Supporting Ad-hoc Tangible Interaction

Research Dissertation for the Degree of Doctor of Philosophy

James Andrew Walsh
Bachelor of Computer and Information Science (Honours)

Supervisor

Bruce H. Thomas

Associate Supervisor

Stewart von Itzstein

Adelaide, South Australia
May 2014

Wearable Computer Lab
School of Information Technology and Mathematical Sciences
Division of Information Technology, Engineering, and the Environment
University of South Australia

Contents

List of Figures

List of Tables

List of Code Excerpts

Glossary

Ad-hoc system	the ad-hoc system under development
AH-Data	ad-hoc data manipulation
AH-Logic	ad-hoc logic manipulation
AH-UI	ad-hoc user interfaces
AR	augmented reality
DOF	degrees of freedom
EUI	embodied user interface
External system	existing, fixed systems outside of any ad-hoc system
Fiducial marker	physical marker with embedded information readable by machines
GUI	graphical user interface
HCI	human computer interaction
IPC	inter-process communication
MCRit	Model, Control, Representation (intangible, tangible) architecture
MVC	Model, View, Controller architecture
NUI	natural user interface
Nullary	a function taking and returning no arguments
OUI	organic user interface
PBD	programming by demonstration
PBE	programming by example
RGBD	red, green, blue, and depth information
SAR	spatial augmented reality
TUI	tangible user interface

Abstract

We all utilise the physical affordances of everyday objects as part of our interactions with other people and systems. Interactive systems or desired functionality can be externalised by the user from an internal mentally-tracked system, to a physical system. This allows different objects to be assigned roles and interactions, playing out some kind of ad-hoc system as required by the user's current context. When paired with feedback technologies such as spatial augmented reality, these systems can become more immersive. As such, previous work by Underkoffler and Ishii has identified the possibility of giving additional meaning and life to inert objects as being both cognitively powerful and intriguing.

Despite the advantages of tangible interaction and the promise of ubiquitous computing, previous digital tangible systems have been purpose built, limiting their application and extensibility for end users. No matter the developer's expertise, they cannot foresee every application of a system. As such, the ability for systems to be extensible for end users without developer experience is appealing. If users can create interactive tangible systems as required, it will result in a more immersive user experience and provide a richer collaborative environment. The ability for end users to be able to program systems using their normal interactions as a means of extending functionality is thus appealing, but until now has not been explored in a tangible context.

This dissertation outlines the development of a software architecture, called Tangible Agile Mapping (TAM), to support the user in developing extensible, ad-hoc, interactive tangible systems. The architecture designed is independent from any specific system implementation (however example implementations are described), and provides a means to represent interactive systems which can easily be extended at runtime without authoring any code. TAM defines a number of different system components and their relationships that define the abstraction and compartmentalisation of the different required application functions. External systems can be easily integrated into the architecture at both design and run time, allowing the implemented system to provide extensibility for user's individual environments and use cases. External application functions appear as if they were native functions of the tangible ad-hoc system, allowing for transparent system integration and usage.

Through previous work and pilot studies examining how users create ad-hoc functionality in a tangible context (as well as the types of interactions used in them), three iterations of development occurred, examining end user extensibility for:

- the manipulation of application functions,
- the manipulation of application data, and
- the manipulation of application logic.

The resulting conceptual architecture is thus described as it relates to these three classes of interaction. Following this, corresponding implementations of the architecture and example applications for such functionality are presented, allowing for ad-hoc extensibility and manipulation by end users. A formal user evaluation of an implementation supporting the creation of ad-hoc logic-based systems is then presented. To evaluate the level of support the architecture has for different tangible systems, the architecture is evaluated against existing tangible classification frameworks, showing that by supporting these three classes of functionality (logic, function control, and data control), comprehensive ad-hoc systems can be developed.

Declaration

I declare that:

- this dissertation presents work carried out by myself and does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any university;
- to the best of my knowledge it does not contain any materials previously published or written by another person except where due reference is made in the text; and
- all substantive contributions by others to the work presented, including jointly authored publications, is clearly acknowledged.

James Walsh
Adelaide, May 2014

Bruce Thomas
Adelaide, May 2014

Stewart von Itzstein
Adelaide, May 2014

Acknowledgements

When I thought about writing an acknowledgements section, I realised just how many people have supported me and my technological curiosity over the years. Regardless of the outcome of this PhD, I am grateful for their never ending support in my endeavours.

I want to thank my supervisor, Professor Bruce Thomas, and associate supervisor, Doctor Stewart von Itzstein for their help and support. Bruce has been a great supervisor, helping me identify a number of traits in my workflow and provide guidance regarding how I can best leverage them, whilst also adapting his supervisory style and support to suit them, all the while helping me with my research. Whilst I have not always grasped what he was trying to get at immediately, everything eventually 'clicked', and I could see he was right from the start. Stewart has been a great help to bounce ideas off of and work through problems in development, always helping me to take a step back and look at things from a clearer, more objective perspective. Stewart always helped me to see a bigger picture. I would also like to thank my reviewers, Dr. Maki Sugimoto and Dr. Henry Gardner, for providing their thoughts and criticisms from an external viewpoint that have helped to strengthen this work to what it has finally become.

Thanks goes out to all the members of the WCL (past and present). It has been a great place to work over the years. I would especially like to acknowledge and thank Thuong Hoang, Ben Close and Michael Marner who always helped me out with whatever questions I had, especially as a ‘Windows guy’ living in a Linux lab.

I would like to thank John Roccisano for his guidance and mentorship in over the years, both in life and technology. John helped me to understand the work-life balance and to focus on the bigger things in life, whilst also being someone who would always be interested in and support whatever piece of technology I was working on (both in time and money).

I would like to thank my friends for putting up with my geeky endeavours and always placating me whenever I have to show off my latest project.

I would like to thanks my parents, Gay and John, for their endless love and support over the years. I have no doubt had I grown up in a different environment, not only without the technology at home, but without parents who supported and nurtured my obsession

with it, I would not have reached this level. Your influences in my life have forever shaped how I perceive, interact and contribute to this world.

Finally, I'd like to thank my wife, Rosie, for putting up with me all these years despite being perhaps a little bit too obsessed with technology, just because she likes the way my "face lights up" when I'm playing with something new. I know I could not put up with someone like me, so thank you for being a part of my life.

List of Publications

Parts of this dissertation have previously been peer-reviewed and appeared in the following publications:

- Walsh, J. A., von. Itzstein, S., Thomas, B. H. (2013). Tangible Agile Mapping: Ad-hoc Tangible User Interaction Definition. Australasian User Interface Conference. R. Smith and B. Wuensche. Adelaide, Australia, CRPIT. 139**:** 3-12.

- Walsh, J. A., von. Itzstein, S., Thomas, B. H. (2014). Ephemeral Interaction Using Everyday Objects. Australasian User Interface Conference. B. Wuensche and S. Marks. Auckland, New Zealand, CRPIT. 150**:** 29-38.
  ***** Awarded Best Paper AUIC 2014 *****

- Marner, M. R, Smith, R. T., Walsh, J. A., Thomas, B. H. (2014). Spatial User Interfaces for Large Scale Projector-Based Augmented Reality. IEEE Computer Graphics and Applications. To appear.

