Gesture elicitation studies for MDEs and MSSs, which attempt to gather candidate gestures for use within an MDE, have elicited a wide variety of *with-device* gestures [9]. This provides evidence that these gestures could improve the usability of an MSS and provides support for these interactions.



Figure 8: Throwing Gesture as a Method of Content Transfer. [26]

3.1.3.2 Gestures & Proximity

Gestures performed on a device have gained wide usage as default parts of the user interface on most modern phones and tablets. These *on-device* gestures – which are performed on the device itself and – *body* gestures – which are performed with the user’s body, have been used in an MSS. These gestures are often coupled with proxemics information, which is defined by Ballendat et al. to include dimensions such as distance, orientation, movement, identify and location [29]. Using the orientation and location dimensions especially, gestures have been used to select and initiate content transfer in an MSS. These gestures are discussed in the following section.

Detailed work has considered how proximity can be incorporated into applications including how proxemic relationships can mediate device connectivity [29]. These ideas were later extended to applications where content transfer is triggered by gestures. In one system, a user selects an object of interest on their device and then performs an *upward swipe* gesture to initiate the transfer of that content from their device to the wall display (see Figure 9) [30]. Alternatively a user can send content to a display by selecting the desired location of interaction and then performing an *downward swipe* to transfer some content from their display to the device.

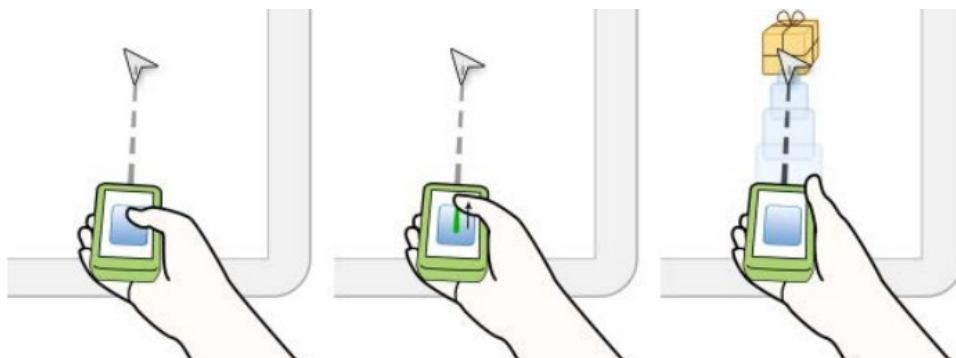


Figure 9: Flicking Gesture as a Method of Content Transfer. [30]

Selecting a digital item and then dragging the object along the screen in the direction of the intended recipient of the device is another gesture illustrated in another system [31].

Gestures can also be performed by users themselves; typically this is done by moving the hands or arms which are tracked by some motion capturing system. This type of interaction can be found early in the literature. One system proposed an interaction where a user could select some content using a voice command or selection gesture performed using a finger [14]. This *point* gesture could also be used to position the destination of the object. In this example the gesture was used for selection while the voice action provided the distinct trigger. In a more recent system a *point* gesture was used to select one digital object and direct it to another device by pointing at the targeted device (see Figure 10) [30].

Gestures augmented with proximity have been supported by several gesture elicitation studies. In these studies users have been asked to propose their own gestures for tasks. An elicitation study focused on content transfer between tablet devices and tabletops replicated the *flick* and *pour* gestures proposed by researchers and proposed several other gestures including a *pull* gesture performed on the tabletop [32]. Other elicitation studies, investigating content transfer in a complete MDE, have elicited numerous gestures performed on the device such as *swipe up* and

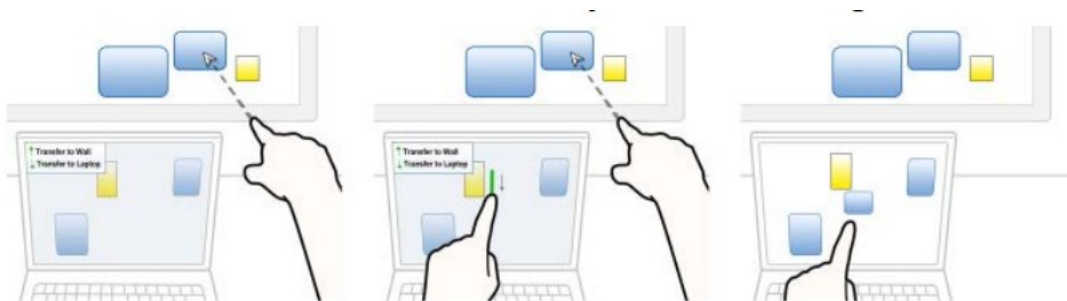


Figure 10: Pointing Gesture as a Method of Content Transfer. [30]

swipe down gestures as well as a wide variety of gestures performed in the air, such as *point* and *grab* [9].

3.1.4 Interactions Summary

A summary of the interactions presented in the preceding sections is presented in Table 1. This table describes the category of approach and briefly describes the gesture and interaction. Partial implementation implies that some additional API might be needed to complete the task (e.g. a gesture detection library to perform detection of gestures performed in the air).

Table 1: Summary of Interactions

Selection Mechanism Approach	System	Selection Details	Input	Output	Content or Control Transfer Details
Menu-Based	[15]	User selects the desired device from a list and	Laptop	Public Display	User could control the public display as an extension to their laptop
Menu-Based	[16]	User drags a window or file to an area reserved on the screen for device.	Laptop, PDA	Public Display, Public Tabletop	User can drag a window or file to another device.
World-In-Miniature	[17]	Selection is done using a menu presented as a spatial layout of the room.	Laptop	Public Display, Other Laptops	User can drag a file or digital object (e.g. URL) to any device which supports it
World-In-Miniature	[18]	Selection from a spatial layout augmented with updates on screen activity.	Laptops, PDA, Tablets	Wall Displays, Other Laptops	Users can control a remote display as an extended display. Can

					drag windows onto remote displays.
Docking	[20]	A continuous swipe across two screens, which have been placed next to each other, identifies the devices interacting and their relative orientations.	Pen Enabled Tablet	Pen Enabled Tablet	Users can drag files from one tablet to another.
Conduit	[23]	An object is bound to a unique file and triggers a dispatch when placed down.	Tabletop	Tabletop	Item is transferred between tabletops automatically.
Conduit	[24]	Users can select an item on one display and then touch down to drop it on another.	Wall Display, Tabletop	Wall Display, Tabletop	Item is transferred between devices automatically.
With-Device-Gesture	[26]	User transfers content to a wall display by performing a throwing gesture with a device.	Tablet, Mobile Phone	Wall Display	Item is transferred between devices automatically.
With-Device-Gesture	[28]	User transfers content to a tabletop by performing a pouring gesture with a device.	Tablet	Tabletop	Web page is transferred automatically.
On-Device-Gesture	[30]	User transfer content by performing a swipe up gesture on a mobile device.	Tablet, Mobileph one	Wall Display	Item is transferred automatically.
Person-Gesture	[30]	Users transfer content from the device to a wall display by pointing.	Tablet	Wall Display	Item is transferred automatically.

3.2 APIs Supporting Multi-surface Systems

Since no single toolkit or API provides all the functionality for adding gestural interactions to an MSS it is necessary to review several types of APIs which each supply some useful feature. As there are only a small number of APIs in this space they are reviewed individually and in detail.

Some APIs which provide support for device communication, such as message passing and device discovery, are reviewed in Section 3.2.1 while APIs which provide spatial or location information are presented in Section 3.2.2.

3.2.1 Device Communication

Several API which have been developed specifically to support communication between devices in a MDE or MSS will be reviewed in this section. These APIs provide two kinds of features, (a) device discovery – where all the devices currently active in the system are collected and made available, (b) message passing – where messages of some format are transmitted between devices and (c) application division – tools which supporting a single application divided over multiple devices. Typically APIs offer solutions that provide message passing or application division, but not both.

3.2.1.1 3MF

3MF is a framework developed to “expose rich device functionalities which are currently available only through local frameworks” [33]. It was developed by Kaufmann et al. around the goal of exposing as services many of the functionalities found on a mobile device, such as accelerometer or touch data. Such functionality can normally only be accessed through local APIs. Exposing this functionality would be useful for creating an application whose interface is distributed over several devices. For example, an interface displaying a map on a large wall display could be controlled by subscribing to orientation data provided by a tablet’s gyroscope. *3MF* is written to work in a peer-to-peer environment without any centralized server to route activities. The API is integrated into each application running through the MSS. Devices are discovered on a shared network using the Bonjour protocol. When a device is discovered its

capabilities (or provided services) are announced to other devices. These can then be consumed (or subscribed to).

3MF is an API that is designed to support shared access to specific services provided by various devices. Since these services are predetermined, developers would need to alter the API to provide additional services and no general method for exposing them, such as routes, is supported by 3MF. Since 3MF does not provide a direct mechanism for content exchange (i.e. the exchange of data types such as images and dictionaries) developers would also have to add this functionality themselves. Finally, the services provided by 3MF do not conform to a REST-ful interface which would allow other clients, without using a client library, to access their services using HTTP formatted messages. Support for REST-ful interfaces is an important feature as it allows for compatibility with any device which is able to generate HTTP messages, this improves the extensibility of the API. It also allows testing and debugging to be done in a straightforward way (e.g. testing can even be done using a web browser).

3.2.1.2 Event Heap

Event Heap is an API used to “coordinate the interactions of applications running on [...] devices that will be common in ubiquitous computing environments”. It was implemented by Johanson and Fox along with an MSS system called the iRoom [34]. The designers of the system initially assert that most realistic MSS will be utilizing traditional applications that will be coordinated together into an ensemble. Therefore, their API does not force users to change their tools and styles of development.

The API itself is written around a programming paradigm called *TupleSpace* where applications coordinate with each other by sharing access to a set of tuples (key value pairs) and where each application can read or write to this space as desired. Applications poll Event Heap and query

tuples which they are interested in. Client libraries exist which can work with Java, C++ and, through a proxying mechanism, web applications.

As Event Heap was designed over ten years ago, it preceded the widespread use of standard web based technologies such as REST-ful APIs. Since the designers envisioned that API to be used on wide range of devices (even projectors and mechanical controls) it might not be feasible to run a full HTTP server. The practice of constantly polling a shared space would create a large amount of communication overhead within an MSS composed of only a few devices. To support the content transfer task it would also be necessary for their API to explicitly deal with data (such as images, files, dictionaries, etc.) but the API explicitly leaves this task to another middleware layer called *DataHeap*.

3.2.1.3 MAGIC Broker

MAGIC Broker is an API which explicitly uses modern web technologies and REST-ful APIs [35]. It was designed by Erbad et al. specifically to support communication between mobile devices and a large public display, which is an additional constraint on the class of general MSS. The framework exposes functionality on the large display using a REST-ful API that mobile devices can then consume. In one system, built using their API, users could enter tags on their mobile devices and see pictures related to those tags (derived from Flickr) displayed on that public display.

Two components make up the design of MAGIC Broker, a broker server that provides the exposed API and an application server that communicates with the broker. As the system was developed to support mobile phones, the broker component will also accept information in the form of SMS and XML messages. The designers do not describe in detail how the application

server running on the public display would receive and process the messages coming from the broker.

As MAGIC Broker exposes the functionality of the interactive display using a REST-ful interface, it is possible for developers to query the server without the need to use a client based API. But because the mobile devices are not themselves exposed using a REST-ful API, it's not clear how bidirectional communication could be supported. This is important for applications with support for gestural interactions because it must be possible to send content from and to any device in the system. Finally, the developers do not describe any convenience methods for helping developers who are not familiar with networking to accomplish straightforward tasks, such as sending an image or dictionary between devices.

3.2.1.4 ROSS

ROSS is a toolkit which the designers describe as being a “way for applications to run across a variety of platforms and devices: tabletop computers, touch-screen mobile devices and responsive walls” [36].

To support such MSSs, the designers propose a novel nested structure to their API. To model the structure of the MSS, each object in the room is represented by an *RObject*, which itself can contain additional objects in a tree like structure. For example, an *RObject* might contain an *RSurface* which specifically represents an interactive surface, while the *RSurface* contains a mobile phone, a finger touch, and a puck – all of which are on the display and are themselves *RObjects*. Developers configure the initial structure using an XML configuration file.

Communication between the actual devices is handled through XML based message passing, specifically Open Sound Control messages.

ROSS is a novel approach to organizing a MSS because it allows developers to structure their application in a nested way with each node being an object in an MSS. However, such an approach does not substantially improve the previously described tasks in an MSS, the transfer of content and control. Messages in the system are still sent as XML and developers would need to send and respond to these manually. One difficulty with this framework is that it forces developers to change the tools and frameworks traditionally used in developing an application (such as UI frameworks and toolkits).

3.2.2 Proximity & Location

Many of the gestural interactions envisioned for a MSS (see 3.1.3) involve knowledge of proxemic dimensions (location, orientation, etc.). APIs which provide this type of proximity information, such as Proximity Toolkit, NearMe and SharedSubstance are reviewed.

3.2.2.1 Proximity Toolkit

Proximity Toolkit, developed by Marquardt et al., is a toolkit for building applications with proxemic interactions [31]. Research into proxemic interaction involves a larger scope than content and control transfer in an MSS. As such, their toolkit was designed to support a wide range of applications. For example, Proximity Toolkit was used to create an application where the interface was adjusted to accommodate for the distance between the user and the application. Because of the difference in focus, some architectural and design decisions are not necessarily optimal for the content and control transfer in an MSS.

Proximity Toolkit itself is composed of a centralized server that provides proxemic information to clients. This server is supplied with proxemic information from a sensor requiring markers being placed on objects; specialized modules convert the sensor data into a format

understandable to the main server. A visualizer provides a 3D visualization of the tracked items and specific libraries provide event-based updates to clients via a distributed data structure.

Proximity Toolkit is primarily used in conjunction with the VICON tracking system, which is able to track objects and people with a high degree of accuracy and range. Unfortunately the cost of such a system is prohibitive and not within the range of consumers. To function the sensor also requires users and devices to wear physical markers, this is acceptable for prototyped applications but not realistic in real world situations. While it is possible to use Proximity Toolkit with a Kinect sensor, which is more affordable, a developing building a MSS would still need to maintain a relationship between the devices and the users who are holding them so that the person's location can be used as a proxy value for the device. This would be a substantial investment of developer effort. Likewise they would also need to integrate data from other sensors, like a gyroscope, themselves.

Proximity Toolkit is also an event driven toolkit, meaning that developers can request updates for particular pieces of proxemic information or relationships. For example, a developer writing an application could request continual updates on the location of a specific person or whenever a user was facing towards a distinct direction. This is a good design choice for systems where rapid updates of proxemic information are used as the basis for specific actions (e.g. updating an interface). However, in supporting gestural interactions a developer wishes to query the spatial state of the room at a given time. For example, if a developer wishes to allow a user to direct a photo to all devices in his field of view, the developer would simply query which devices existed in the user's field of view at that time. Queries rather than events would be more convenient for this task. The server component in Proximity Toolkit provides updates to client APIs using a distributed data model based on the TCP protocol. This is effective given the event driven

approach mentioned above. When using queries, a more natural architecture style is to follow a REST approach as described by Fielding [37]. This architecture allows information known to the locator server to be queried using HTTP and is structured in a style that is consistent with APIs based on web technologies.

3.2.2.2 Easy Living

The Easy Living system is a framework developed at Microsoft Research by Brummit et al. and it attempts to support systems where there is a “dynamic aggregation of diverse I/O devices into a single coherent user experience” [38]. While this focus is larger in scope than an MSS, the API could be used to construct such a system. The system provides person tracking based on stereo cameras.

The system is built around the Easy Living Geometric Model or (EZLGM) which provides a geometric layout of devices in the room. Developers define each entity in this layout to have measurements for position and extent, as well as uncertainty values for those particular measurements. Developers can query this to determine the geometric relationships between entities and which entities fall within a certain radius of another entity. EZLGM captures this information using stereo cameras which are not described in detail.

Since Easy Living was developed before consumer level tracking systems were widely available it relies on stereoscopic cameras for positioning. Similar to Proximity Toolkit, the API provides information about people but cannot reliably track devices such as a tablet. Developers using Easy Living would need to directly implement a relationship between devices and people to query the locations of mobile devices.

Querying the system is done using SOAP, but since the creation of Easy Living this technique has been replaced by REST-ful interfaces, the designers do not mention if a client library to make these queries more convenient for developers.

3.2.2.3 NearMe Server

The *NearMe* project describes a server that provides proximity information gathered from wireless networking [39]. Usually such systems require calibration where signal strength is correlated to specific distances. In this server, users can accomplish this calibration by pairing this strength information to known locations themselves. Because this information is based on network signal strength it cannot provide location but only proximity. The system is intended for a larger range (about 30 – 100 m) and would not be accurate enough for positions within a room where an MSS might be located. This work shows an early example of a central server that provides proxemic information to clients.