Chapter 1. Introduction

> *"THE PROPOSITION OF GIVING ADDITIONAL MEANING AND ANIMATE LIFE TO ORDINARY INERT OBJECTS IS A COGNITIVELY POWERFUL AND INTRIGUING ONE" (UNDERKOFFLER AND ISHII, 1999)*

All of us utilize everyday objects to convey information about some mental 'cognitive system', as a means of reducing errors compared to when users simulate the system mentally (Myers, 1992). If a teacher were explaining the reactions between different chemicals, they might pick up different objects to represent elements, moving them closer together to indicate a reaction, changing the chemicals' state and showing additional information about the reaction. Despite the simple rules for these interactions, complex configurations can easily be created, and as such, tracking these roles, states, and how these items interact with a digital system introduces a cognitive overhead for both primary and collaborative users. In addition, the integration of external information or functionality based on ad-hoc (short term, improvised) interactions is currently limited based on the skills and available time of the user. Despite these limitations, there is a need for these kinds of short term, ephemeral interactions that leverage spatial intelligence and interactions, enabling a deeper, more natural and 'as-required' method of digital interaction.

This dissertation explores how to support tangible, ad-hoc and ephemeral interactions, through the creation of a software architecture to enable the run-time creation of these interactions by end users. This dissertation explores three discrete forms of tangible interactions:

- ad-hoc UI controls,
- ad-hoc manipulation of application data, and
- ad-hoc manipulation logic.

Three discrete implementations were created, each focusing on the different aspects of ad-hoc interaction as above. However, they are only to serve as examples that demonstrate the core concepts and capabilities of the architecture. Wider applications exist within the ubiquitous application space.

Aside from using physical objects as proxies, physical controls also provide a number of unique affordances when compared to pure digital interaction; enabling tactile feedback,

the ability to leverage spatial intelligence and multi-tasking, whilst also supporting collaborative interactions. Tangible User Interfaces (TUIs) allow us to leverage these affordances of objects as part of our interactions with a system (Ullmer, 2002). However, despite the promises of TUIs and the opportunity for creating ad-hoc interactions, previous tangible systems have been written from scratch for a specific, fixed use case. As such, this dissertation presents a software architecture and example implementation with associated interaction techniques to support the ad-hoc, run time definition of novel TUIs without requiring end users to author code.

Despite the promises of tangible interaction and the recent developments in tracking systems, there remains a distinct separation between those who not only have the skills, but also the time and finances to develop tangible systems, and those who will actually benefit the most from using them, and will most likely have the most domain specific knowledge (Hutchins et al., 1985). In the early days of computing, programmers wrote user-specific software which was modified to suit the specific idiosyncrasies of each use case by in-house developers (Halbert, 1984). Developers were cheap, computers were not. General purpose, commodity software now means users can purchase a system off the shelf with the expectation the software will do what they want. The problem is, they no longer have the developer to customise it, only themselves, as the end user is not always a software developer (Halbert, 1984), creating a gap between the creation and the utilisation of systems.

Previous research in Graphical User Interfaces (GUIs) has explored manually adaptable and automatically adaptive user interfaces, where the end user can adapt the system to match the current task and environmental context. Despite all of us using physical objects as proxies as part of our everyday interactions, there remains no way for a system to support the user for these kinds of ad-hoc physical interactions. The user already has the knowledge regarding how they want to interact with the system and what they want the system to do. It is up to the system to decipher what the user is wanting to achieve. As summarised by Paley (1998), “they [users] know what they're trying to do, and it's my job to build tools to help the computer figure it out”. Despite all the required technology currently available to users (and even already present in many living rooms as with the Microsoft Kinect[1] depth sensor), users cannot just pick up an object, place the object on a table, and use it to interact with a system.

[1] http://www.xbox.com/en-US/kinect

We are at a turning point in human computer interaction (HCI), where the previous transition from command-line interactions to graphical interfaces is now moving towards natural user interfaces (NUIs) (utilising touch, gesture, voice, etc.) and continuous NUIs (where NUIs and interactions are not system-specific or limited between devices) (Petersen and Stricker, 2009). HCI interaction is becoming more intuitive, more natural, and as a result, more multimodal. However, even with this progression towards more natural interaction, previous tangible systems have only been designed for a single, specific use case, and do not allow the user to either extend their functionality or change how they are controlled (let alone support new modes of interaction), limiting their capability for real world use. In addition, previous systems have seen tangible interaction as being a distinct, separate part of the user's digital interaction. No longer is this the case. If anything, by TUIs pervasively bringing the digital world into the real world, systems must acknowledge and interact the other digital systems that also exist in that world. Whilst systems and frameworks have been developed that lower the level for developing basic reconfigurable systems (such as Fails and Olsen (2002), Marquardt et al. (2011), and Hardy and Alexander (2012)), they are focused on creating systems to support the developers, not the end users. There still remains distinct development and usage stages in the system's lifecycle, with any changes to the system requiring the user to go back to the development stage, implement the change, and then resume normal use. Despite the regular occurrence and benefits of ad-hoc interactions, there is currently no way to leverage/integrate digital functionality. Ideally, the user could author new content and functionality using their normal tangible interactions with the system, essentially *authoring by interaction*. This investigation seeks to enable such functionality.

1.1 Motivation

Despite the affordances offered by TUIs, before tangible objects can be used, as either physical proxies or controls, a dedicated interaction system must first be developed to support that specific use case. This distinct separation of incorporating new objects and functionality and using the final system does not support the natural work flows occurring with these kinds of interactions, where people walk into a room and want to begin interacting with the system immediately, such as described by Coutaz (2007). In addition to this, the majority of users, and thus those standing to benefit the most from tangible interaction, are those who do not have the ability to create them. One of the barriers of TUIs for wider adoption is support for their use outside dedicated, designed-from-scratch

systems, lowering the threshold for extension and adoption. *There are no generic tangible applications available*. As a result, there are few tangible systems in everyday use.

For these reasons, a new paradigm is required where new functionality and controls can be authored within the system at run time, allowing the user to directly author new functionality and control system behaviour based on the current task and context. The user should be able to describe new functionality to the system, incorporating that functionality as a primary means of using the system. These techniques need to be simple, fast, and flexible, and adapt to the individual workflows of the user in the current context.

1.2 Research Questions

This dissertation addresses the need to extend support for the creation of TUIs outside of their current, purpose built, non-extensible scope. The work presented in this dissertation began with the following question:

- How can users introduce previously unknown TUIs and use them to effectively interact with the system?

In order to address this question, three iterations of research were conducted, focusing on supporting the different types of functionality required for ad-hoc interaction:

- manipulating the user interface,
- manipulating the application data, and
- manipulating the application logic.

Specifically, this dissertation answers the following questions:

- If there is a known set of application functions, how can a system support the ad-hoc creation of tangible user interfaces to control those functions?
- How can systems leverage ad-hoc interaction to support interacting with application data in a tangible context?
- How can systems support the ad-hoc definition of interactive systems using only the associated logic?

Given the focus of this research on ad-hoc interactions, the primary goal is the definition of new systems and functionality (i.e. defining the UI or the logic), rather than editing existing ones.

1.3 Research Goals

As a general goal, the focus of this investigation has been on creating a generic architecture, removing the existing distinct development and usage stages, and enabling

a more fluid method of supporting tangible ad-hoc interaction that can be utilised by a wider audience. That said, support for tangible interactions should obviously still be incorporated into systems as early as possible as a distinct goal. However, as the original developer cannot always envision every possible use case required (Stuerzlinger et al., 2006), there will always remain a need for end user extensibility, enabling:

- the ad-hoc manipulation of a tangible UI (AH-UI),
- the ad-hoc manipulation of data (AH-Data), and
- the ad-hoc manipulation of logic (AH-Logic).

This is especially true since the user, and not the designer, is the domain expert (Hutchins et al., 1985), and thus may have additional ideas for application functions (McDaniel, 1999).

1.4 Contributions

This dissertation presents a number of contributions in the field of tangible and ad-hoc user interfaces. The contributions are as follows:

- An extensible architecture to support generalised ad-hoc interactions, identifying the isolation and encapsulation of key components and their roles as they relate to the larger requirements.
- A working system supporting the run-time creation-of and interaction-with physical ad-hoc controls from passive materials.
- Individual implementations of the architecture demonstrating different aspects of the ad-hoc functionality across the manipulation of functions, data, and logic.
- Evaluations of ad-hoc systems by end users, showing that end users are capable of effectively using such systems.