3.2.2.4 Shared Substance

Shared Substance describes a middleware layer for supporting multi-surface systems. The API provides a set of tools for organizing applications across devices as well as sharing resources inside this application. Specifically, it provides shared access to a VICON tracking system which provides highly accurate location tracking.

The middleware follows the data-oriented programming paradigm, a rarely-used alternative to object oriented programming. This construct works at a lower level than other frameworks for sharing proximity information, developers create applications on a specific device by attaching it a tree representing the application.

It is not clear how applications built using this approach would work with other popular GUI toolkits for constructing applications such as Window Presentation Framework or Cocoa. The

developers do not describe their middleware as providing any interface or augmentation to the raw information provided by the VICON cameras. Developers would need to access this information directly from the cameras themselves.

3.3 API Usability

As one of the goals of MSE-API is to produce a usable API, it would be useful to here review some of the research work into API Usability. This research is the application of usability research – such as that typically done for user interfaces– to APIs.

In our methodology, developers that use the API are treated as users and the API itself like a system whose usability can be evaluated and improved. In Section 3.3.1 the common issues affecting API usability are discussed and in Section 3.3.2 the methodologies and methods used to evaluate an API are presented.

3.3.1 API Usability Issues

Several studies have attempted to identify the main problem categories that cause API usability problems. These issues were found to be (a) naming and conceptual issues, (b) design issues, and (c) documentation and supporting material issues. Studies identified these problem categories in different ways, such as usability studies on a particular API [40], through a survey given to a group of developers [41] and through manual analysis on newsgroup comments related to a specific framework [42]. Two studies attempted to categorize and summarize existing literature published on the field of API Usability [43,44]. And another study analyzed the bug reports associated with large open-source APIs [45]. Detailed discussion of these core problem categories are presented in the follow sections.

3.3.1.1 Naming & Concepts

Since developers primarily interact with an API through exposed functions, interfaces, and classes, the names that a designer chooses for these artifacts will have an impact on the *learnability* and *discoverability* of the API. Names in the API need to map well to the underlying concepts of the domain. One paper studied the relationship between internal-representation of domain concepts and the usability of an API and proposed metrics to help developers improve the relationship between these [46]. Another work considered the entire API as a communication artifact and investigates how specific choices, including the names of classes and interfaces, affect this communication [47]. Other work has discussed how excessively generic or abstract names lead to names with poor expressiveness which do not convey any information about the role and purpose of the artefact [48].

3.3.1.2 Design

The design of an API can influence its usability and studies have investigated how certain design patterns and styles impact the usability of an API. User studies found that a standard constructor was better for instantiating a class than a static method [49] and a factory method [50]. It was also found that requiring developers to provide parameters to a constructor was not as usable as allowing them to instantiate the class and then set the parameters [51]. APIs with a large number of classes require more searching time before developers can find the class they need, its recommended that important classes be presented separately from utility classes [49]. When considering the entire design space for an API, work has recommend separating the architectural design decisions from those related to language level decisions [52].

3.3.1.3 Documentation

The relationship between the usability of an API and its documentation has been subject to a great deal of research. A large-scale field study conducted at Microsoft, using interviews and surveys, found that developers felt that poor documentation and learning resources were the major cause of poor API usability [53]. A qualitative analysis of the data collected highlighted several common issues with documentation; these include issues with code examples, intent of documentation, penetrability, and the format used for presentation. Another study investigated the usefulness of API documentation for users missing domain knowledge related to the API [54]. They suggest that documentation should include background information for users without experience in the domain. Studies around API documentation repeatedly stress the importance of providing quality code examples [55,56]. But creating and maintaining a large set of code examples is expensive. Researchers have proposed methods to assist API designers with this task, evaluating the idea of using unit tests as examples [57] and synthesizing or suggesting the examples from open-source software in public repositories using the API [58,59]. Finally, researchers have proposed a series of tools for improving the exploration of API documentation [60,61].

3.3.2 Evaluation Strategies

Several methods have been proposed for evaluating a specific API. The most widely used is a user study, where a user is asked to complete a task with an API. The usability can then be assessed based on the experience of the developers and whether or not they completed the required tasks. Reviews of an API can be conducted where a developer “walks through” an API with an API designer and provides feedback similar to a design critique. Finally, several techniques attempt to evaluate an API by defining precise usability metrics.

3.3.2.1 User Studies

A relatively straightforward way to evaluate the usability of an API is to ask other developers to complete a task using the API, track the number of users who completed the task and what errors or difficulties they encountered, and measure how long it took developers to complete a task.

Typically the tasks chosen must be reasonable in size so that developers can complete them during the time appropriate for a user study. User studies must have tasks of this size as longitudinal studies are more expensive and time consuming to conduct. The results of user studies, therefore, focus on learnability and not the usability for long-term users. When attempting to assess the learnability of an API, it is necessary the developers participating in the study do not have previous experience using the API before.

One study applied this approach as a part of a study on the usability of service-oriented APIs [62]. During their study developers were required to complete a task using a specific real-world service oriented library. While they were completing this task they were asked to follow the “think-aloud” protocol by describing their actions and reasoning as they worked. The authors organized and classified the types of errors that users encountered while using the software and presented those that would be common to many service oriented APIs.

Another user study was conducted at SAP as part of an ongoing effort to redesign an API for creating and updating business rules [63]. The designers proposed a prototype API and then asked developers to perform three tasks (during a limited time) to assess its usability. During this process developers were also asked to use the “think aloud” protocol. The authors propose that this method, evaluating a prototype API using repeated reviews over several months, is an effective process for developing a usable API.

3.3.2.2 Review Processes

While a user study typically involves a small task which a developer performs, an API review is more detailed and does not necessarily involve the independent completion of a task. One kind of review, called an API peer review, has been proposed by researchers at Microsoft as a methodology for reviewing the usability of an API [64]. During this process a feature owner (i.e. the API designer) walks a group of developers through a specific code sample where the API is used to perform a concrete task. These developers provide feedback aimed specifically to improve the *learnability* and *discoverability* of the API. Elements that might affect those qualities, such as poor name choices and inadequate exposure of methods might be raised by the developers. The study proposing this method found that it compared favorably for finding usability “bugs” compared to the cognitive dimensions framework.

3.3.2.3 Measurement Methodologies

Another way to determine the usability of an API is to apply a set of metrics and attempt to “measure” the usability of the API. One set of metrics is called the cognitive dimensions of notations framework. Its authors describe it as “a broad-brush evaluation technique ... [which] sets out a small vocabulary of terms designed to capture the cognitively-relevant aspects of structure” [65]. Originally it was designed to highlight the aspects that affect the usability of an interface or programming language. Each of the dimensions can be stated in the form of a question to designers of the given system, for example, to evaluate the *consistency* dimension the question is asked “when some of the language has been learnt, how much of the rest can be inferred?”.

The cognitive dimensions approach was first applied to the development of APIs at Microsoft [66]. The authors decided to modify the original cognitive dimensions framework to make it

more appropriate to the task of APIs, adding new dimensions such as *API Viscosity* which measures “the barriers to change” which an API faces. Describing their process in more detail in a later work, the authors show how API designers must balance the dimensions based target develop for their API. [67].

The cognitive dimensions framework is a useful tool for thinking about the aspects that impact the usability of APIs. But since each dimension is qualitative, it’s difficult to aggregate the responses. Performing the evaluation over time (such as in a longitudinal case study) or in a group setting (such as during an API review) might improve the impact that the cognitive dimensions framework has.

Another methodology that has been applied to the evaluation of APIs is *concept maps* [68].

These maps are a directed graph where *concepts* related to some general field are represented as nodes and their relationships between concepts as a directed edge. This tool was originally proposed as an instructional tool for teaching science to children.

Researchers adapted this mechanism to help evaluate the usability of an API. While developing an application using a specific API, the developers on the project create a concept map relating aspects of API to different parts of their system. During each week developer spent a 30 – 60 min session (done once a week for five weeks) updating their map. By studying the changes made to these concept maps designers can determine the mental model of the developers and attempt to understand which areas of the API might be problematic. For example, if a relationship was repeatedly altered or if a relationship had to be changed even after weeks of development it might indicate that concepts involved are difficult for developers to properly understand.

This technique envisions that using an API typically involves a long term learning process that extends past the few hours usually allocated to a user study. It also gives designers a specific view of the mental model of their developer-users. However, analysis of the changing map would need to be done carefully so that confounding issues do not impact the usability analysis of the API. Likewise it is also more difficult to control the environment completely in a longitudinal study and confounding variables (e.g. developer training, staff changes, requirements changes, etc.) could impact the study.

Many of the methods so far proposed to measure the usability of an API have been highly qualitative. It's possible the different users will answer questions related to the *cognitive dimensions* differently or construct a *concept map* in different ways. Researchers have attempted to reduce some quality measurements to quantifiable metrics.

One work proposed a systematic approach for assessing the usability of software components [69]. In their procedure, they attempted to reduce concepts to specific attributes and then assign those attributes a derived measure. For example, to determine the “quality of documentation” they would calculate coverage metrics for the manual such as “the percentage of functional elements described in the manuals”. This procedure has construct validity issues because it's difficult to associate the quality present in the API to the specific metrics they suggest. For example, how can we know that the quality of documentation is proportional to the number of functional elements described in it? Further, the calculation of these metrics would also be prone to interpretation issues and would be a tedious task.

Other metrics based approaches have attempted to assess the complexity of an API on the inference that highly complex APIs would be less usable [70]. Using metrics designed to measure the cognitive complexity of code the authors compute the complexity of all the exposed

interfaces and classes provided by the API. The paper does not attempt to validate this method by comparing it to qualitative interpretations of complexity.

3.4 Conclusion

In this section, a set of interaction methods were discussed for implementing content and control transfer in an MSS. Some of these are gestural interactions that involve proxemics and take advantage of the spatial layout of the MSS. It is a central goal of MSE-API to support these types of interactions. Existing APIs which provide features that could be used to build these gestural interactions are then discussed. Finally, since the goal of MSE-API is to be a usable API, some results from the field of API usability were presented.

Chapter Four: MSE-API

In Section 1, the motivation for MSE-API was discussed. It was found that using a multi-surface approach provides several benefits for users. It allows users to take full advantage of their devices, supports collaboration, and allows for novel interactions. But the absence of such systems in industrial and consumer settings was also noted. It was speculated that this might be due to the increased developer effort required to build an MSS. This work proposed the creation of an API, called MSE-API (Multi-Surface Environment API), to support developers in this task and to concentrate especially on the control and content transfer tasks. The API was intended to be usable for developers. In this section we will review the requirements for the API in detail, discuss the structure of the API, and compare MSE-API with other APIs in this space.

4.1 Requirements

The requirements for MSE-API can be divided into three general types: constraints which are imposed by the practical needs of its intended users; the functional requirements which include the major functionality the API needs to provide; and the non-functional qualities which the API needs to have.

4.1.1 Constraints

MSE-API was designed in order to be used. This means that the API must be accessible to the majority of developers who would be interested in creating and maintaining an MSS. This leads to constraints which are not related to features of the API but to the needs of the users of the API themselves – the designers and developers who will build multi-surface applications. Because the designers of the API have experience building an MSS, these constraints are derived from their experience. Through the experience of building applications and through discussions and demonstrations to prospective end users, several constraints were derived. These constraints are

related to the type of hardware required, the preferences of users, the platforms which are supported, and how this support is accomplished.

4.1.1.1 Consumer-Accessible Hardware

The first major constraint placed on the API is that it must not require developers to purchase or use any hardware that is not consumer grade. Providing an exact definition for consumer grade hardware is difficult, but in order to meet this goal we have aimed not to absolutely require any hardware which is not marketed or directed towards consumers. Because MSE-API allows users to choose which component devices to use in the system (such as tablets, laptops, wall displays, etc.), this constraint applied mostly to the tracking system. This limited us to using tracking systems which are associated with consumer applications, such as the Microsoft Kinect and Intel Perceptual Computing Camera, which cost between one to two hundred dollars. This is in contrast with professional-grade motion capture cameras which can cost upwards of fifty-thousand dollars. We consider a camera system which costs less than one-thousand dollars to be a consumer level price. This requirement can be summarized as:

1. *MSE-API will work with hardware available at consumer-level prices.*

4.1.1.2 Removing Markers

Since MSE-API is intended to support consumer applications, the type of hardware used cannot impose barriers which prevent users from being able to enter the system and begin using it immediately. Like the earlier constraint, this applies entirely to the type of tracking system used with the API. Some of the available spatial tracking systems require the use of digital markers which assist the camera in detecting and tracking the position of users. These markers, however, are not likely to be acceptable in an industrial situation or in a casual “walk up to use” situation.

Cameras such as the Vicon system or the OptiTrack system, which require such markers, would not be suitable for MSE-API. This requirement is stated as:

2. *MSE-API will use marker-less hardware.*

4.1.1.3 Platform Support

The API should eventually be portable to platforms which are common to the hardware which is envisioned for use in an MSS. Since tabletops such as the Microsoft SUR-40 and the SMART Table both run Windows environments and their touch frameworks require native applications it is necessary for the API to work using C# and .NET. Since many tablets run iOS, and the iPad is the market leader in this space, the API must work in Objective-C and iOS. In the future we plan to expand the support to additional platforms. This leads to the next major constraint which is stated as:

3. *MSE-API will be portable to important platforms for tabletops, tablets smartphones, and wall displays, which are the principal devices used in an MSS.*

Another issue relates to how the API integrates with the given platforms. It is important that the usual workflows and techniques which developers use to create applications not be interrupted by the use of the API. By this we mean that the API should not make it difficult to use standard interface and other libraries and shouldn't create incompatibilities with standard tools and IDEs. This means that:

4. *MSE-API must integrate with commonly used toolkits needed to build applications on the supported platforms.*

With these constraints, which govern design decisions made in the development of the API, it's now possible to consider what features or functions the API must provide to a developer.

4.1.2 Functional Requirements

The functional requirements for MSE-API are derived from the original vision of an MSS described in Chapter 1. Since an MSS is a system that is composed of multiple independent devices, communicating between devices will be a major task faced by developers. It is therefore crucial for usability that communication between those devices be simple for developers to implement, in terms of setting up the communication, initiation of transfers, etc. Therefore, support for inter-device communication is a necessary part of the API. Further, since the usability of an MSS can potentially be improved by the use of spatially augmented gestures, the API must also support developers to incorporate these gestures into the interactions supplied by their systems. The requirements for each are discussed in the next sections.

4.1.2.1 Inter-Device Communication

An MSS must enable developers to accomplish communication between devices, but several unique requirements exist for an MSS. One specific requirement is that the devices must be able to identify the devices they wish to communicate with before they can dispatch messages. This can be accomplished by providing each device with a unique identifier, but the process of hard-coding names introduces brittleness into the system when new devices are added or devices do not exist when the system expects them to. To account for this, the API should provide “device discovery”, which allows devices to broadcast themselves to other devices on the network. This leads to the first requirement related to inter-device communication:

5. *Devices running MSE-API will automatically announce themselves on their network to discover one another.*

Another concern is how the API will send and receive messages. A major inconvenience when dealing with message passing is the process of serializing and de-serializing specific data types.

It should be possible for developers to exchange the most common data types without having to write much code. However, this should not be the only mechanism provided as it constrain developers who wish to create more complex features. The message passing system should both provide support to less experienced developers while not unnecessarily constraining more experienced developers. This leads to the next requirement:

6. *MSE-API should provide a straightforward way to exchange common data types without writing a large amount of code.*

4.1.2.2 Spatial Information

To support gestures augmented with spatial information as described in Chapter 2, it is necessary to maintain a collection proxemic values for certain objects within the MSS and allow developers to query this collection.

Early constraints, such as (1) and (2), place limits on the type of tracking which can be accomplished. As only consumer level and marker-less tracking systems are appropriate for the application scenarios that we want to support, it is necessary to find an approach which allows for mobile devices to be tracked. Since an MSS is primarily composed of devices, the API needs to insure that mobile devices are tracked in a room. This leads to the first requirement related to spatial information:

7. *MSE-API will provide spatial information (location and orientation) for devices.*

Once the proxemic information (location and orientation) is collected for devices, it is next necessary to provide an interface which developers can use to query this information and use it within their applications. Following the principle mentioned several times before, it would be nice to provide developers who are not familiar with networking with the ability to access this data without forcing them to write networking code. However, we wish to leave open the

possibility of interacting with the API on non-supported platforms. This leads to another requirement for Spatial Information:

8. *MSE-API will expose the collection of devices to queries using HTTP, but will also provide client libraries which provide these queries automatically to supported platforms.*

4.1.3 Usability Requirements

The last requirement for the API relate to the usability of the API itself. As mentioned in Chapter 1, usability is often defined as being composed of several dimensions. As these dimensions are not independent of each other, it is possible that improvements to one dimension detract from another dimension. It is necessary therefore to focus on which aspects of usability are the most important for the API. This leads to the only requirement related to usability:

9. *MSE-API will provide a learnable and discoverable API to inexperienced developers while still remaining an efficient tool for experienced developers.*

This will ensure that novice developers can begin using the tool without a great deal of experience with spatial tracking systems or networking, but at the same time ensure that experienced developers can continue to use the tool in a flexible and efficient way.

4.2 API Components

MSE-API is made up of three components: the locator, which collects and provides location information; the visualizer, which displays a representation of the room as seen from the locator; and the client libraries, which provide functions for developers to use on the devices in the system. In a given room all these components are active (see Figure 11). Each of these components is discussed in the following sections.

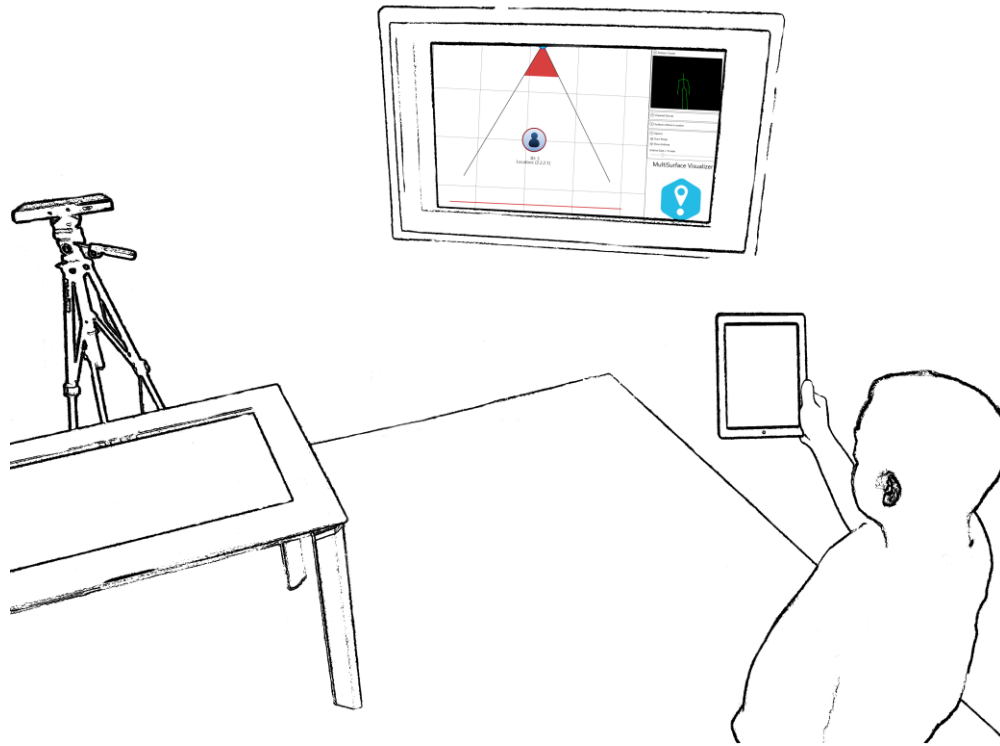


Figure 11: MSE-API Components in an MSS

4.2.1 Locator

The locator collects location and orientation information for devices in the room. A sensor is used to detect the position of users in the system while internal mechanisms in the devices themselves supply orientation information. Location information for static devices, such as a tabletop or wall display, is entered in by users when configuring the multi-surface system. The locator information can be queried by devices when they need location information for some task. It is this location information which devices use to support spatially augmented gestures. An architectural diagram describing the locators connection to other classes can be seen in Figure X. The service provided by the locator is discussed in the following sections.

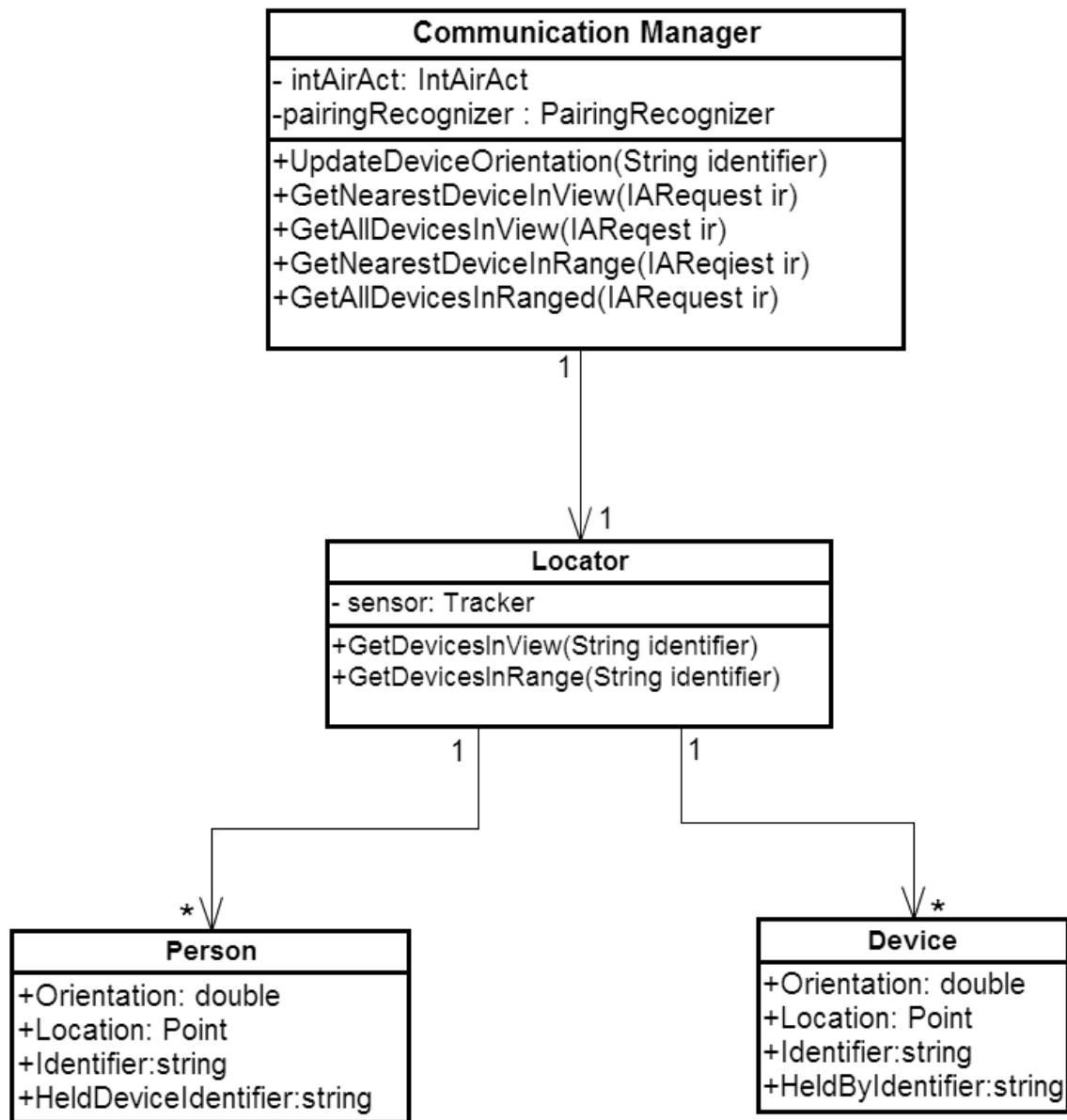


Figure 12: Simplified Architecture of MSE-API Locator

4.2.1.1 Locating Devices

In collecting mobile device information, the constraints limiting the cost of the sensor (1) and the use of markers (2) make it impractical to track devices directly. This is because the accuracy of sensors in the consumer range cannot track the devices directly without the use of markers.

However, spatially-augmented gestures can still be used even without precise positional information if the gesture only needs to know the intended target device to be useful. Therefore, the API utilizes the position of the user holding the device – which consumer level sensors can accurately detect – as a proxy measure for the position of the device itself. As a proxy measure this position is likely accurate enough because users will likely target the person holding the device (when they wish to send content) rather than the device they are holding. The heuristic therefore takes advantage of a kind mental model in which users associate the person holding a device with the device itself. The limitation of this approach occurs whenever the implicit connection between the device and the person breaks down, such as when the device is placed down or exchanged with another user.

However, for this heuristic to work, it must be possible to know which device is being held by which person. To accomplish this, we created a pairing between device and person by having a user perform a waving gesture (see Figure 13). This gesture is simultaneously detected by the sensor and the device, allowing for a correct match.

To collect orientation information (i.e. which direction is a device pointing), we use the internal gyroscope available on many devices. When a user first begins using the system they calibrate the gyroscope towards the Kinect, this provides an absolute orientation relative to the room. Current hardware cannot provide this absolute orientation without this calibration step. This

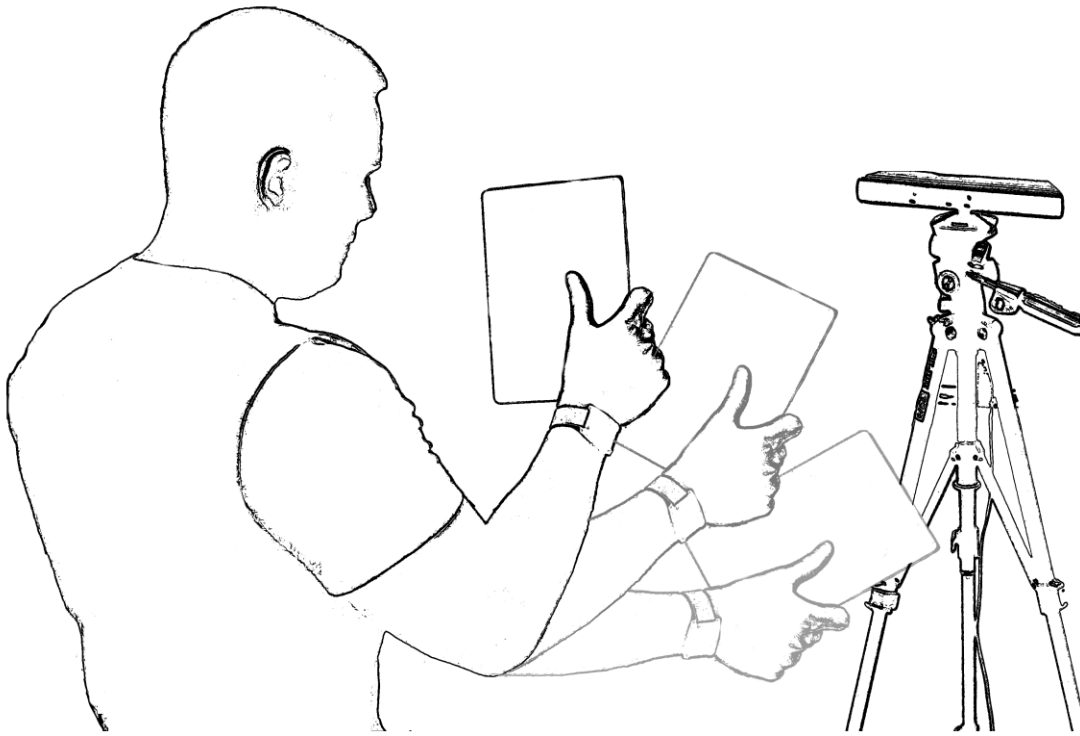


Figure 13: Pairing Gesture Performing With Device

information is sent from the device to the locator. Developers only need to start the API for this information to be collected automatically.

Some devices in an MSS are not mobile and their position need only be determined once and not continuously throughout a session. Devices such as a digital tabletop or a wall display fall into this category. To position these devices, the visualizer (see Section 3.2.3) allows the user to move the device into its correct position. The updated position value is persisted on the static device and will remain unchanged until the room is reconfigured. This process requires some user intervention while positioning mobile devices is done automatically.

4.2.1.2 Querying the Locator

Queries on the locator come in two forms: view-based and proximity-based. A view-based query allows the device to determine which other devices it is facing. A proximity-based query allows

the device to determine which devices it is close to. These queries are necessary to support different types of spatially augmented gestures.

4.2.1.3 Locator Methods

The functionality provided by the locator is exposed using as routes. A route is a unique path defining a specific resource being exchanged. To access the functionality an HTTP request is sent to this route and a response is issues by the locator. Some routes return information about a resource and these are defined using a GET method while other are designed to receive information and these use a POST method. The routes provided by the Locator service are summarized in Table 2. Note that updates to location are provided by accessing the information directly from the sensor.

Table 2: Routes Provided by the Locator

Functionality	Route	Method	Parameters	Output
Update the pairing status of the device when a waving gesture is detected	{identifier}/pairingStatus	POST	Status = Attempting Pair	None
Update the orientation of a device	{identifier}/orientation	POST	Orientation = Orientation in Degrees	None
Request all devices in the current view of the device	/devices/view/{identifier}	GET	None	Collection of Devices in View
Request all devices in a specified range	/devices/view/{identifier}	GET	None	Collection of devices in range

4.2.2 Client Libraries

Specific client libraries are provided for iOS and .NET to assist developers in integrating devices into an MSS. The client libraries accomplish three tasks: (a) detecting and coordinate the process of pairing a device (b) collecting the orientation information required by the locator; and (c) supporting common data exchange tasks in the API.

Before a mobile device is usable in an MSS, several tasks must be handled by that device. Each device using MSE-API is responsible for detecting the pairing motion and communicating this information to the server. Once the pairing is completed, the device begins calculating its orientation using its internal gyroscope and dispatching this to the locator. This is done every 50ms to allow users to visually see their orientation (on the Visualizer) and to determine if calibration is needed.

Developers must themselves decide when their application performs a query, when it sends content and requests with other device and how their device will respond to requests and content from other devices. To assist developers the API provides an additional networking layer that simplifies the process of sending and receiving specific data types. These allow a novice developer to perform simple content transfer tasks without understanding the details of networking issues. These convenience methods allow developers to send and receive dictionaries, images and binary data to another device without needing to understand the details of serialization, message encoding or deserialization. When developers wish to go beyond these convenience methods they are able to work directly with lower-level networking details. Specifically they can define a route, which is a unique path defining a specific resource being exchanged. Such routes are the target of an HTTP request and are common to web development

and programming. Developers can specify their own routes directly and handle the process of serialization, deserialization and responses directly.

4.2.3 Visualizer

4.2.3.1 Visualization of Devices

The visualizer provides a graphical interface showing the location of devices in the system as seen by the locator (see Figure 14). This information is presented as a top-down layout of the system. Devices are presented as squares and are placed outside the room space if they are not currently paired. When these devices are paired they are connected to the circles which represent users in the room. To help facilitate pairing, pairing status is indicated on the device and the user.

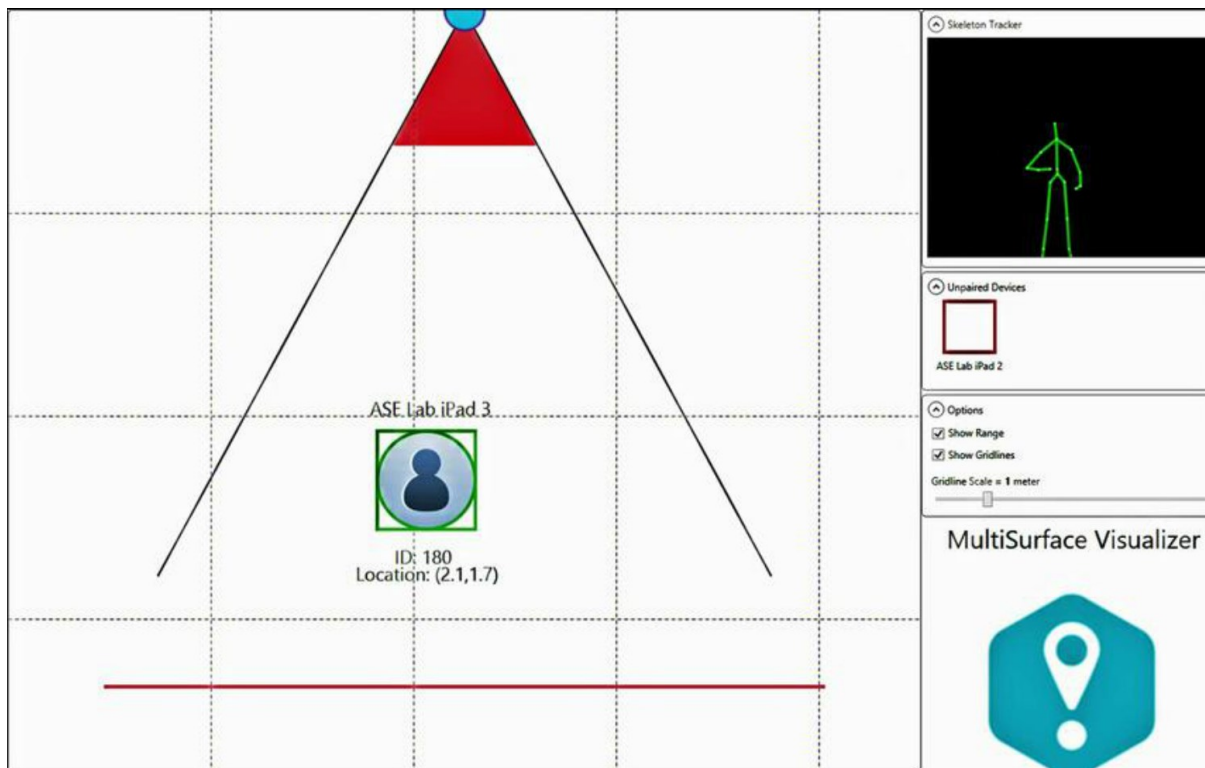


Figure 14: Visualizer displaying a paired person and device

4.2.3.2 Room Configuration

In addition to providing a visualization the visualizer can also be used to configure the layout of the room. By layout we mean the location of fixed devices such as tabletops and wall displays. These devices aren't tracked by the location sensor, so a user must define their location. This is done by dragging a static device as representing by a square on the visualizer. To indicate its position the location is displayed underneath the device.