This dissertation focuses on extending support for tangible systems outside of dedicated, fixed systems. The applications of this research have only begun to be explored using the implementations, example systems, and use cases presented, however such concepts represent real world applications for the research.

1.5 Dissertation Structure

The remainder of this dissertation is structured as follows. Chapter 2 provides a discussion of relevant literature, covering user interface theory (including tangible, ephemeral and reconfigurable interfaces), ad-hoc and reconfigurable interfaces, along with the psychology of interaction and briefly, augmented reality (AR). Chapter 3 provides a description of the outcome of this work as the theoretical framework to support ad-hoc

interaction, in context with the high level vision of this investigation. Chapter 4 describes two studies designed to explore how users would explore and utilise ad-hoc systems in everyday contexts. Following the development of systems utilising the architecture and the exploration of how users would utilise them, Chapter 5 provides a description of a system to support ad-hoc UI creation (AH-UI), Chapter 6 describes a system to support ad-hoc data manipulation (AH-Data), and Chapter 7 describes a system to support the definition of ad-hoc logic (AH-Logic) by end users, including an evaluation regarding the feasibility of enabling AH-Logic for end users. Chapter 8 provides a discussion of this investigation, identifying the features, limitations, and how the proposed architecture relates to and supports previous work in the tangible area. Finally, Chapter 9 provides conclusion discussing final thoughts and future work.

Chapter 2. Background

This chapter provides a summary of previous work as it relates to this dissertation. The chapter begins with an overview of tangible user interfaces, outlining their origin, development and notable applications. Reconfigurable interfaces are then discussed, outlining different approaches utilising hardware and software as well as systems that support programming by demonstration (PBD). Existing architectures for supporting different types of graphical and tangible user interfaces are explored, as is their relevance towards supporting ad-hoc interaction. The intrinsic psychological component of HCI is then discussed, outlining key theories defining how users interact with adaptable systems. A brief overview of augmented reality is then provided as it relates to providing feedback for tangible systems. The chapter concludes with a discussion, placing the work presented in this dissertation in-situ amongst the previous work. Whilst the use of commercial tracking systems is an underlying technology used in later chapters, this dissertation does not purport to make any contribution to the tracking field, and thus is not discussed in this chapter.

2.1 Tangible User Interfaces

As previously mentioned in the Introduction, recent advances have shown HCI is at a turning point. The transition from command-line interfaces to Graphical User Interfaces (GUIs) – utilising windows, icons, menus and pointers (WIMP) – has extended into Natural User Interfaces (NUIs) (using touch, gesture, etc.) and even Continuous NUIs, where user interactions are not distinct between devices (Petersen and Stricker, 2009).

Given TUIs utilise a more natural form of interaction using direct manipulation (Shneiderman, 1983), they innately support computer-support cooperative work (CSCW) (Grudin, 1994) for digitally assisted collaboration. Weiser (1991) presented ubiquitous computing as "the computer for the 21st century", where digital systems, whilst present in all aspects of the environment, fade into the background, passively supporting the user in their various tasks within that environment. As such, TUIs are just one realisation of the ubiquitous computing vision (Edge, 2008), where computation becomes more 'real', with the task becoming the user's focus, and not the digital system being used to support the task (as is currently the case). In the case of a GUI, the screen is the sole window into the digital world, whereas bringing the UI into the physical realm with TUIs enables us to physically exist within and interact with the interface itself (Hornecker and Buur,

2006). “Rather than pull people into the virtual world of the computer, we are trying to pull the computer out into the real world of people” (Brooks, 1997).

What is essentially the ultimate user interface is described by Ishii and Ullmer (1997) as Tangible Bits, and later as Radical Atoms (Ishii et al., 2012), similar to earlier themes from Weiser and Brown (1996) on ‘calm’ computing. In the ultimate user interface, materials can dynamically change form and appearance, enabling physical matter to be as dynamic as a pixels on a screen, truly giving physical form to digital data. Such interfaces would allow users to leverage the innate human skills present for spatial manipulation, something previous input systems have not been able to fully utilise.

In search of this interface, physical interaction with digital data has been explored. The Bricks system (Fitzmaurice et al., 1995) was the first to create a distinction for such interfaces, labelling them as Graspable UIs. Graspable UIs presented a number of advantages in its design philosophy. In using Bricks, the user could use two physically tracked ‘bricks’ to interact with digital content, for example using the bricks as anchors to directly deform digital content (Figure 1). Bricks encouraged bi-manual and parallel interaction whilst heavily leveraging the user’s spatial reasoning and offering support for collaborative interactions.

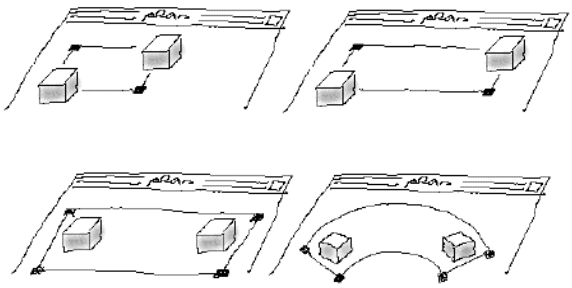

Figure 1 Using two bricks to scale an object (top) and using two bricks for object transformation (bottom) (Fitzmaurice et al., 1995), reprinted with permission from ACM

Ishii and Ullmer (1997) later explored Graspable UIs as Tangible User Interfaces, defining them as any UI that utilises physical objects, spaces and surfaces as a coupled interface to digital information. Given tangible manipulations directly result in changes to the real world, the merging of system input with the system output creates a form of Embodied UIs (EUIs) (Fishkin et al., 1999), where the system input is the output. An abacus would be the clearest example of such an interface, where the input is in itself the output. Such a direct association of digital content with physical objects is clearly demonstrated by the Marble Answering Machine (Crampton Smith, 1995), where

received messages are associated with physical marbles, that can be placed in an associated cradle to be played. This was a precursor to the MediaBlocks system (Ullmer et al., 1998), using digitally tagged blocks as a primary means of moving, copying and manipulating digital information. This use of 'tokens' was later explored by Holmquist et al. (1999), identifying three main types of representation in TUIs: tokens, tools, and containers.

Exploration of the TUI space was incorporated into the Digital Desk (Wellner, 1991) as a means of system interaction whilst sitting at a desk, where digital functionality (e.g. a calculator) was projected onto tangible elements, such as papers on the desk. As part of defining TUIs, the MetaDesk system (Ullmer and Ishii, 1997) enabled physical objects to be used as proxies for real world buildings on a projected map. Moving the objects around allowed the user to directly manipulate (translate/scale/warp) the intangible map. As part of their exploration of tangible interaction, they proposed a number of equivalencies between traditional GUI control elements and their TUI counterparts:

- Lenses as a replacement for GUI windows,
- Phicons (physical icons) as replacement for icons,
- Trays which have multiple slots for items as a menu replacement,
- Phandles (physical handles) using a physical object as a control handle, and
- Instruments as purpose-designed devices for replacing GUI controls.

Leveraging TUIs as a more natural method of interaction, Hinckley et al. (1994) used a prop-based interface for neurological visualisation. Using a six degrees-of-freedom (DOF) tracked doll's head and a clear acrylic plane, a surgeon could visualise medical scans (on an external monitor) as if they were slicing through the patient's skull at the location of the acrylic as relative to the dolls head (Figure 2).