4.3 API Usage Example – Pour Gesture

To demonstrate how the various components of MSE-API work together, we will consider in this section how to implement the pour gesture demonstrated in previous work [9]. The pour gesture can be used to trigger content transfer when a user rotates a mobile device in a way analogous to pouring out a liquid. In our example, a user will use a pour gesture to transfer an image to a digital tabletop from his mobile device. To accomplish this task, we will need to detect the pour gesture and query the locator to determine whether any devices are nearby. Once these devices are detected, we will dispatch an image to them. For completeness, we will also show how a developer can define responses to the arrival of the image.

4.3.1 Detection & Query

The detection of the pour requires the device to monitor the gyroscope to determine if the device is rotated to the relevant position. While this is not provided by the device directly, MSE-API provides utility methods to support the detection of a pouring motion. Once the pour has been detected, it is necessary to begin the query to the locator for other devices in a close proximity. The client library provides convenience methods for querying by proximity and the user provides a callback method to be run when the detection is completed. Both these steps are illustrated in Figure 15

```

//Detects when the pour gesture has been detected
-(void)gestureRecognizer:(DSPourGestureRecognizer *)recognizer didDetectPourGesture:(NSI
{

    //Queries the server for any device within 0.5m of range
    [self.mse.locator nearestDeviceInRange:0.5 success:^(MSEDevice *device) {

        //Send Image
        [self sendImageToDevice:device];

    } fail:^(NSError *error) {

        //Inform the user that no device was in range
        [self updateWithPourFailed];

    }];
}
}

```

Figure 15: Detecting Pour Gesture & Querying Locator

4.3.2 Sending & Receiving the Image

Once a device has been detected it is then necessary to send the image. For simplicity, we have illustrated this in a separate method. MSE-API provides a convenience method for sending images because images are a common data type which developers will likely wish to send. Developers must simply create the appropriate image object and then send the object to the device (see Figure 16). On the other device – a tabletop in this example – the developer must define the behaviour that should occur when an image is received (see Figure 16). It is clear that, once the image has been received, the developer is free to treat it in any manner considered appropriate.

4.3.3 Sequence of Messages

In order to implement this scenario several messages must be exchanged between the devices and the locator. Figure 17 indicates the sequence of messages related to pairing and orientation updates which must take place before a user can begin using the system. After this has happened shows the sequence by which devices in range are found and content transferred to it.

```

- (void) sendImageToDevice:(MSEDevice *) device
{
    //Capture the appropriate image
    UIImage *image = [UIImage imageNamed:@"SomeImage.jpg"];

    //Send Image As a Pour Gesture
    [self.mse sendImage:image withName:@"SomeImage.jpg" toDevice:device withGesture:[MSEGesture pourGesture]];
}

internal void SetupMultisurface(MSEMultiSurface mse)
{
    mse.ReceivedImageHandlers.Add(delegate(System.Drawing.Image image, String imageName, MSEDevice originDevice, MSEGesture originGesture)
    {
        Application.Current.Dispatcher.BeginInvoke(new Action(() =>
        {
            DisplayImage(image);
        }));
    });
}

```

Figure 16: Sending & Receiving an Image

4.3.4 Summary

From this example we can see that with less than 100 lines of code it is possible to implement an important, spatially-augmented gesture. MSE-API provides useful and appropriate convenience which are useful for carrying out this task.

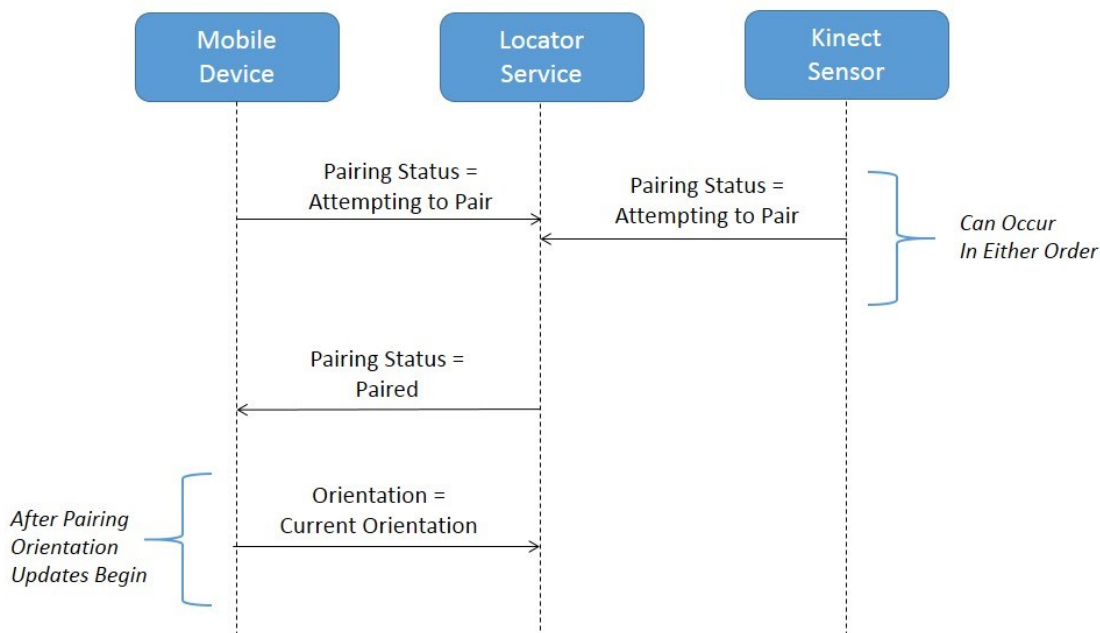


Figure 17: Sequence of Pairing & Orientation Updates

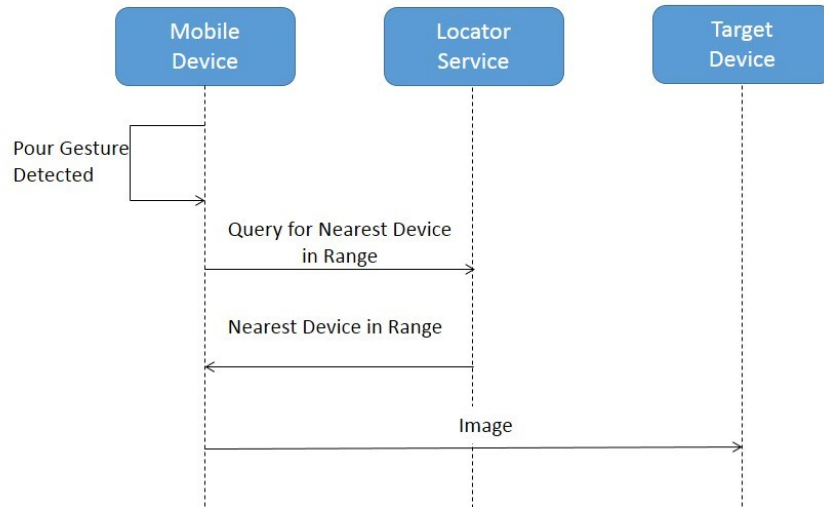


Figure 18: Sequence of Locator Query & Content Transfer

4.4 Feature Comparison

Given these constraints and requirements it is reasonable to ask if any existing API provides these features. In this section the existing APIs which were presented in detail earlier are compared with MSE-API. These comparisons are done separately for APIs that support communication tasks only and those which provide spatial information.

4.4.1 Communication Features

In order to support developers in building multi-surface systems it is necessary that the API assist developers with communication between devices. This support is connected to the main task in an MSS, the content and control transfer task. In order to accomplish this transfer, some form of communication is necessary. In Table 3 several communication APIs are compared based on their support for these features.

Table 3: Comparison of Existing APIs on Communication Features

Feature	3MF	Event Heap	Magic Broker	ROSS
Automatic Device Discovery	YES	PARTIAL	NO	YES
Straightforward exchange of common data types	NO	NO	NO	NO
Extensible mechanism for other exchanges.	NO	YES	PARTIAL	PARTIAL
Familiarity with common networking approaches and tools	PARTIAL	PARTIAL	YES	NO

4.4.2 Spatial Locator Feature

The spatial location features for MSE-API allow developers to support spatially augmented gestures. Supporting these gestures is a major goal of the API. In Table 4 several APIs and toolkits are compared based on their support for features related to spatial location.

Table 4: Comparison of Existing APIs on Spatial Location Features

Feature	Proximity Toolkit (VICON)	Proximity Toolkit (Kinect)	Easy Living	Near Me	Shared Substance
Requires only consumer level hardware	NO	YES	NO	YES	NO
Marker free tracking	NO	YES	YES	YES	NO
Provides position and orientation information for mobile devices	YES	PARTIAL	PARTIAL	PARTIAL	YES
Allows the use of standard toolkits and libraries	YES	YES	NO	NO	NO
Toolkit provides convenience methods for accessing spatial information	YES	YES	NO	NO	PARTIAL

4.5 Conclusion

MSE-API was designed as an API for supporting multi-surface systems that are augmented with spatial gestures. Since MSE-API was designed to support practical and realistic scenarios, a number of constraints apply to the API. These constraints on the underlying sensors related to cost and practicality, end users cannot be expected to buy extremely expensive hardware or use awkward physical markers. Other requirements were defined related to features and the usability of the API. The components of the API were reviewed, confirming that it meets the requirements laid out earlier. Finally, a usage example was provided to demonstrate how MSE-API's functionality can be used.

Chapter Six: Skyhunter Case Study

As part of the overall evaluation, I wanted to determine how efficient developers were in building applications which used the API. Efficiency is a separate facet of usability which is different from learnability or discoverability. In order to assess this facet, we wanted to determine how long it would take for experienced developers to complete a medium-sized application using the API. Since only the authors of the API were sufficiently experienced with the API to conduct this study, it was decided to pursue a self-evaluation of the API. Two developers, the author of this thesis and a colleague, developed the application over a one-month period. During that time, time logs and qualitative experiences were collected.

In this chapter the requirements and issues which the application needed to solve will be presented, along with a description of the features provided by the application. Following this is a breakdown of the time spent developing the application. Finally, we'll present an argument that the time spent is low considering the application built and propose that the evaluation indicates that the API supports efficient development.

6.1 Skyhunter & Data Problems in Oil & Gas Exploration

Skyhunter is a local industrial partner working in the area of oil and gas exploration. They have developed technology that can detect chemicals in the air that can indicate the presence of underground reservoirs. The output of this technology is maps which indicate the likelihood of an underground reservoir existing at a given location. To be useful, these maps must be combined with other data from other oil and gas specialties, data such as seismic data, well logs, land use information, etc. This integrated data is also usually analyzed and presented to a group of specialists from different disciplines. Building an application to support this analysis process requires that several different data types be supported. Further, it is necessary to support a multi-

user, multi-device scenario with different users have different roles within the system. Issues related to content transfer are important in this type of environment and so we felt a multi-surface application would be appropriate.

6.1.1 Types of Data

Several types of information are necessary as part of oil and gas exploration. Some of this data is provided directly by Skyhunter and others are collected through other means. All of this information is geographical in nature and is stored and prepared using a GIS system. Three types of map data were considered particularly important by Skyhunter: microseep data, subsurface data, and well data.

6.1.1.1 Microseeps

The surveys conducted by Skyhunter involved the use of specialized sensors fitted to the nose of an aircraft. The aircraft flies near to the ground in a specific grid pattern. The captured sensor information is then interpolated and compiled into a map, which indicates the areas where microseeps (trace hydrocarbons that are aerosolized above and indicate the presence of an underground reservoir) exist and their level of intensity. A grid pattern flown for a particular survey conducted in Australia was chosen as the example dataset for this application; (as seen in Figure 19).

6.1.1.2 Subsurface Data

According to the information provided by Skyhunter, microseeps alone are not sufficient to make accurate predictions regarding the location of oil and gas reservoirs. It is necessary to combine this information with data about the subsurface. This data was provided as iso-depth contours,

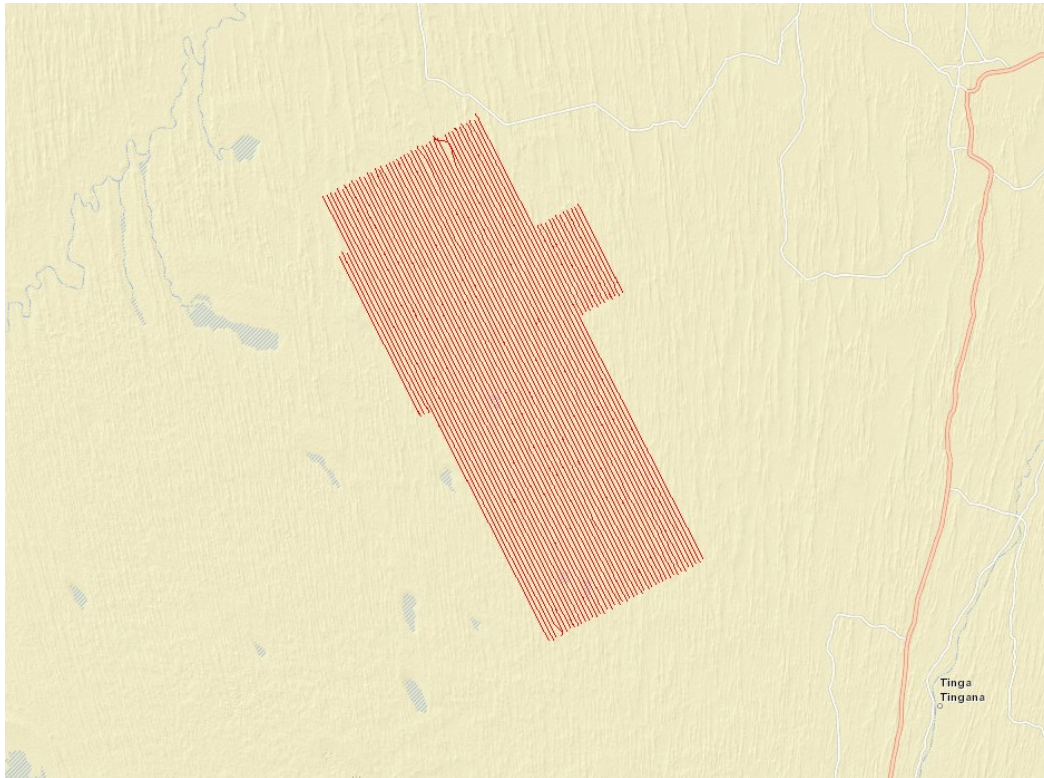


Figure 19: Flight Grid Pattern

which provide a topographic representation of the subsurface area for specific geological zones.

This information is shown as contours in Figure 20.

6.1.1.3 Well Data

Well data, information about previously drilled wells, is the last type of data needed for the system. This information is a kind of ground truth because it reveals whether or not the particular well was successful (i.e. intersected a reservoir) and which were dry (i.e. did not intersect a reservoir). This information is provided for many wells within the geographical area of the conducted survey.

6.1.2 Roles in the System

Exploration for oil and gas data, according to Skyhunter, involves several different specialties such as geologists, geophysicists, reservoir engineers, and landmen. Each of these specialties has its own data and analysis techniques. Traditionally, the analysis by these different specialties was handled separately. But bringing these roles together during the analysis and exploration phase has been a major goal in the oil and gas industry. Supporting collaborative analysis by these separate roles is a major objective of the application.

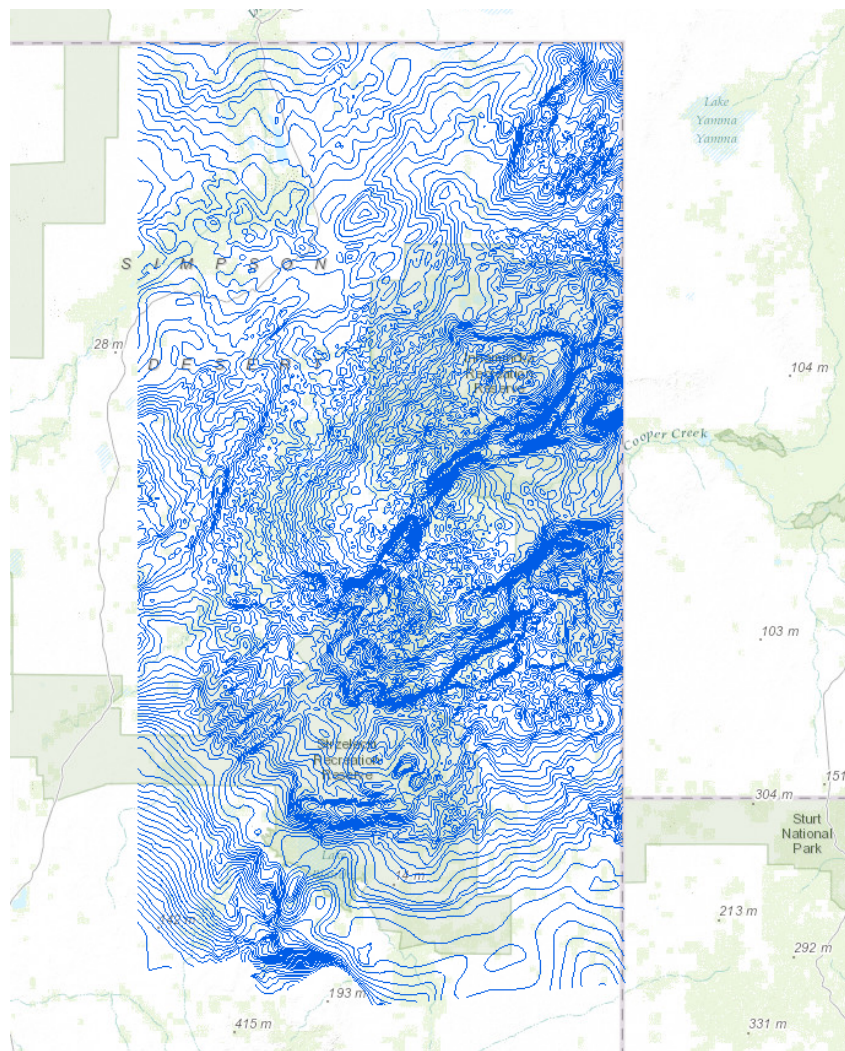


Figure 20: Subsurface Contours

6.1.3 Issues & Difficulties

Building an application to support the exploration of oil and gas reservoirs, with a special emphasis on a multi-role environment, raised many issues. Because the system must work for different roles, it must be straightforward to bring data into the system and take it out. Because the analysis would involve a group of users working, sometimes individual and sometimes in subset groups, data would need to be moved fluidly between different users during the course of analysis.

6.1.3.1 Data Entry and Exit

One of the major issues raised by Skyhunter was the difficulty of bringing data (such as the above mentioned data types) into and out of the system. They mentioned that specialists would store data on their computers and tablets, but that bringing this data into the system was often tedious and time consuming. Likewise, if some final work products were created during the analysis it should be easy to capture them for later review.

6.1.3.2 Fluid Data Transfer

In scenarios described by Skyhunter, analysis would often take place with the entire group and sometimes with individual or subset groups. To support this, we suggested the use of tablets and a tabletop in the system. They mentioned that these groups should be able to move data between themselves and the main group fluidly.

6.2 Skyhunter MSS Application

Given the scenario and requirements described above, we suggested to Skyhunter that their application would be an excellent candidate for a multi-surface system. As the major problems faced by the users related to fluid transfer of information between devices, the application appeared to be a good opportunity to use MSE-API to build a multi-surface system around the

exchange of oil and gas data. An application was designed to address the previously-mentioned issues. The application, its components and features, and how it addresses the issues raised by Skyhunter are discussed in this section.

6.2.1 System Components

To support collaboration we decided to use a tabletop as the central hub for information in the system. Specifically the tabletop was a Microsoft SUR-40 running a custom map display application. In addition to the tabletop, several Apple iPads were used – also running custom software. The tablet application was designed to switch between roles, with separate data for geologists, landmen, and geophysicists (see Figure 21).

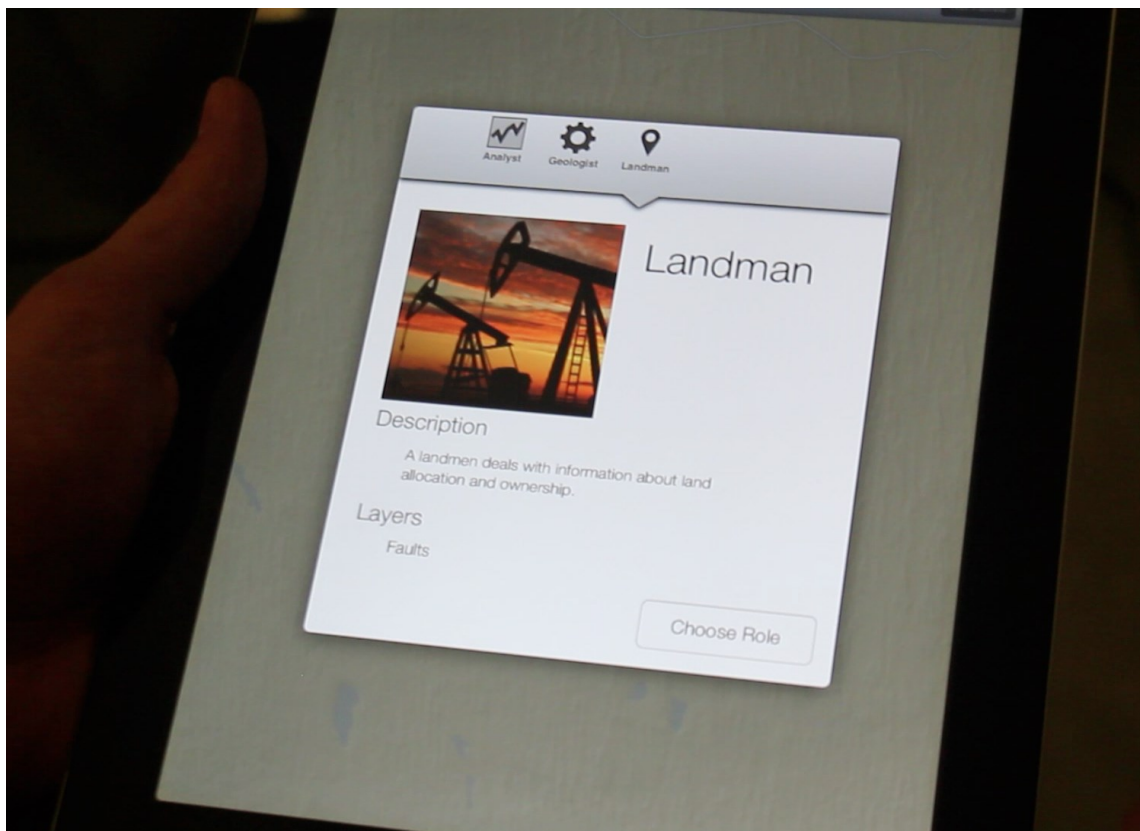


Figure 21: Role Selection in Skyhunter MSS

6.2.2 Data Transfer Features

Another major issue raised by Skyhunter was regarding fluid data transfer during analysis tasks. This is connected to the content transfer task, which was discussed earlier. The content in this case was different types of geological data. All this data was geographical and could be represented as a map. Therefore, the core feature in the system would be transferring maps between the different component devices. Different spatially-augmented gestures were chosen for several distinctive transfer tasks.

6.2.2.1 Tablet to Tabletop

Assuming the users arrived with some relevant data on their tablet device, it was necessary to support transfers from tablet to tabletop. To trigger this transfer, we chose to use a pour gesture. In this gesture, the user stands over top of the tabletop and inverts their tablet. This is done in a manner similar to how someone might pour the contents of a binder onto a table. To distinguish which layers a user wished to transfer, a special layer dock panel was created on the tablet application. Once the layers were selected, the user could approach the tabletop and perform the pour gesture (see Figure 22). Since the screen of the tablet is not visible during the gesture, audio feedback is used to indicate whether the transfer was completed successfully.

6.2.2.2 Tablet to Tablet

Another design issue mentioned by Skyhunter was the need for smooth and fluid transfer of data between users or groups of users. To support this, we provide a gesture, suggested by previous studies, called the flick gesture. In this gesture the user performs a swiping motion towards the target of the transfer. In our application the map, which is panned and zoomed using swiping actions, would normally interfere with such a swiping action. Therefore, the gesture is performed in the separate dock panel. On the targeted iPad the map appears as a notification on

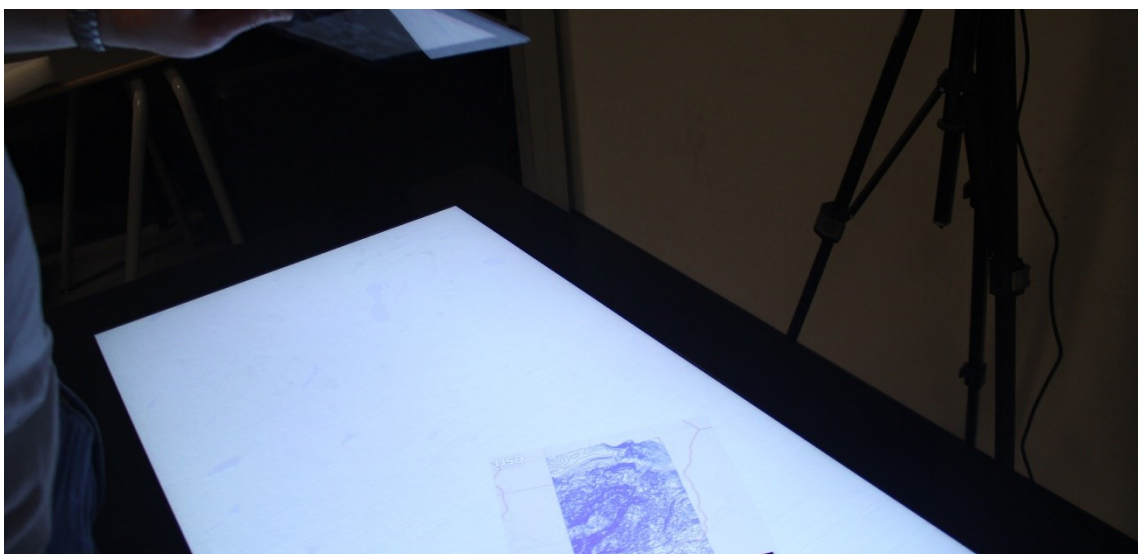
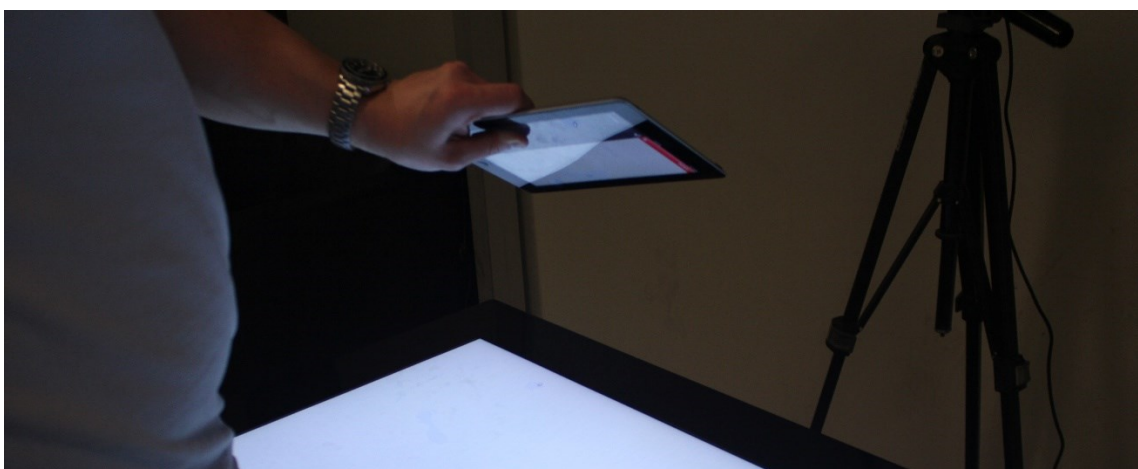


Figure 22: Pour Gesture to Transfer Layers

the bottom of the screen. Once selected, the map is then added to the current map view (see Figure 23).

6.2.2.3 Tabletop to Tablet

Once data has been added to the tabletop and analyzed, it is necessary that users are able to capture that data on their tablets. To do this, we provide a gesture called the camera gesture, which like the previous gestures was elicited from users in a previous study [7]. To use this gesture, the user aims their tablet as if they intended to capture a picture. Pressing the camera icons provides a list of layers which are currently loaded on that device. Users can select whichever layers they want to capture. The layers are then loaded onto their own device (see Figure 24).

6.2.3 Issues Addressed

The application created for Skyhunter used MSE-API to solve issues related to supporting analysis tasks oil and gas reservoir exploration. The application requirements were derived from Skyhunter and were defined as (a) straightforward data entry and exit from the system, (b) fluid data transfer during analysis.

6.2.3.1 Supporting Data Entry & Exit

To support this task, we allow users to bring data with them on their own devices. When the tablet application loads, it grabs this data based on a configuration file, but this could be extended easily to use the file system directly, meaning that users could add data from other sources at their convenience. Once their tablet is brought into the Skyhunter MSS, they can move data to other users and to the tablet using the previously mentioned flick and pour gestures. Once data has been analyzed and brought together from various sources, it can be captured back to the tablet using the camera gesture.



Figure 23: Flick Gesture to Transfer Layers

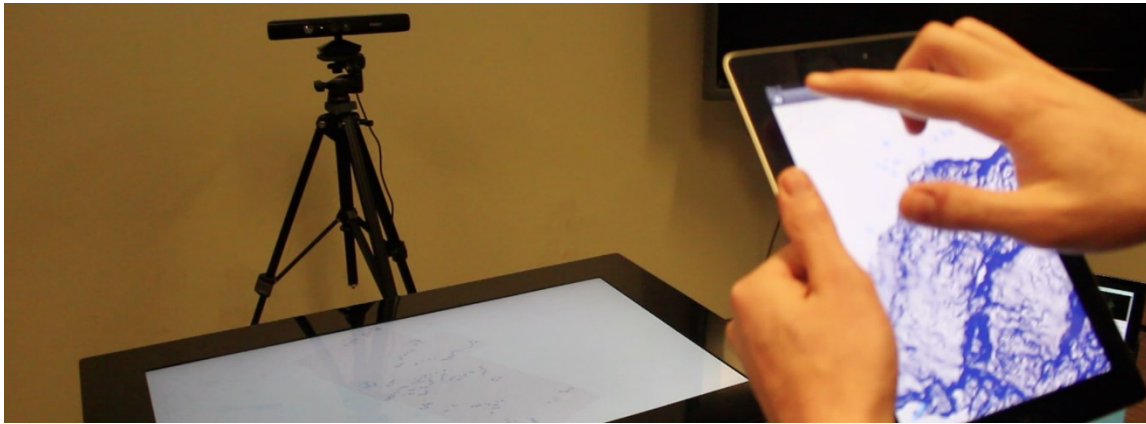


Figure 24: Camera Gesture to Transfer Layers

6.2.3.2 Fluid Data Movement

As the users are working together, it's possible for them to move data around the system easily.

Data can be transferred to other users using the flick gesture, transferred to the tabletop using the pour gesture and capture from the tabletop using the camera gesture.

6.3 Study Results

The Skyhunter application was developed over the course of a month and involved roughly forty-two hours of work. When development began all of the required data was currently processed and stored in a cloud based GIS system. However, the applications had to be created to use this GIS data. It involved two software applications, one running on a Microsoft SUR-40 tabletop and the other running on an Apple iPad. For each component of work completed the time and details of the task were logged. Each of these logged tasks were categorized as utility tasks, interface tasks, or API tasks. In this section we'll review the time allocated for the project and some qualitative lessons learned. The threats to the validity of this study will also be addressed.

6.3.1 Results

Several categories of tasks were completed in the Skyhunter project. Interface tasks were defined to be those which involved creating interface elements. This usually meant customizing standard controls or adding in third party controls. Utility tasks were those which involved adding or configuring third party libraries, such as the GIS library used for presenting maps. Finally API tasks were defined widely as any task which involved the use of the MSE-API library. Of the forty-two hours spent on the application, twenty-two hours were spent on interface elements, twelve were spent on the API, and eight on utility tasks. A breakdown of these can be seen in Figure 25.