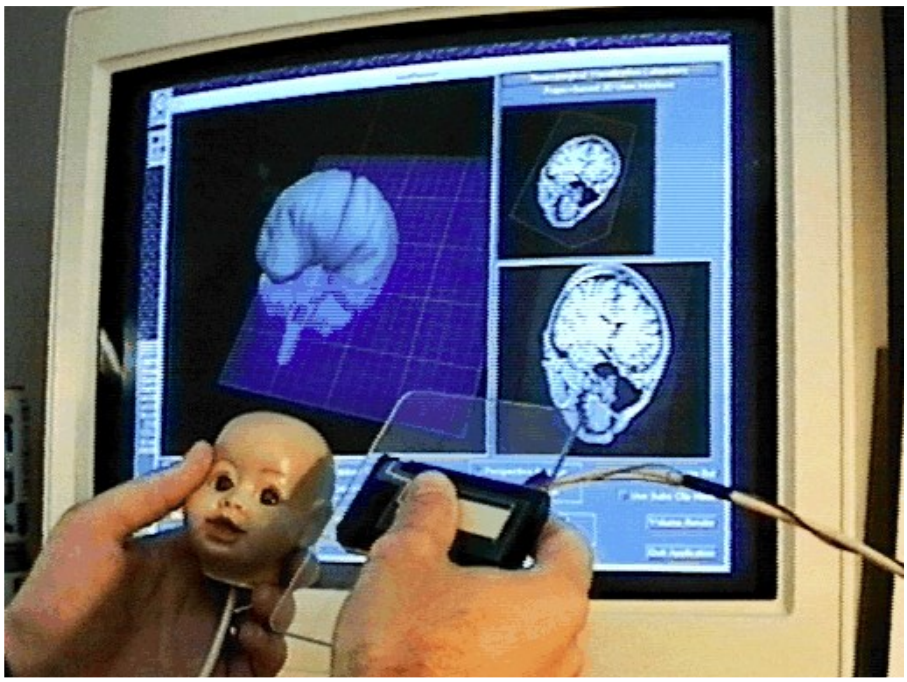

Figure 2 User creating a cutting plane on a human brain using physical props (Hinckley, 1996), reprinted with permission from Ken Hinckley

Illuminating Light (Underkoffler and Ishii, 1998) explored different optical holographic configurations using a virtual system (Figure 3). Different fiducially tracked blocks were used to reflect, split or join a projected light beam, allowing novice users to rapidly explore different configurations at little cost and risk (as opposed to using real and expensive holographic equipment).

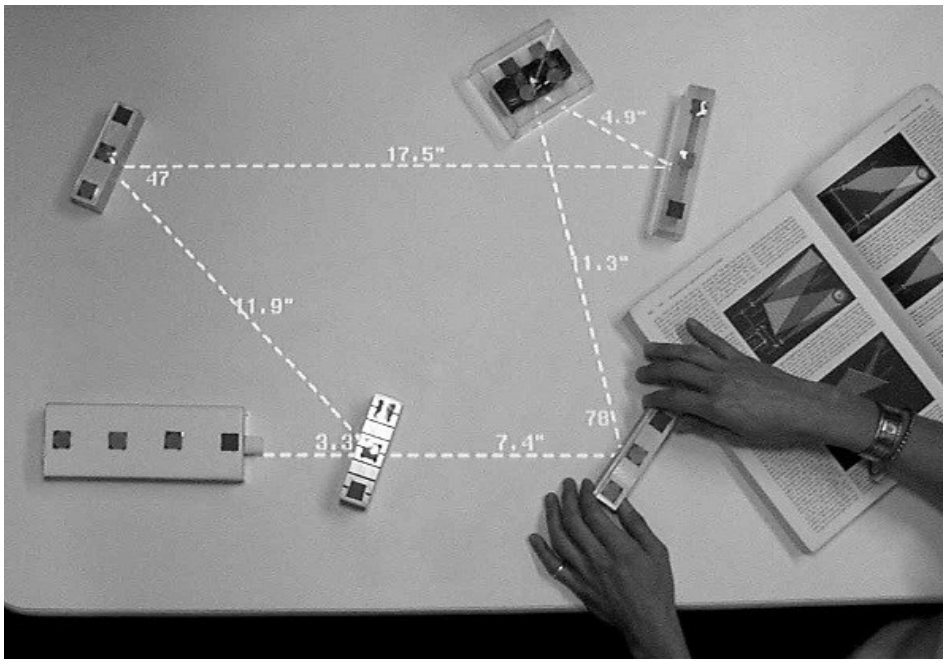

Figure 3 Demonstration of Illuminating Light to aid holography (Underkoffler et al., 1999), reprinted with permission from ACM

Such an approach was extended in the Urban Resource Planning (URP) system (Underkoffler and Ishii, 1999), where scale models of buildings could be placed on the table relative to one another, with realistic shadows, reflections, traffic, and fluid simulations projected that took into account the current configuration of buildings (Figure 4). Interaction with the digital properties of an object was supported with a dedicated 'double-sided' tangible wand, allowing the user to change a building's surface from brick to glass (affecting reflections) by tapping the respective end to the surface of a building, demonstrating the use of physical tools to edit virtual attributes. Shadows and reflections were controlled by a virtual sun, controlled with an analogue clock (as a metaphor for time of day). Fluid simulations could be visualised on the table, showing how winds and pressure systems would be affected by the given layout.

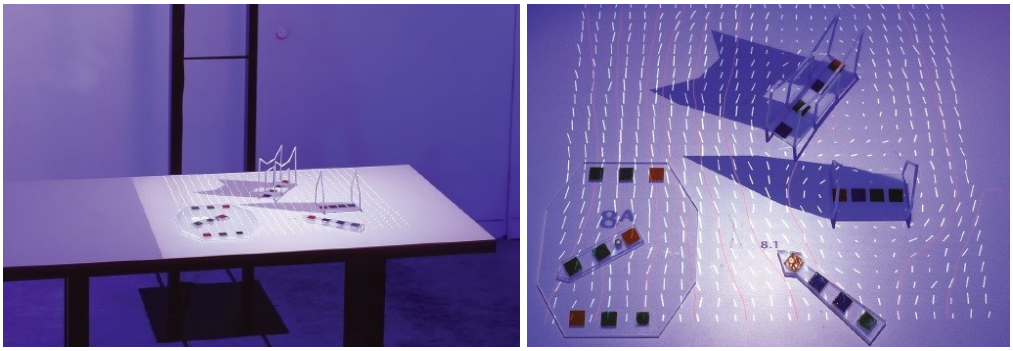

Figure 4 URP tabletop show shadows and reflection system (Ishii et al., 2012), reprinted with permission from IEEE (© 2012 IEEE)

The use of physical props allowed a more direct and natural way for users to interact whilst also seeing the results presented alongside the input controls in the real world. As part of URP, a continuum was identified defining an object as a pure object to a completely reconfigurable tool (Figure 5), with objects becoming more abstract as they move left-to-right.

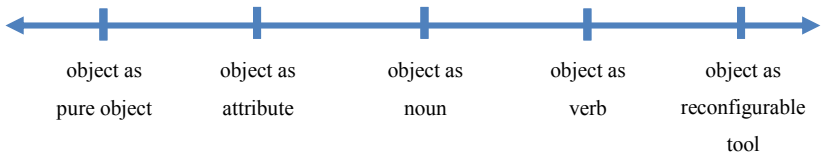

Figure 5 Continuum of an object as pure object versus object as a reconfigurable tool, reproduced from Underkoffler and Ishii (1999)

Continuing this urban design focus, Illuminating Clay (Piper et al., 2002), and later SandScape (Ishii et al., 2004), looked at terrain modelling. Using a depth camera and projector mounted above a table allowed users to sculpt terrain using either clay (Figure 6) or sand (possibly using system guidance from the projector) and then watch fluid simulations operate on the various paths and troughs in the terrain.

Figure 6 Modelling with Illuminating Clay (Piper et al., 2002), reprinted with permission from ACM

Collaborative tangibles were explored as part of the BUILD-IT system (Fjeld et al., 1998), allowing any number of people around a table to use physical blocks to move digital content projected onto the table (e.g. furniture for the layout of a room). Like Bricks, the tangible controls could be used to manipulate and perform transformations on the digital content. In addition, the blocks could also be used to manipulate intangible attributes, such as the positioning of light sources (resulting in shadows) or the position of a virtual camera when viewing the design in 3D on an external screen.

Physical-Virtual tools (Fjeld et al., 2002) enabled interaction with digital data using physical tools as the primary means of interaction. For example, both the I/O Brush (Ryokai et al., 2004) and Slurp systems (Zigelbaum et al., 2008) where physical tools that used light sensors to extract digital media from physical objects, such as selecting a colour with a physical eye dropper (Slurp) or rubbing a brush on a surface to 'pick up' that colour or texture (I/O Brush). Similarly, instead of trying to directly give a physical representation for digital information, Tangible Tiles (Waldner et al., 2006) used fiducially tracked, hand-held tiles to literally 'scoop' projected data off a table for relocation (Figure 7), copy/pasting or cutting, allowing digital operations to be performed on digital data via physical manipulation.

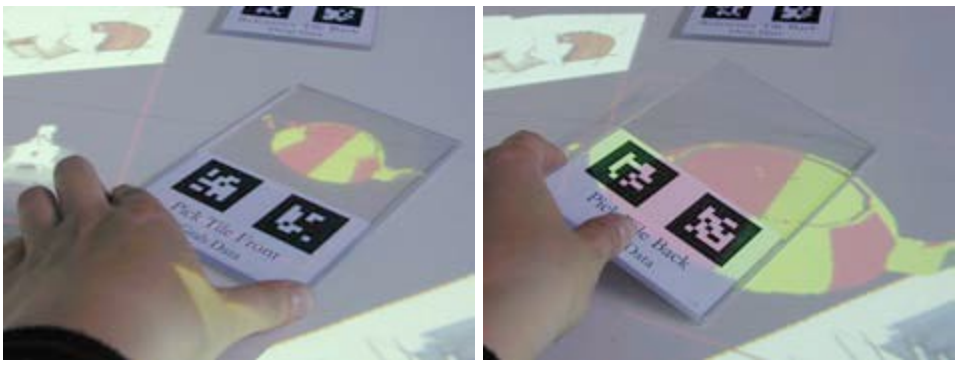

Figure 7 Tangible Tile containing digital content (left) and sliding the content off (right) (Waldner et al., 2006), reprinted with permission from Manuela Waldner

2.1.1 Classification of Tangibles

In identifying support for TUIs across different interaction domains, Ullmer and Ishii (2000) categorised physical objects into three categories; spatial, relational and constructive. The last of the categories gives way to what Ullmer (2002) later described as Tokens and Constraint (TACs), where the affordances of an object help dictate how that object can be used as a tangible controller. For example, a groove along the side of an object lends itself to be used with a marble as a tangible slider (Figure 8), giving physical constraints that can guide the user for how they can interact with that control, with function following form.

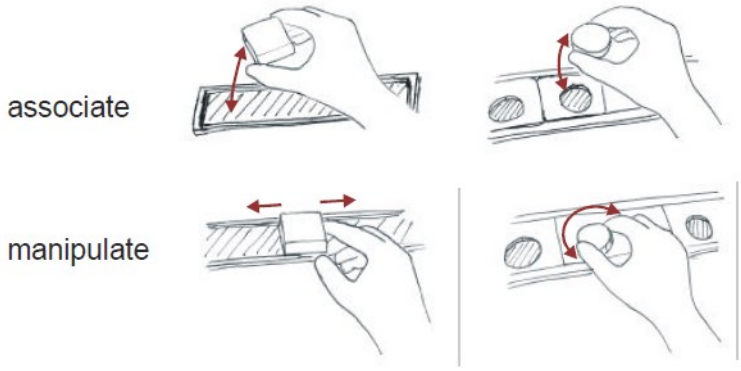

Figure 8 Physically guiding and controlling user input with TACs (Ullmer, 2002), reprinted with permission from Brygg Ullmer

In addition to TACs, Ullmer identified interactive surfaces (e.g. Illuminating Clay) and constructive assemblies as categories of tangible interaction. Similarly, in different work, Klemmer et al. (2004) surveyed different TUI systems, classifying them into four categories:

- spatial (interaction in the Cartesian plane),
- topological (relationships between objects),
- associative (objects linked to digital content), and

- forms (process of paper-interactions).

Such classification of tangible interfaces is important, as the categories can be used to evaluate not only how comprehensive a tangible system is, but also the types of the interactions that the system supports. The design principles presented by Kato et al. (2000), closely resembled those later presented, highlighting the utilisation of unique affordances, use of spatial interaction and the leveraging of bi-manual and parallel interaction.

In addition to previous work, Ullmer (2002) explored a tangible equivalent to the traditional Model-View-Controller (MVC) architecture (Olsen, 1998) used for GUIs (Figure 9). Within the traditional MVC architecture, a digital model exists as the digital state/functionality. This model is then represented, traditionally, using a screen display which constitutes the digital transformation of the model to displayable content, as well as the physical display itself. To control the system, an input control element exists, existing as in both the physical realm as the control itself, and the associated digital functions/state. For example a mouse has the physical component, but also the state and function of the mouse, e.g. is the mouse a pointer, a selection tool, a paint brush, etc. In modifying the MVC to support tangible interaction, the model itself remains entirely digital. The control, now tangible, exists only in the physical (i.e. tangible) realm. The previous view is now split into tangible (graspable) and intangible (sound, video, etc.) representations. This Model-Control-Representation (intangible and tangible) is referred to as the MCRit model. In relation to the MVC, Klemmer et al. (2004) identified that toolkit support for the traditional MVC view has remained separate from toolkit support for the control (inputs), as well as classifying combined control/view entities as *widgets* (Conner et al., 1992).

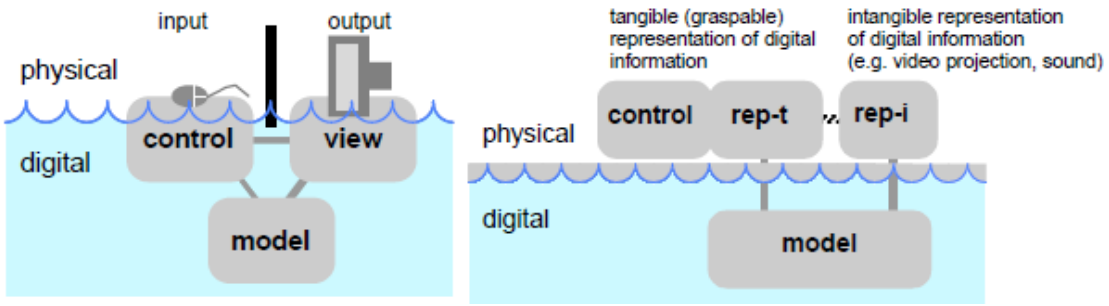

Figure 9 MVC and MCRit tangible equivalent (Ullmer, 2002) , reprinted with permission from Brygg Ullmer

Both Underkoffler and Ishii (1999), and later Fishkin (2004) identified tangibles UIs as being a spectrum, rather than a binary value. Underkoffler and Ishii proposed a single continuum to identify tangibles, showing a transition from an 'object as a noun' to an

'object as a verb' (as previously highlighted earlier in this section), contrasting Fishkin who used two-axes (embodiment and metaphor) to identify tangibles within that space.