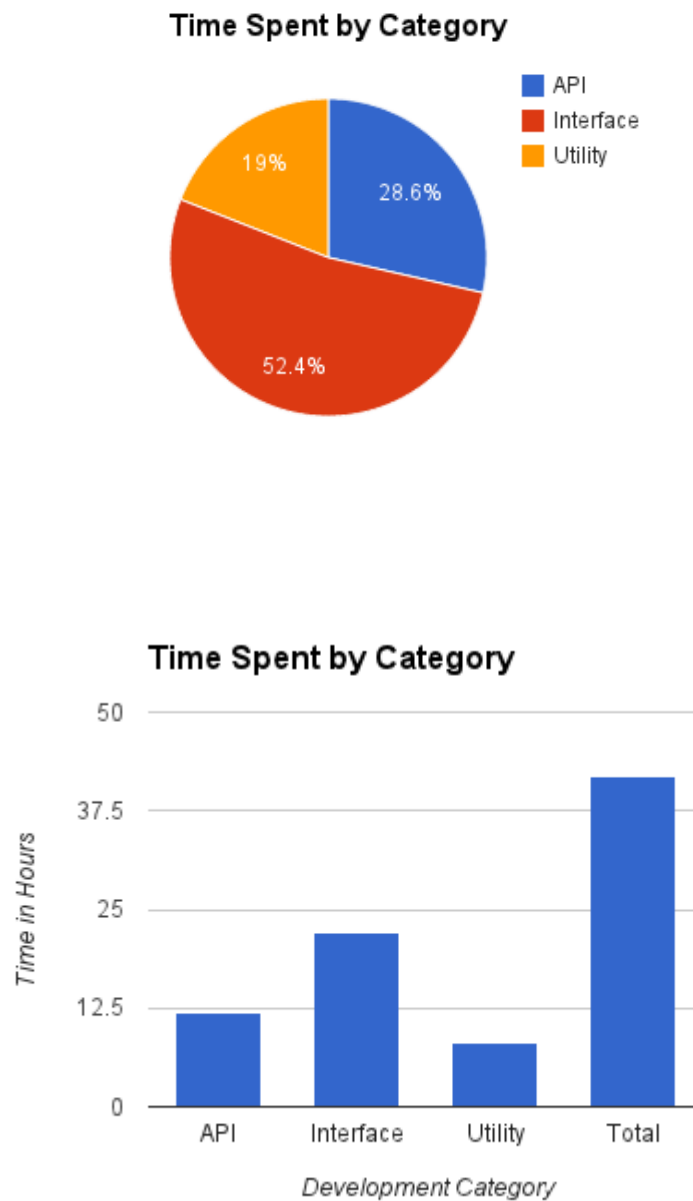


Figure 25: Development Time (Person-Hours) by Category

6.3.2 Discussion

6.3.2.1 Time to Completion

Two developers worked developing the API, sometimes pairing and sometimes working individually. The time spent on the API accounts for only 28% of the total time on the project. This figure is broadly defined it includes any task that involved the use of MSE-API. For example, the task of serializing the map data to be sent via the API is included in this category. We consider this value to be quite low considering the size and complexity of the project and the number of features supported. The total person-hours of the project amount to a single full time week of work. On this basis we see supporting evidence that MSE-API is an efficient tool for building multi-surface systems. However, we concede that our results cannot definitely show this.

6.3.2.2 Other Lessons Learned

While the developers of the Skyhunter MSS project were knowledgeable about the API, some interesting lessons were learned while developing this project. The most interesting of these was the relative importance of using standard web technologies. Because the communication used in MSE-API is based on standard HTTP messaging, it is very straightforward to test and debug applications. For example, in the camera gesture described earlier, the targeting iPad performs a GET request on the currently available layers on whatever device it is targeting. To debug this feature it is necessary to just enter the correct URL into a web browser (see Figure 26). This design decision turned out to be quite useful during the course of development.

Another major benefit was the decision to provide the convenience of helper functions for sending and receiving common types of data objects. By encoding the necessary information for loading a map into a dictionary (a set of key-value pairs), we were able to accomplish nearly all

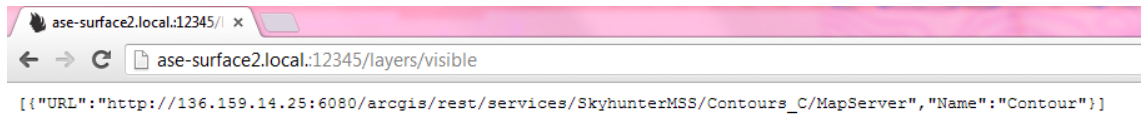


Figure 26: Examining Visible Layers Using a Web Browser

the data transfer tasks. However, for the camera feature it was necessary provide data for a specific request. To accomplish this we took advantage of the lower level networking functionality exposed by the API.

6.3.3 Threats to Validity

Several issues threaten the validity of this study. First, because the authors of the API were evaluating the API themselves, their knowledge of the internal structures of the API and its implementation gave them an advantage over other developers who were new to the API. It is also possible that the task chosen was not a good representative of a general multi-surface system. To address the first issue, a second study will be performed with novice developers over the course of several weeks, this study is presented in Chapter 6. Because the developers previous experience building application prototypes for Skyhunter this might have also caused less time to be expended building the system then might have otherwise been required. Likewise the experience of the Skyhunter applications helped influence the design of the API and this too might have caused a reduction in development time.

6.4 Conclusion

One of the main goals of MSE-API is to provide an API which is learnable and discoverable to novice developers while still being efficient for experienced developers. To assess this second criteria, we built a relatively complex application prototype using MSE-API. Two developers, one of whom was the lead developer of the API, created the application in forty-two person-hours of work. Of this, only twelve hours were spent directly on work with the API. We consider

that this number is relatively low for such a complex application and that this suggests, but doesn't definitely show, that the API is an efficient tool for experienced developers. This argument is based on the fact that the developers conducting the study were experienced with the API.

Chapter Seven: C4I Case Study

As one of the goals of the API was ensure that it was both learnable and discoverable, we conducted a limited and initial case study with several student intern developers. Studying how inexperienced developers use the API will allow us to assess its learnability and discoverability. Using a case study rather than controlled experiment allows for a more realistic assessment of the API's usability. It insures that the features that developers are asked to create are realistic and part of a real world multi-surface application. Since development normally takes place in several sessions over a period of weeks, using a case study provides a better approximation of how developers would actually use the API.

During this case study, several developers used MSE-API to add multi-surface features to an application already under development. This study took place during a single iteration of the project which lasted three weeks. The application is a real world system being actively developed with an industrial partner for emergency planning and simulation. During development, detailed time logs were kept for each feature added and a questionnaire was completed by the developers at the end of the study.

In this chapter, some background regarding the application, the issues which it intended to solve, and the features added during the case study will be presented. Results will then be presented from the study itself, including time logs and qualitative metrics. We then argue based on this evidence that the API is discoverable and learnable to novice developers.

7.1 C4I & Emergency Planning

C4I Consulting is a Calgary area company working in the area of emergency planning. As part of their business they have developed software which supports planning and simulating emergency scenarios. This software allows governments and businesses to create and validate plans for

dealing with emergency situations. Current versions of their software currently run only on desktop PCs. C4I expressed an interest in creating a multi-surface prototype of their software. Since emergency simulations involve small groups of working together within a single room, this application is a good candidate for a multi-surface application.

7.1.1 Emergency Planning & Simulation

In order to deal with emergency situations such as chemical spills, major fires and other accidents, governments and businesses create plans in advance. These plans provide instructions and steps for various emergency responders to follow in the event of an emergency. But how can these agencies feel confident that these plans are good? Software that can simulate both an emergency and the planned response allow teams to validate their plan.

Once the plan has been created it is simulated in the most realistic manner possible. This typically occurs in a command centre or control room. This would be the environment where a real emergency would be managed. During the simulation, different users are present representing different agencies, such as police, fire, EMS and hazardous materials (HAZMAT). As events happen in the simulation, new data and information is made available to the participants. Typically, these external events are driven by a director who causes the events to appear on the simulation system. As different users respond to events and carry out the plan it is possible to find defects in the plan itself – such as unrealistic assumptions – and to gauge its practicality.

7.1.2 Roles & Content

During the simulation different users will have different roles. Some of these roles are external to the application such as the user driving the various events in the simulation. As mentioned before each agency involved in an emergency will be represented by a user. During a real emergency,

each of these users would be responsible for communicating with their own agency, bringing in new data and representing the concerns of that agency. In a practical sense this means that each user has their own concerns and data which they will sometimes wish to share with other users or with the whole group – one of the ideal use cases of MSSs described earlier.

The majority of data in an emergency simulation is geospatially referenced and is stored in a GIS. Some of this data may be the location of specific entities, such as police cars, emergency vehicles, and other responders. This might also include the location of the incident and the exclusion zone created around it. In addition to this, users might also wish to create annotations either for their own agency or for the whole group. Because this data is confined to a relatively small geographical area it is important it does not become confusing or clustered.

7.2 C4I MSS Application

A multi-surface application was created to support emergency response planning and simulation. The application is an MSS-based tool designed to run on an MSS composed of one Microsoft SUR-40 tabletop and many iPad tablets. The tabletop and the iPads display relevant GIS information which is pulled from an ArcGIS Server. While the system provides many features related to emergency planning, we will focus specifically in this section on the data transfer features which use MSE-API. The general structure of the system is described along with the features of the application.

7.2.1 Structure of the System

The system is divided into two separate applications: the tabletop application and the tablet application. During an emergency planning session we envision the tabletop being placed at the centre room while users, each holding a tablet, work around it.

7.2.1.1 Tablet Application

Depending on the role of the users, ePlan MSS will display different content on their tablet and different functionality will be available. Users must choose their role before using the application, the appropriate data is then loaded for that tablet. Depending on their role, users can annotate as well as pan and zoom the map.

7.2.1.2 Tabletop Application

The tabletop application provides a public space in the centre of the control room. The application displays entities and annotations which have been transferred to it by users. It is also possible to perform other functionality, such as choosing a path for a specific entity to travel and annotating the map directly on the tabletop.

7.2.2 Transfer Features

In order to address the usability issues mentioned earlier, several data transfer features were added. It was decided by the industrial partner that transfers between devices in the room was not a high priority, so transfer features target the tabletop.

7.2.2.1 Pouring Annotations to Tabletop

Annotations can be created by users on their tablet application. While these annotations are usually visible only to the role which created them, it's sometimes desirable that annotations might be made visible to the entire group. To do this, the annotation layer can be transferred from the tablet to the tabletop. This transfer is initiated by a pour gesture which mimics the action of pouring content out onto the tabletop and which was discussed in detail in Chapter 2. A user performs this transfer by standing near the tabletop and performing the pour action with their tablet. Once this action has been performed, the annotation layer appears on the tabletop (see Figure 27).

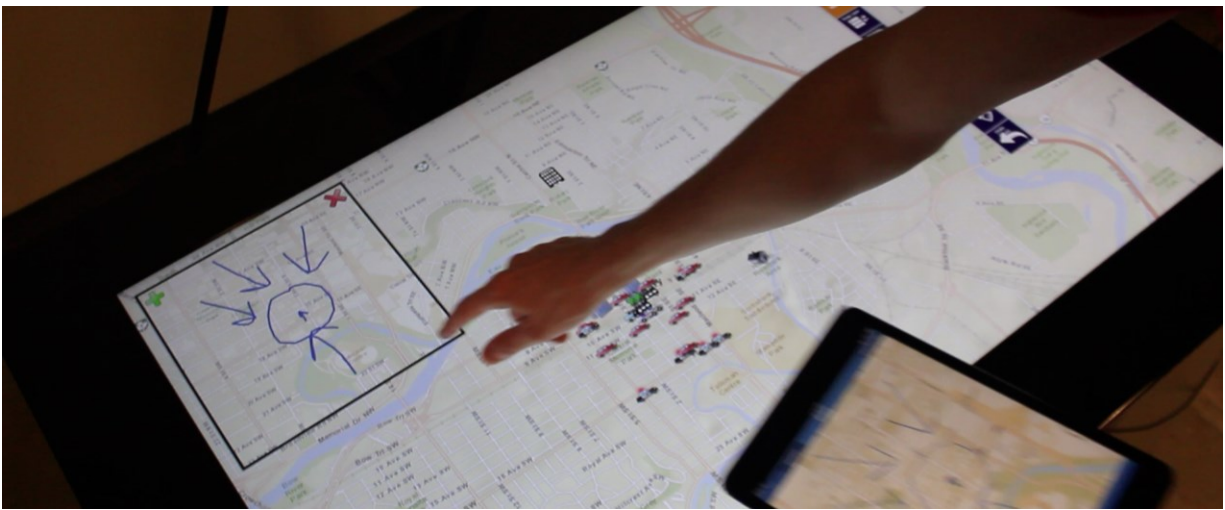


Figure 27: Pour Gesture to Transfer Annotations

7.2.2.2 Capturing Entities from Tabletop

By default, the tablet application displays only those entities which are related to a specific role. That is, the location of police cars are visible only to users with the police role. If a user wishes to see more entities, he can capture those currently visible on the tabletop. To do this, a user points his device towards the tabletop and presses a button to retrieve the entities. Once captured, these entities are displayed on the tablet (see Figure 28).

7.2.2.3 Sending Extent to Tabletop

In exploring or navigating through geographical data, it's often necessary to pan or zoom to specific area of a map. This position and the degree to which the map is zoomed is called an extent. If a user had to navigate to a specific area to show other users some important feature of that area he would need to repeat the process of panning and zooming on the tabletop. To avoid this inconvenience, we allow users to dispatch their current extent to the tabletop. This is done using a button: a user points towards the tabletop, presses the button, and the map is moved to the extent of the users device.

7.3 Study

Our study occurred during a single iteration of the C4I project, encompassing about three weeks of development. The participants had been working on the C4I project during previous iterations before the study. All the participants were intern developers in the lab and undergraduate students in Computer Science or Software Engineering. Three male and one female participant made up the development team. The participants had moderate to minimal experience using Objective-C and C# languages, being comfortable with the syntax of the language but having some difficulties with the associated frameworks (see Figure 29).

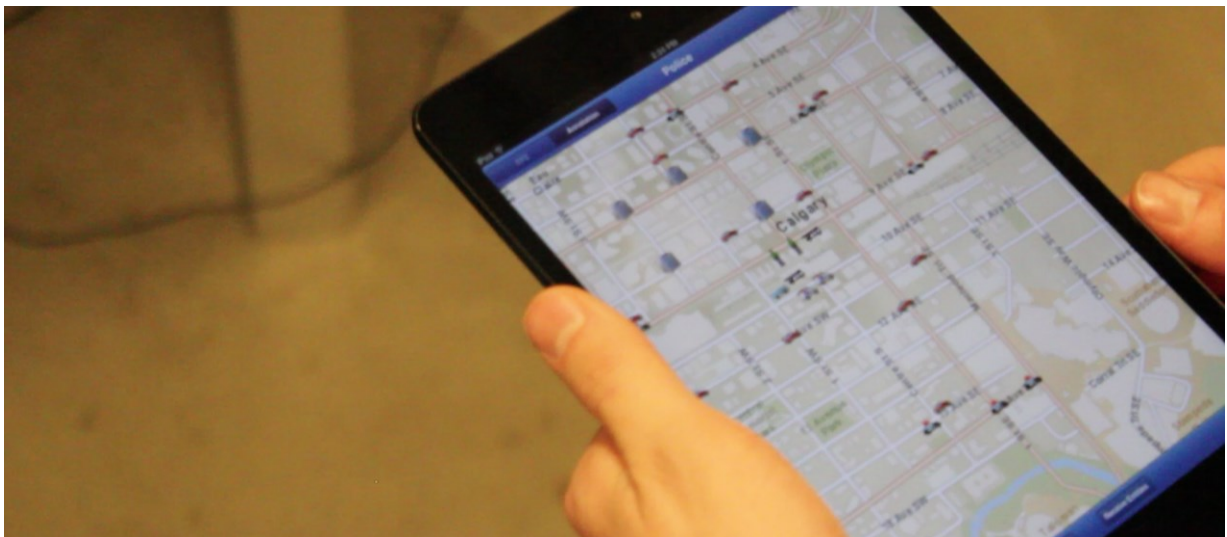
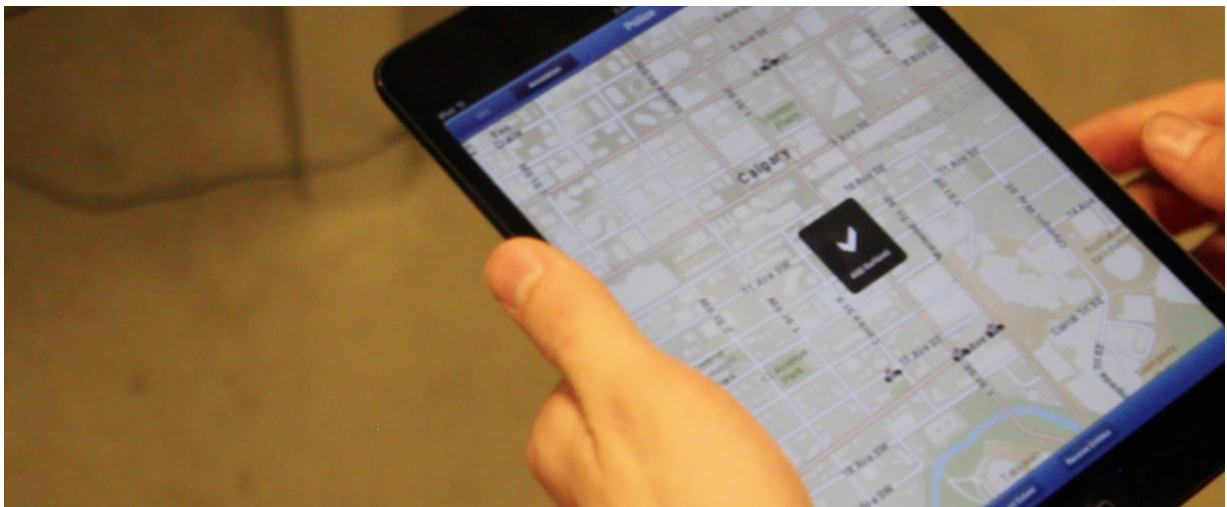
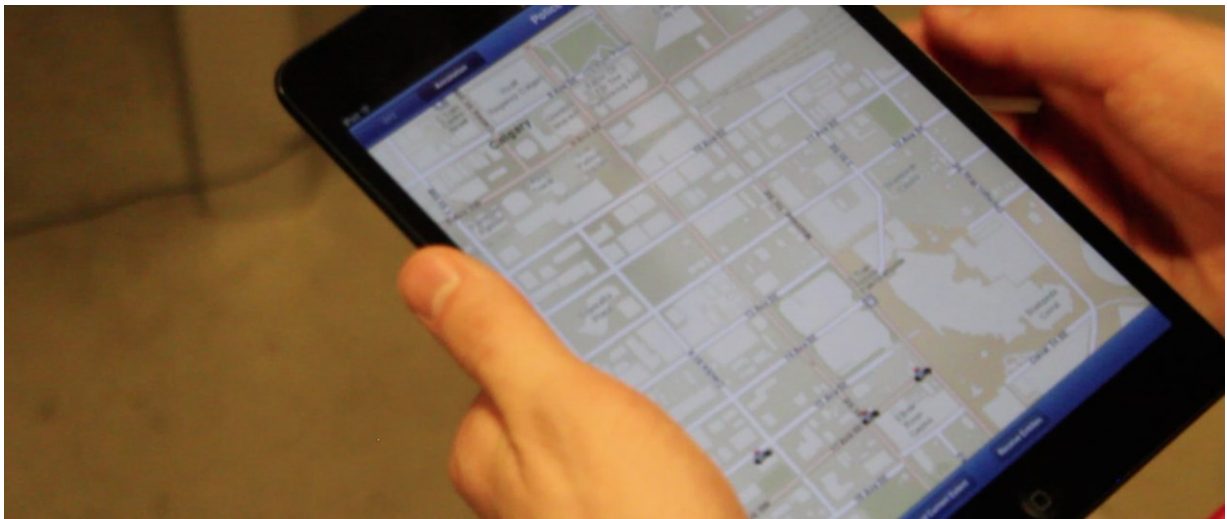