The level of embodiment within a tangible is important, as as the embodiment increases, the coupling between the input and output increases, resulting in EUIs. Within the EUI space, input directly equals output, with the most pertinent example already mentioned as the abacus, as there is no distinction between the input and output. The exploration of such mapping was explored by Koleva et al. (2003), noting that as the coherence with digital elements increases, richer forms of interaction between the physical and digital objects need to be established to convey to users the current system state so that they can make sense of their interactions. In fact, Sharlin et al. (2004) noted that in the real world, there is little distinction between inputs and outputs, with the closest approximation being cause and effect. It has only been since the introduction of digital systems that this decoupling of action and perception space has been introduced, whilst also introducing a level of uncertainty about the interaction state. As a result, they argue that TUIs are capable of overcoming these limitations by unifying input and output space.

This digital embodiment can be increasingly complicated based on the level of multiplexing (time or space) used by the input (Ullmer, 2002). The computer mouse is an example of a time-multiplexed device, and has a low level of embodiment, as does a studio mixer, but is a space-multiplexed device (Figure 10). Following this, Sharlin et al. (2004) identified the 'degrees of integration' as the ratio between the DOF of an object and the DOF of the application, noting that designers need to ensure as many 1:1 mappings as possible, i.e. not to overload inputs.

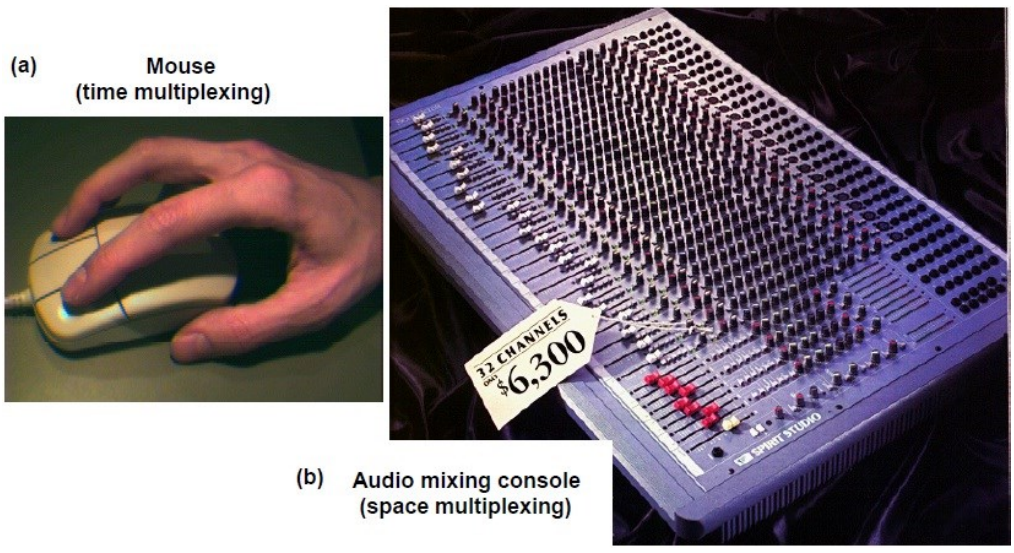

Figure 10 Showing a mouse (time multiplexed) versus a studio audio mixer (space multiplexed) (Ullmer, 2002), reprinted with permission from Brygg Ullmer

Koleva et al. (2003) proposed a framework for TUIs based around the idea of the *degree of coherence* between physical and digital objects. Like Underkoffler and Ishii, they define TUIs across a spectrum, with a number of criteria that can be used to classify them according to their ability for Transformation, Scope of Interaction, Configuration, Lifetime, and Autonomy. The spectrum closely resembles that of Underkoffler and Ishii, but instead uses the coherence between the digital information and the physical object. Increases in the level of coherence create Organic User Interfaces (OUIs) (Holman and Vertegaal, 2008), where input equals ouput, and function follows form. The shape of an object will dictate how it can be used, both for input and output. As the level of coherence increases to approach OUIs, interactions begin to more closely resemble how the real world operates, with little distinction between input and output as previously highlighted in this section.

Also similar to the previous themes, Hornecker and Buur (2006) proposed a framework looking at the device, its purpose and its involvement in the collaborative environment, identifying the themes of Tangible Manipulation, Spatial Interaction, Embodied Facilitation and Expressive Representation. The framework is interesting as it notes the specific affordances offered by tangible interaction to support collaborative environments, as opposed to other classifications that just look at the device's function.

Bowman et al. (2001) identified input devices as either having discrete or continuous events, with the control either generating one event at a time, versus a stream of events. Whilst digital controls have the ability to generate distinct events, tangible controls lack that (primarily) digital affordance, instead relying on a distinct physical interaction as the event.

When examining the fundamental types of possible input, Foley et al. (1984) identified six fundamental interactions for graphical systems: select, position, orient, path, quantify, and text. However it was noted that in the physical world, any number of interactions can be performed with an object (e.g. squeeze, stroke, push, etc.). Whether these are fundamental types of inputs or just different forms of input for other fundamental types of input is a point of discussion. Like the identification of tangible categories, identifying what types of input are possible within a system can be used to evaluate how well a single system supports the full spectrum of possible inputs.

In summary, tangibles have been classified according to a number of different frameworks and categories. Ullmer and Ishii's work along with Ullmer's TACs classifies

TUIs based on their physical properties and affordances, which are further classified by Klemmer et al. as being part of the spatial and topological TUI categories, alongside associative and forms. Despite TUIs possibly falling into these categories, their identification as TUIs is based on a spectrum, rather than a binary value, such as with the level of embodiment (Fishkin et al.) and level of coherence (Koleva et al.) or DOF (Sharlin et al.). Including this concept of embodiment, Hornecker and Buur identified themes of TUI interaction based on their collaborative aspects, with Bowman et al. classifying tangible input as creating discrete or continuous events.

2.2 Ad-hoc and Reconfigurable User Interfaces

As previously mentioned in the Introduction, in the early days of the digital age, systems were customised or written entirely from scratch for individual installations as computers were expensive, programmers less so (Halbert, 1984). As prices decreased, commodity systems became more pervasive, meaning users could purchase software off the self, expecting it to do what they want. However, they no longer had the programmer to customise the software as required, only themselves. As such, the ad-hoc nature of real world interactions creates the need for reconfigurable interfaces, where the input control and functioning logic can be manipulated at run time. Whilst UI definition/mark-up languages exist, such as UsiXML (Limbourg and Vanderdonckt, 2004) and UIML (Abrams et al., 1999), they are primarily designed to support the UI for a system across multiple devices, i.e. one UI, multiple devices. This ability for UIs to adapt to their context of use has also been described as the level of plasticity (Thevenin and Coutaz, 1999), where a formal model of the UI can be used to generate device/context specific implementations for that fixed model.

However, the context and requirements for the UIs themselves may vary (in addition to the device/platform being used), from supporting different use cases as time progresses to the legal requirement that some systems need to support adaptation for users with disabilities (Greenberg and Boyle, 2002). In this vein, various systems have explored different options, using hardware, software, or artificial intelligence. This section provides an overview of reconfigurable tangible interaction across these areas, as well as their evaluation.

2.2.1 Ad-hoc Ephemeral Interaction

The move to connected systems means interactions are no longer limited to a specific device or specific platform, as instead users now live in a heterogeneous digital world.

Functionality and the associated interactivity are now performed across devices and across modalities. Coutaz (2007) classified this as Ambient Computing that utilises 'meta-UIs', shifting from single workstations to interactive spaces, blurring the lines between the physical and digital realms in the process. However, pre-packaged solutions are no longer adequate, and although techniques are being developed for ambient computing, they are on a case-by-case basis (Coutaz, 2007).