Figure 28: Button Press to Capture Entities

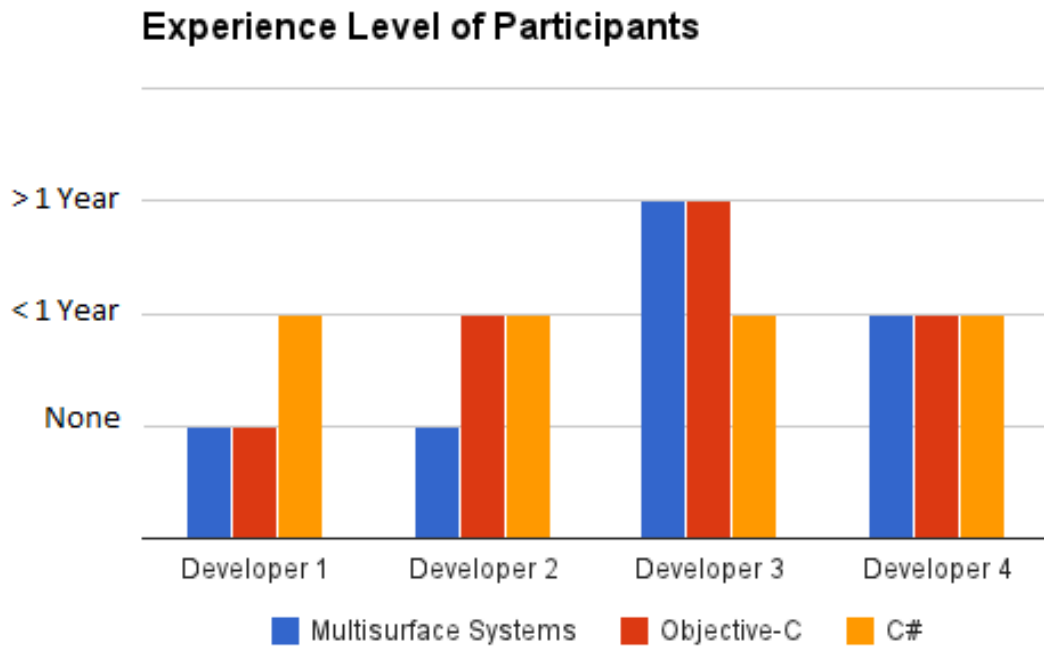


Figure 29: Experience Level of Participants

During this iteration other features were implemented, but only those related to MSE-API were logged and recorded. The study involved implementing the three previously described features for transferring data using MSE-API.

In this section, we'll review the time that was expended completing these features and the feedback provided by the intern developers.

7.4 Results

7.4.1 Time Logs

During the study, developers were asked to record the time they spent working on each feature. In total, eight person-hours were spent on of three features involving the API. The time spent on each feature is summarized in (Figure 30).

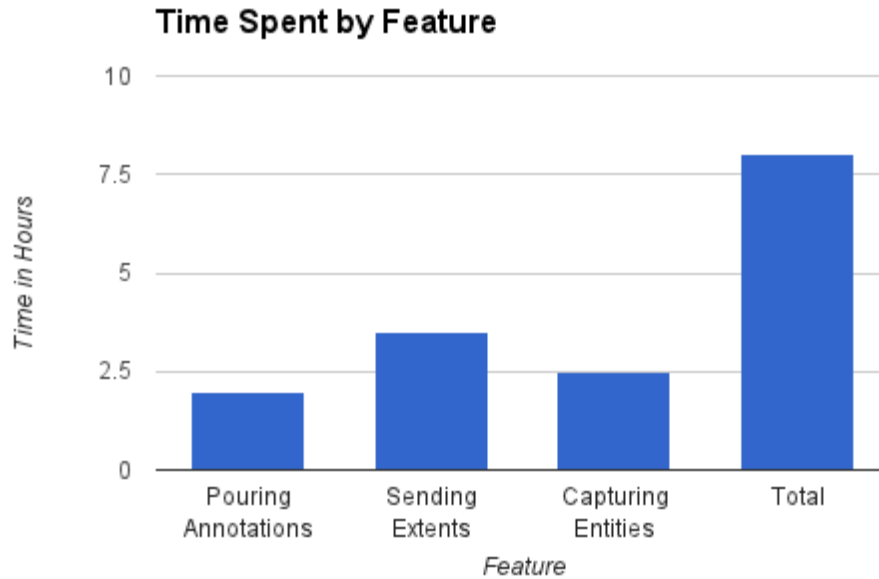


Figure 30: Time Spent on Tasks

During the development process different developers worked in pairs and not every feature was worked on exclusively by the same developers.

7.4.2 Code Analysis

Because the output of the study was code written for the application. It is possible to analyze and review the code to see if the API is being used in the correct manner. We will analyze the code written for the tablet application and the tabletop application separately.

7.4.2.1 Initialization & Setup (Tablet)

Several aspects are important when setting up MSE-API on tablet environment. The most important aspect is that all code for responding to messages must be setup before the application loads, so that other can see which messages are supported. The developers writing the initialization code realized this and setup their response handlers before the application finished its initial setup (see Figure 31).

```

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)
{
    self.mse = [MSEMultiSurface new];

    //Update the View Controller with MSE
    ViewController * viewController = (ViewController *) self.window.rootViewController;
    viewController.mse = self.mse;

    //Setup Routes
    [self setupRoutes:self.mse];

    return YES;
}

```

Figure 31: Initializing the API

7.4.2.2 Sending Extents (Tablet & Tabletop)

Another major feature in the application was sending the extent from the tablet device to the tabletop. In this feature the developers use the API correctly. As can be seen in Figure 32 the developers correctly check that the device is currently paired before sending out a dictionary. They use the serialization mechanism provided by the API to store the information regarding the extent and use the error blocks correctly. They are also able to deserialize the dispatched object correctly and load the layers as the appropriate UI element on C#.

7.4.2.3 Capturing Entities (Tablet & Tabletop)

One of the feature developed for on iOS, allows the tablet to capture the entities currently available on the tabletop. To accomplish this the developer must dispatch a message to the tabletop so he can determine information about the entities layer. This is subtly different the earlier features where the developer can just dispatch information to his target. In this feature the developer must query and retrieve information. For this feature the developers recognized this problem but used the convenience methods provided rather than the more advanced functionality. As can be seen in Figure 33 the developers send an empty dictionary to

```

- (IBAction)sendExtent:(id)sender
{
    if([self.mse.pairingRecognizer pairingState] == Paired) {
        [self.mse.locator devicesInView:^(NSArray *devices) {
            for (MSEDevice *device in devices) {

                NSMutableDictionary* dictionary = [self packExtentAndLayers];
                [self.mse sendDictionary:dictionary ofType:@"dictionaryExtent" toDevice:device withGesture:nil];

                [self showDeviceSentMessage:device.identifier];
            }
        } fail:^(NSError *error) {
            [self showErrorMessage:@"No Device Found"];
        }];
    } else {
        [self showErrorMessage:@"Must be Paired"];
    }
}

else if (dictionaryType.Equals("dictionaryExtent"))
{
    this.Dispatcher.Invoke((Action)(() =>
    {
        string[] layerURLS = dictionary["layerURLS"].Split(',');

        ScatterViewItem svi = CreateScatterView(dictionary["sentExtent"], layerURLS);
        WindowScatterView.Items.Add(svi);
    }));
}

```

Figure 32: Sending & Receiving Extents

the device being targeted. When this dictionary is received the entity information is then sent back to the device which sent the empty dictionary. A more advanced developer might have accomplished this by creating a route on the target device and responding to the request using the entity information. While the developers didn't use the advanced functionality they still completed the feature successfully.

7.4.3 Questionnaire

In addition to the time logs, developers were asked to complete a short questionnaire at the end of the iteration. This questionnaire asked general questions about their experiences with the API, such as how easy to use the API was compared with other APIs they had worked with in the past, how long it took them to become comfortable using the API what issues and problem they had while working with the API.

```
- (IBAction)receiveEntities:(UIBarButtonItem *)sender {
    // Code called when "Receive Entities" button is pressed
    if([self.mse.pairingRecognizer pairingState] == Paired)
    {
        [self.mse.locator devicesInView:^(NSArray *devices) {
            for(MSEDevice *device in devices)
            {
                // Create the request
                NSMutableDictionary *dictionary = [NSMutableDictionary new];

                // Send request to all devices found
                for(MSEDevice *device in devices)
                {
                    [self.mse sendDictionary:dictionary ofType:@"ReceiveEntities" toDevice:device withGesture:nil];
                    [self showDeviceSentMessage:device.identifier];
                }
            }
        } fail:^(NSError *error) {
            // Failure
            [self showErrorMessage:@"No Device Found"];
        }];
    }
    else {
        [self showErrorMessage:@"Must be Paired"];
    }
}

else if (dictionaryType.Equals("ReceiveEntities"))
{
    this.Dispatcher.Invoke((Action)(() =>
    {
        FeatureLayer entityLayer = MyMap.Layers["ScenarioFeatureLayer"] as FeatureLayer;
        string url = entityLayer.Url;

        Dictionary<string, string> entityDictionary = new Dictionary<string, string>();
        entityDictionary.Add("URL", url);

        mse.SendDictionary(entityDictionary, "ReceiveEntities", originDevice, null);
    }));
}
```

Figure 33: Requesting Entities

7.4.3.1 Experience & Language Issues

The development team did not have a great deal of experience using either the C# or Objective-C languages and their associated frameworks. Many only began using this languages as part of the C4I development project. As such several developers complained that they ran into difficulties using the language while trying to use the API. Developer 2 commented that they *“didn’t run into any difficulties other than with the Objective C language”*.

7.4.3.2 Feedback on Usability

All the developers using the API felt that it was easy to use. They generally claimed that they were able to start using the API quickly. Developer 1 commented that *“I felt comfortable using the API after the second time [work session]”* while Developer 4 said *“compared to other API’s that I have used in the past (i.e. ESRI’s API), the MSE API is far more straightforward”*.

Developer 3 commented that methods and objects exposed by the API were logical, saying that *“the different publicly facing functions are easy to understand and they do what you expect”*.

7.4.3.3 Feedback on Documentation & Examples

The developers were also asked to comment specifically about their experience with the documentation provided by MSE-API. The participants were broadly positive about the documentation but some participants felt that the documentation was not equally complete for both the two languages supported. On this topic Developer 2 said *“the Objective C and C# documentation was different, and one was more helpful than the other (the C# being the most helpful.”*.

7.4.3.4 Criticism of the API

When the developers were asked if they ran into any problems with the API they responded with some issues. One developer felt that the approach used for responding to messages which

contained a dictionary (i.e. a set of key-value pairs) was confusing. Developer 1 said *“I think I would change how you send a request to receive data because it is a bit weird to send a dictionary as if you are sending information to the device”*. Developer 2 pointed out that the serialization functionality included in the API is limited, saying *“I found it annoying to only be able to send strings in a dictionary element”*. By this they are referring to that fact that dictionaries sent between devices must have both the key and the value as a string and that no automatic conversion exists for other types or objects.

7.4.4 Discussion

Given the functionality of the system, we were encouraged to see how quickly the developers were able to begin using the API to implement features. While many of the features were straightforward and within the intended scope supported by the API, they were able to implement them very quickly. We argue that this is supporting evidence that the API is learnable and discoverable by developers. Further evidence is found in the feedback provided by the developers which was universally positive. Developers claimed to have quickly become comfortable using the API and their complaints focused generally on language issues and features they would like added to the API. Developers requested, for example, better support for serialization, support for additional platforms and support for 64 bit systems.

7.4.5 Threats to Validity

Several issues threaten the validity of the study. Because the features were derived from the needs of an industrial partner, it was not possible to carefully control what work was completed by whom. It's possible, therefore, that easier features were completed by stronger developers, causing the total time spent on the features to fall. Since the developers were co-workers and colleagues of the main developer of the API, it's possible that some conflict of interest occurred

in their qualitative feedback. This might have caused a positive bias in the qualitative results that were provided. The study itself was also limited in scope, providing a relatively small number of features which were themselves quite basic. Only a small number of developers participated in the study and each feature worked on was not comparable to the other features. This could mean that the results presented are not generalizable to a larger group of developers.

7.5 Conclusion

The main research question of this study has been to show that MSE-API is a learnable and discoverable API. To assess this, a team of developers completed several features in an ongoing application using MSE-API. The time spent completing these features was recorded and qualitative feedback was provided through a questionnaire. This team of developers was able to quickly add several useful features related to data transfer to their application in a relatively short period of time. In addition, the developers provided positive feedback on its usability and felt they were able to quickly become comfortable using the API. We feel this provides initial but not conclusive evidence that the API is learnable and discoverable by developers.

Chapter Eight: Conclusions

This thesis presents an API called MSE-API for building multi-surface applications with spatially augmented gestures. First, multi-surface systems were placed in their correct context, developing out of an environment with new types of devices and the desire for new types of interactions. Spatially-augmented gestures were explained, and their origin contextualized within previous research projects and elicitation studies. We then proposed that the lack of such multi-surface applications in the real world might be due to insufficient developer tools, specifically a lack of APIs to assist developers with tasks common to multi-surface systems. The requirements for such an API were explained, and our API, called MSE-API, was presented. In addition to meeting the requirements of the API, we wished for the API to be usable. That is, MSE-API must be both learnable and discoverable to novice developers while still being efficient for experienced developers. Two case studies were then presented: one conducted by the author as a self-evaluation and the other a traditional case study with independent developers to evaluate the usability of the API. Based on the evidence provided by these case studies, we feel that the API has met its goals regarding usability.

8.1 Thesis Contributions

The first contribution of this work was a careful literature review of previous work in the area of multi-surface systems. This review provides a clear path that shows how multi-surface systems developed, what core problems were encountered in this area, and how they gradually grew in complexity to involve more complex interactions, including spatially-augmented gestures. A smaller literature review collected previous work in the field of API usability, highlighting previous attempts to measure API usability and issues which have been described by previous researchers working in the area.

The next major contribution provided is MSE-API itself. This API meets all the major constraints and requirements documented in detail in Chapter 3, the API also provides a tool to configure a room dynamically with an arbitrary collection of devices. This is an important advance over existing APIs.

In addition to the API itself, several case studies were conducted providing evidence that the API is both learnable and discoverable while still allowing experienced developers to be efficient.