As previously mentioned (beginning of Chapter 1), we all utilise the objects around us as part of our everyday interactions based on their physical properties of purely physical proximity at the time of need, often assigning roles to these objects (Coutaz, 2007). Indeed, "the proposition of giving additional meaning and animate life to ordinary inert objects is a cognitively powerful and intriguing one" (Underkoffler and Ishii, 1999). Town planners at lunch might utilise different objects on the table as major landmarks in a town, planning out different configurations as they talk about them over lunch (Figure 11) (Coutaz, 2007). Despite the simple roles and rules of interactions used, complex configurations can quickly be created. This interactive system creates a cognitive overhead that must be tracked not only by the user creating it, but any other users participating. By using a tracking system, the system can track these objects and create a coupling between the physical object and digital role assigned (Coutaz, 2007). Whilst not presenting a system, Coutaz (2007) instead classified such interfaces according to the existence, observability traceability, and controllability of the service being used, whilst classifying the tasks performed by the user as one of discovery, coupling, redistribution, and remoulding. They identify the use of a user defined context, involving a set of entities, a set of roles/functions that those entities may satisfy, and a set of relations between them (Coutaz and Calvary, 2012). It is those entities, roles and relationships that need to be expressed/captured by the system, and as yet have not been supported in an ad-hoc context. When using objects in such a manner, Cheng et al. (2010) noted that whilst purpose built devices will always have an advantage, selecting and using objects based on ad-hoc affordances could aid use as an auxiliary controller, even if they cannot be used for major input.

Figure 11 Planning a problem by assigning roles to available objects, e.g. table utensils (Coutaz, 2007), reprinted with permission from Springer Science and Business Media

Doring et al. (2013) presented Ephemeral User Interfaces as temporal UIs that have at least one element intentionally designed to last for a limited time, ideally leveraging multi-sense perception. The primary exploration of Ephemeral UIs was using materials of a physically temporal nature (bubbles, fog, ice, etc.). They defined a design space based on the:

a) materials,
b) interactions (input vs. output), and
c) aspects of ephemerality used.

Using this space, the goal of this research can be classified according to the user selecting the correct material for the object (a), primarily as a form of input (b) using objects whose role and purpose (including objects used together) is temporary. *Despite objects being persistent on their own, it is their utility together that is ephemeral.*

Edge (2008) developed a system for peripheral tangible interaction, where TUIs were used as complimentary, persistent controls to an existing system. Edge focused on the "episodic engagement with tangibles, in which users perform fast, frequent interactions with physical objects on the periphery of their workspace, to create, inspect and update digital information which otherwise resides on the periphery of their attention". This focus provides continued support for the ubiquitous Luminous Room and Digital Desk vision. The system utilised three different types of tokens (task, document and contact), to allow the user to select associated elements for further control. Whilst limited in its focus, the work provides further justification for the use of tangible ad-hoc controls.

2.2.2 Hardware

The most obvious avenue to support physical interfaces that are reconfigurable is to enable physical controls to be replaced and substituted. Greenberg and Fitchett (2001) developed Phidgets (physical widgets) as physical instantiations of GUI controls. Using controls wired to a custom control board that connected to a computer via USB, the end devices could be replaced with equivalent controls at run time, e.g. replacing a linear potentiometer (Figure 12) for a radial potentiometer. In addition to traditional controls, the system also supported unique controls such as Radio Frequency Identification Device (RFID) scanners and LCDs. The primary focus Phidgets was to lower the entry bar for developing physical input devices, enabling software developers to quickly prototype input systems in minutes without any hardware knowledge.

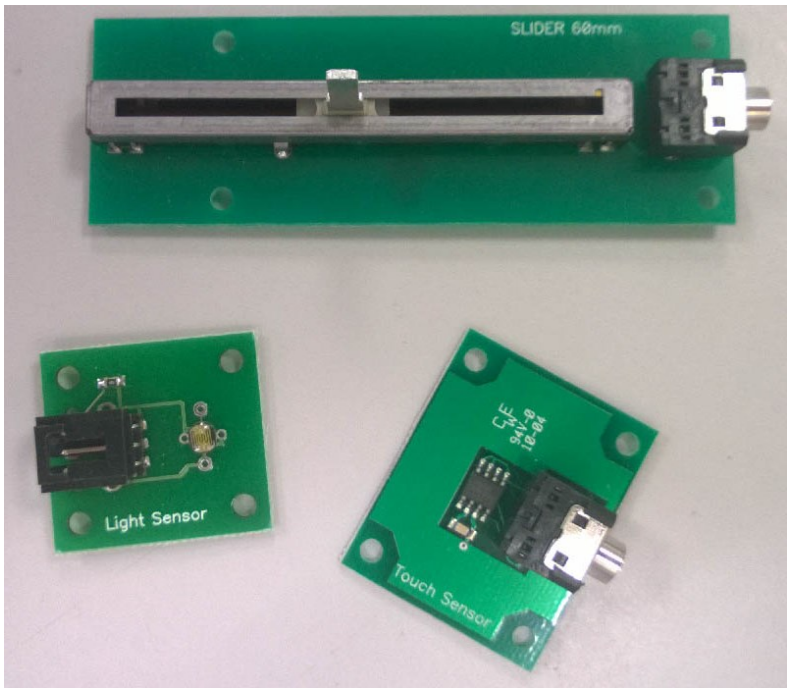

Figure 12 Phidgets linear potentiometer, light sensor and touch sensor

This approach was almost identical to Lee et al. (2004b) in the Calder Toolkit, where small, push-pin controls (both wired and wireless) could be inserted into passive foam models to aid in developing active prototypes. Like Phidgets, the controls registered the appropriate input in a software framework to be managed by the user.

VoodooIO (Villar et al., 2006, Villar, 2007) offered a similar concept as 'Flexible Interfaces', however utilised push-pin controls with an active foam substrate consisting of two sheets of conductive material. Each control would make contact with both layers, allowing it to receive power and communicate via the sheet to the host computer. Sheets could easily be cut and joined with wires, allowing the placement of sliders, dials, etc.

anywhere in the user's immediate workspace (Figure 13). Whilst being more run-time friendly than Phidgets, VoodooIO was also targeted to developers, requiring code to be written to actually interpret controls as they are added, managing their associated functionality by hand with application code. Later applications allowed the values of controls to be directly associated with inputs in Adobe Flash[2] (Spiessl et al., 2007).

Figure 13 Showing a VoodooIO dial control and substrate (left) with parts of the substrate distributed across a desk (right) (Villar, 2007), reprinted with permission from ACM

Using a mix of silicone formed objects and rear-surface projection, SLAP widgets (Weiss et al., 2009) allowed passive silicon controls to be augmented with context-relevant information as they are used. Markers on the edges of each control were detected by the system, with relevant data then projected from beneath the surface. This enabled users to bring in controls as needed, however the control functionality is fixed, and controls could not be used/created that had not previously been constructed.

Avrahami and Hudson (2002) looked at using RFID to enable active controls to be added to a system wirelessly ad-hoc. RFID chips were embedded into controls, with the chip's circuit remaining broken until the user's input physically completed the circuit. This is simple for a push button where the pressure physically completes the circuit, but for slider and controls, numerous RFID chips were used to communicate the position of the valuator. This approach is quite interesting, as it allowed the incorporation of active components without any additional hardware configuration. This approach was later extended by Simon et al. (2014) to support a greater range of values, whilst using a RFID-enabled glove so controls could be placed anywhere.

[2] http://www.adobe.com/products/flash.html

2.2.3 Software

Previous work has explored tangible ad-hoc interaction in a limited manner. iCon (Cheng et al., 2010) allowed users to attach fiducial markers to objects that were then tracked by a camera above the work area. Using a basic GUI, users could bind three basic gestures (tap, rotate, and translate) to five basic tasks using a GUI: PC and music control, and photo, document and web page navigation (Figure 14). The application would then perform pre-defined macros to simulate input for the task. In their user studies they noted that even though an object's affordance may prevent its use as a primary input, they found support for its use as an auxiliary controller. One notable finding was that such interaction was unsuitable for irreversible tasks (such as deletion).