Taken together, this answers the major research goal presented in the Introduction. To our knowledge, this is the only API for supporting multi-surface systems with spatially-augmented gestures without requiring expensive specialized hardware.

8.2 Future Work

There are several directions for continuing work on MSE-API. The first involves improving the spatial engine, or the core components of the locator which determines which devices intersect with which other devices. Another major goal would be to achieve the fusion and integration of multiple sensors to expand the range and accuracy of the API. Finally the evaluation of the API could be expanded and used to guide the further development of the API.

8.2.1 Improving the Spatial Engine

The spatial engine of the API is responsible for computing the intersections between devices and determine what devices are in view when queried. The current system has several drawbacks.

Since it doesn't store the width and depth of objects in the room, they all appear to occupy the same physical space. This can cause issues if a device is extremely large, such as a wall display which encompasses the entire wall of a room. To address this issue, the way that spatial calculations are performed could be modified. Each device could be given an appropriate width and depth and calculations could be based on intersections with this object. This would have the

added benefit of computing an intersection point between the device and its target. This could be useful for animating or providing visual feedback to transfers.

8.2.2 Data Fusion

Another area of future work would be to incorporate additional sensors into the system. This would be especially useful if additional Kinect sensors could be fused together, as this would expand the range in which MSE-API works. In addition, other sensors such as the LEAP Motion could be incorporated to provide higher accuracy within a small area inside the larger area tracked by the Kinect.

8.2.3 Further Evaluation of the API

As the API is developed further to add more features, it would be useful to incorporate regular API evaluations to guide the design. New features and API changes should be evaluated by developers in usability studies in the same way that new features to a traditional application are evaluated by users. This could be done using traditional user studies and with case studies of development teams using the API as part of development practice. In order to conduct such studies it would be necessary that additional developers use the API. Building a community of developers using MSE-API is therefore also a major goal.

References

- [1] Brad Johanson, Shankar Ponnekanti, Casear Sengupta, and Armando Fox, "Multibrowsing: Moving Web Content across Multiple Displays," in *Proceedings of the Conference on Ubiquitous Computing*, Atlanta, USA, 2001, pp. 346-353.
- [2] Andruid Kerne, William A Hamilton, and Zachary O Toups, "Culturally based design: embodying trans-surface interaction in rummy," in *Proceedings of the Conference on Computer Supported Cooperative Work*, Seattle, USA, 2012, pp. 509-518.
- [3] James R Wallace, Stacey D Scott, Taryn Stutz, Tricia Enns, and Kori Inkpen, "Investigating teamwork and taskwork in single- and multi-display groupware systems," *Journal of Personal and Ubiquitous Computing*, vol. 13, no. 8, pp. 569-581, November 2009.
- [4] Jacob T Biehl et al., "IMPROMPTU: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and Its Field Evaluation for Co-located Software Development ," in *Proceeding of the Conference on Human Factors in Computing Systems*, Florence, Italy, 2008, pp. 939-948.
- [5] Stefan Bachl, Martin Tomitsch, Karin Kappel, and Thomas Grechenig, "The Effects of Personal Displays and Transfer Techniques on Collaboration Strategies in Multi-touch Based Multi-Display Environments," in *Proceedings of the Conference on Human-Computer Interaction*, Lisbon, Portugal, 2011, pp. 373-390.
- [6] Chris Burns et al., "Interpretative Visualization of Fused Hydrocarbon Microseep and Reservoir Data," in *Geoconvention*, Calgary, Canada, 2012.

- [7] Teddy Seyed et al., "MRI Table Kinect: A multi-surface application for exploring volumetric medical imagery," in *Proceedings of the Workshop on Safer Interaction in Medical Devices*, Paris, France, 2013.
- [8] Roswitha Gostner, Enrico Rukzio, and Hans Gellersen, "Usage of spatial information for selection of co-located devices," in *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, Amsterdam, Netherlands, 2008, pp. 427-430.
- [9] Teddy Seyed, Chris Burns, Mario Costa Sousa, Frank Maurer, and Anthony Tang, "Eliciting usable gestures for multi-display environments," in *Proceedings of the 2012 ACM international conference on Interactive tabletops and surfaces*, Cambridge, USA, 2012, pp. 41-50.
- [10] Samuel G McLellan, Alvin W Roesler, and Joseph T Tempest, "Building More Usable APIs," *IEEE Software*, vol. 15, no. 3, pp. 78-86, May 1998.
- [11] Jeffrey Stylos, "Making APIs More Usable with Improved API Designs, Documentation and Tools," Carnegie Mellon University, Pittsburgh, USA, Doctoral Thesis CMU-CS-09-130, 2009.
- [12] Jakob Nielsen and Jo Ann T Hackos, *Usability Engineering*. San Diego, USA: Academic Press, 1993.
- [13] Jeffrey Stylos and Brad A Myers, "The implications of method placement on API learnability," in *Proceedings of the Symposium on Foundations of Software Engineering*, Atlanta, USA, 2008, pp. 105-112.

- [14] Richard A Bolt, "'Put-that-there': Voice and gesture at the graphics interface," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, Seattle, USA, 1980, pp. 262-270.
- [15] Jim Wallace, Vicki Ha, Ryder Ziola, and Kori Inkpen, "Swordfish: user tailored workspaces in multi-display environments," in *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, Montreal, Canada, 2006, pp. 1487-1489.
- [16] Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood, "Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media," in *Proceedings of the 16th annual ACM symposium on User interface software and technology*, Vancouver, Canada, 2003, pp. 159-168.
- [17] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments," in *Proceedings of the 3rd international conference on Ubiquitous Computing*, Atlanta, USA, 2001, pp. 56-75.
- [18] Jacob T Biehl and Brian P Bailey, "ARIS: an interface for application relocation in an interactive space," in *Proceedings of Graphics Interface*, London, Canada, 2004, pp. 107-116.
- [19] Jacob T Biehl and Brian P Bailey, "Improving interfaces for managing applications in multiple-device environments," in *Proceedings of the working conference on Advanced visual interfaces*, Venezia, Italy, 2006, pp. 35-42.

- [20] Ken Hinckley, Gonzalo Ramos, Francois Guimbretiere, Patrick Baudisch, and Marc Smith, "Stitching: pen gestures that span multiple displays," in *Proceedings of the working conference on Advanced visual interfaces*, Gallipoli, Italy, 2004, pp. 23-31.
- [21] Ken Hinckley, "Synchronous gestures for multiple persons and computers," in *Proceedings of the 16th annual ACM symposium on User interface software and technology*, Vancouver, Canada, 2003, pp. 149-153.
- [22] Peter Tandler, Thorsten Prante, Christian Müller-Tomfelde, Norbert Streitz, and Ralf Steinmetz, "Connectables: dynamic coupling of displays for the flexible creation of shared workspaces," in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, Orlando, USA, 2001, pp. 11-20.
- [23] Norman Streitz et al., "i-LAND: an interactive landscape for creativity and innovation," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, Pittsburg, USA, 1999, pp. 120-127.
- [24] Jun Rekimoto, "Pick-and-drop: a direct manipulation technique for multiple computer environments," in *Proceedings of the 10th annual ACM symposium on User interface software and technology*, Banff, Canada, 1997, pp. 31-39.
- [25] Andrew Wilson and Hrvoje Benko, "Combining multiple depth cameras and projectors for interactions on, above and between surfaces," in *Proceedings of the 23rd annual ACM symposium on User interface software and technology*, New York, USA, 2010, pp. 273-282.

- [26] Raimund Dachsel and Robert Buchholz, "Natural throw and tilt interaction between mobile phones and distant displays," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, Boston, USA, 2009, pp. 3253-3258.
- [27] Nabeel Hassan, Md Rahman, Irani Pourang, and Peter Graham, "Chucking: A One-Handed Document Sharing Technique.," in *Proceedings of Interact*, Uppsala, Sweden, 2009, pp. 264-278.
- [28] Julian Seifert et al., "MobiSurf: improving co-located collaboration through integrating mobile devices and interactive surfaces," in *Proceedings of the 2012 ACM international conference on Interactive tabletops and surfaces*, Cambridge, USA, 2012, pp. 51-60.
- [29] Till Ballendat, Nicolai Marquardt, and Sault Greenberg, "Proxemic interaction: designing for a proximity and orientation-aware environment," in *ACM International Conference on Interactive Tabletops and Surfaces*, Saarbrücken, Germany, 2010, pp. 121-130.
- [30] Andrew Bragdon, Rob DeLine, Ken Hinckley, and Meredith Ringel Morris, "Code space: touch + air gesture hybrid interactions for supporting developer meetings," in *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, Kobe, Japan, 2011, pp. 212-221.
- [31] Nicolai Marquardt, Ken Hinckley, and Saul Greenberg, "Cross-device interaction via micro-mobility and f-formations," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*, Cambridge, USA, 2012, pp. 13-22.

- [32] Ekaterina Kurdyukova, Matthias Redlin, and Elisabeth André, "Studying user-defined iPad gestures for interaction in multi-display environment," in *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*, Lisbon, Portugal, 2012, pp. 93-96.
- [33] Bonifaz Kaufmann, Martin Grazer, and Martin Hitz, "A Service-Oriented Mobile Multimodal Interaction Framework," in *Proceedings of the Workshop on infrastructure and design challenges of coupled display visual interfaces (PPD'12)*, Capri, Italy, 2012.
- [34] B Johanson, A Fox, and T Winograd, "The Interactive Workspaces project: experiences with ubiquitous computing rooms," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 67-74, Apr-Jun 2002.
- [35] Aiman Erbad, Michael Blackstock, Adrian Friday, Rodger Lea, and Jalal Al-Muhtadi, "MAGIC Broker: A Middleware Toolkit for Interactive Public Displays," in *Proceedings of the conference on Pervasive Computing and Communications*, Hong Kong, 2008, pp. 509-514.
- [36] Andy Wu, Sam Mendenhall, Jayraj Jog, Loring Scotty Hoaq, and Ali Mazalek, "A nested API structure to simplify cross-device communication," in *Proceedings of the Conference on Tangible, Embedded and Embodied Interaction*, Kingston, Canada, 2012, pp. 225-232.
- [37] Roy Fielding, "Architectural styles and the design of network-based software architectures," University of California, Doctoral Thesis 2000.
- [38] Barry Brummit, Brian Meyers, John Krumm, Amanda Kern, and Steven A Shafer, "EasyLiving: Technologies for Intelligent Environments," in *Proceedings of the*

- International Symposium on Handheld and Ubiquitous Computing*, Bristol, UK, 2000, pp. 12-29.
- [39] John Krumm and Ken Hinckley, "The NearMe Wireless Proximity Server," *UbiComp 2004: Ubiquitous Computing*, pp. 283-300, 2004.
- [40] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi, "Methods towards API Usability: A Structural Analysis of Usability Problem Categories," in *Human-Centered Software Engineering*, Marco Winckler, Peter Forbrig, and Regina Bernhaupt, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2012, pp. 164-180.
- [41] Martin P Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, vol. 26, no. 6, pp. 27-34, November 2009.
- [42] Hou Daqing, "Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions," in *Proceedings of the Conference on Program Comprehension*, Kingston, Canada, 2011, pp. 91-100.
- [43] Minhaz F Zibran, "What Makes APIs Difficult to Use? ," *International Journal of Computer Science and Network Security*, vol. 8, no. 4, pp. 255-261, April 2008.
- [44] Chris Burns, Jennifer Ferreira, Theodore D Hellmann, and Frank Maurer, "Usable results from the field of API usability: A systematic mapping and further analysis," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computer*, Innsbruck, Austria, 2012, pp. 179-182.

- [45] Minhaz Fahim Zibran, "Useful, But Usable? Factors Affecting the Usability of APIs," in *Proceedings of the Working Conference on Reverse Engineering*, Limerick, Ireland, 2011, pp. 141-155.
- [46] Daniel Ratiu and Jan Jurjens, "Evaluating the Reference and Representation of Domain Concepts in APIs," in *Proceedings of the IEEE Conference on Program Comprehension*, Amsterdam, Netherlands, 2008, pp. 242-247.
- [47] Marques Luiz Afonso, F. de G. Renato Cerqueira, and Clarisse Sieckenius de Souza, "Evaluating application programming interfaces as communication artefacts," in *Proceedings of the Psychology of Programming Interest Group*, London, UK, 2012, pp. 8-31.
- [48] Alan F Blackwell, Luke Church, and Thomas Green, "The Abstract is 'an Enemy': Alternative perspectives to computational thinking," in *Proceedings on the Workshop of the Psychology of Programming Interest Group*, Lancaster, UK, 2008.
- [49] Thomas Scheller and Eva Kuhn, "Influencing Factors on the Usability of API Classes and Methods," in *Proceedings the Conference on Engineering of Computer-Based Systems*, 2012, 2012, pp. 232-241.
- [50] Brad Ellis, Jeffrey Stylos, and Brad Myers, "The factory pattern in API design: A usability evaluation," in *Proceedings of the Conference on Software Engineering*, Minneapolis, USA, 2007, pp. 302-312.

- [51] Brian Ellis, Jeffrey Stylos, and Brad Myers, "The Factory Pattern in API Design: A Usability Evaluation," in *Proceedings of the Conference on Software Engineering*, Minneapolis, USA, 2007, pp. 302-312.
- [52] Jeffrey Stylos and Brad Myers, "Mapping the Space of API Design Decisions," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, Coeur d'Alene, USA, 2007, pp. 50-60.
- [53] Martin P Robillard and Robert DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703-732, December 2011.
- [54] Andrew J Ko and Yanne Riche, "The Role of Conceptual Knowledge in API Usability," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computer*, Pittsburgh, USA, 2011, pp. 173-176.
- [55] Brad A Myers et al., "Studying the Documentation of an API for Enterprise Service-Oriented Architectures," *Journal of Organization and End User Programming*, vol. 22, no. 1, pp. 23-51, January 2010.
- [56] Janet Nykaza et al., "What programmers really want: results of a needs assessment for SDK documentation," in *Proceedings of the Conference on Computer Documentation*, Toronto, Canada, 2002, pp. 133-141.
- [57] Seyed Mehdi Nasehi and Frank Maurer, "Unit Tests as API Usage Examples," in *Proceedings of the Conference on Software Maintenance*, Timișoara, Romania, 2010, pp. 1-10.

- [58] Raymond P.L Buse and Westley Weimer, "Synthesizing API usage examples," in *Proceedings of the Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 782-792.
- [59] Lee Wei Mar, Ye-Chi Wu, and Hewijin Christine Jiau, "Recommending Proper API Code Examples for Documentation Purpose," in *Proceedings of the Asia-Pacific Conference on Software Engineering*, Saigon, Vietnam, 2011, pp. 331-338.
- [60] Daniel S Eisenberg, Jeffrey Stylos, and A Brad Myers, "Apatite: a new interface for exploring APIs," in *Proceedings of the Conference on Human-Factors in Computer Systems*, Atlanta, USA, 2010, pp. Apatite: a new interface for exploring APIs.
- [61] Jeffrey Stylos, Brad A Myers, and Zizhuang Yang, "Jadeite: improving API documentation using usage information," in *Proceedings of Extended Abstracts on Human Factors in Computer Systems*, Boston, USA, 2009, pp. 4429-4434.
- [62] Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Stylos, and Brad A Myers, "Usability challenges for enterprise service-oriented architecture APIs," in *Proceedings of the Symposium on Visual Language and Human-Centric Computing*, Herrsching am Ammersee, Germany, 2008, pp. 193-196.
- [63] Jeffrey Stylos et al., "A Case Study of API Redesign for Improved Usability ," in *Proceedings on Visual Languages and Human-Centric Computing*, Pittsburgh, USA, 2008, pp. 189-192.

- [64] Umer Farooq and Dieter Zirkler, "API peer reviews: a method for evaluating usability of application programming interfaces," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, Savannah, USA, 2010, pp. 201-210.
- [65] Thomas Green and Marian Petre, "Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131-174, June 1996.
- [66] Steven Clarke and Curtis Becker, "Using the Cognitive Dimensions Framework to evaluate the usability of a class library," in *Proceedings of the First Joint Conference of EASE PPIG*, Keele, UK, 2003, pp. 359-366.
- [67] Steven Clarke, "Measuring API Usability," *Dr. Dobbs Journal*, vol. 29, pp. 6-9, January 2004.
- [68] J. Gerken, H. Jetter, M. Zöllner, M Mader, and H Reiterer, "The concept maps method as a tool to evaluate the usability of APIs," in *International Conference on Human Factors in Computing Systems, 2011*, Vancouver, BC, 2010, pp. 3373-3382.
- [69] Manuel Bertoa, José Troya, and Antonio Vallecillo, "Measuring the usability of software components," *Journal of Systems and Software*, vol. 79, no. 3, pp. 427-439, March 2006.
- [70] Cleidson R. B. deSouza and David L. M. Bentolila, "Automatic Evaluation of API Usability using Complexity Metrics and Visualizations," in *Proceedings of the Interational Conference on Software Engineering*, Vancouver, Canada, 2009, pp. 299-302.