Figure 14 iCon Fiducial markers and control application showing basic tasks (Cheng et al., 2010), reprinted with permission from ACM

BOXES (Hudson and Mankoff, 2006) supported rapid prototyping using a dedicated circuit board with thumbtacks and tin foil used as capacitive buttons connected to a dedicated control board communicating with a PC. Users could prototype different devices, e.g. an MP3 player (Figure 15), by placing the thumbtacks where buttons would be located, and then associating that input with 'Play/Pause' in their software emulator. Despite requiring the user to author software as part of the reconfiguration of functionality, BOXES allowed different devices to be physically prototyped in a matter of minutes.

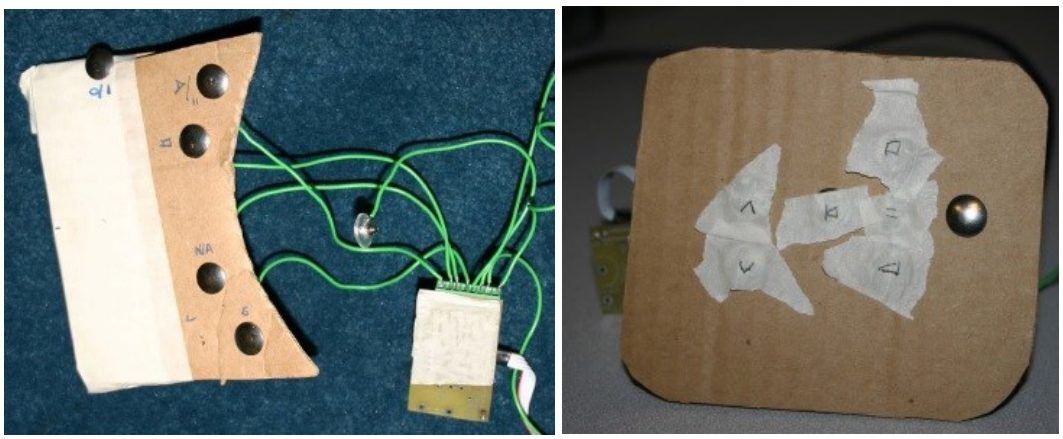

Figure 15 BOXES capacitive buttons and control board (left) and prototype MP3 player with labelled buttons (right) (Hudson and Mankoff, 2006), reprinted with permission from ACM

Similar to BOXES, DisplayObjects (Akaoka et al., 2010) used basic physical prototyping materials (foam ,etc.) to physically model a device, which is then augmented using a projector and tracking. Pre-defined digital I/O components defined in software on the computer (buttons, screens, etc.) can be selected and then dragged into place on the physical model by tracking the user's finger (Figure 16). This also allows the user to interact with the placed controls, triggering the associated functionality within the system, which may update other controls, e.g. a screen control.

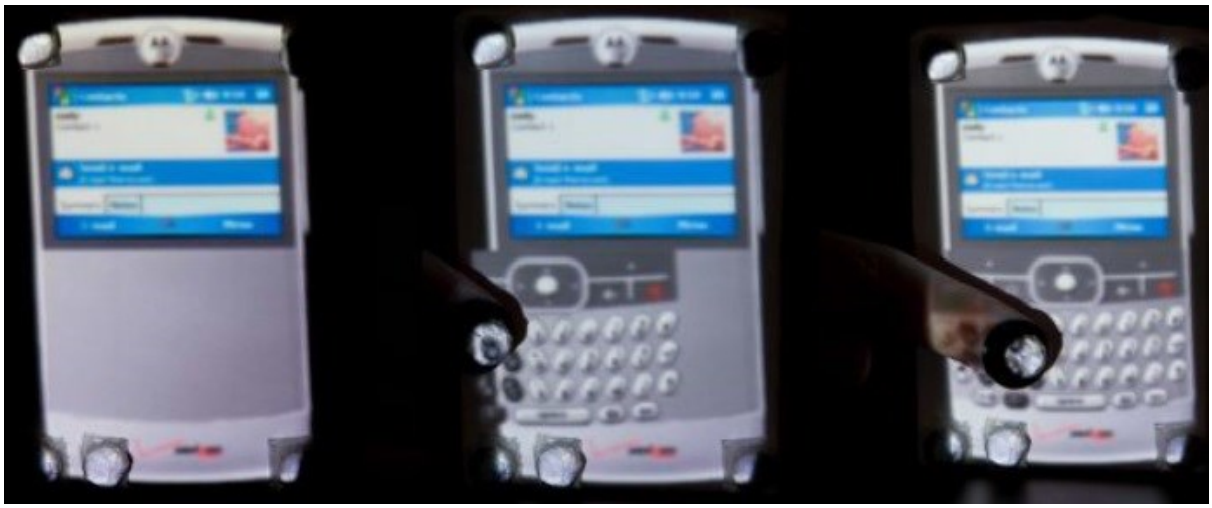

Figure 16 Dragging a keyboard control onto a foam phone prototype (Akaoka et al., 2010), reprinted with permission from ACM

Light Widgets (Fails and Olsen, 2002) examined ubiquitous interactions by allowing the user to highlight areas visible by webcams placed throughout their environment. These areas could then be mapped to functions within the software. For example, placing your hand over a section on the coffee table might turn the TV on (Figure 17). In addition to nullary functions, the system supported valuators in the form of sliders and dials. Whilst controls could be created on-the-fly, the author could not create controls within the

current context, and instead had to create them first using the computer, creating distinct authoring and interaction stages.

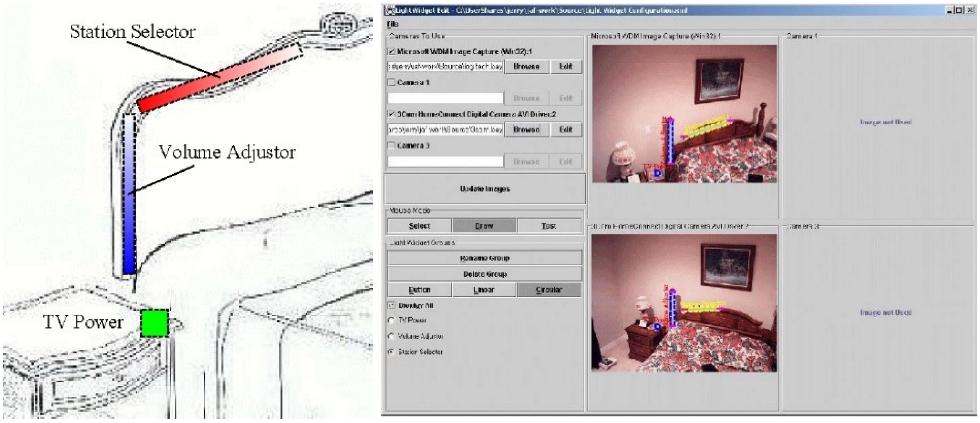

Figure 17 Creating Light Widgets in a bedroom using the GUI (Fails and Olsen, 2002), reprinted with permission from ACM

Similar to Light Widgets, WorldKit (Xiao et al., 2013) enabled the creation of touch-based controls, however they used a depth camera and projector instead of traditional webcams. Instead of using a GUI as Light Widgets did, users 'paint' a surface by rubbing their hand over the surface to create button, dial, or slider controls (Figure 18). Whilst taking the same approach as Light Widgets, the use of the projector and depth camera allowed the creation of controls using the user's current modality represents a step forward by removing the need for an external PC or GUI. However like the other tangible systems, WorldKit was an isolated system, ignoring the heterogeneous environment the user is in, resulting in another fixed system, albeit with a partially adaptable UI.

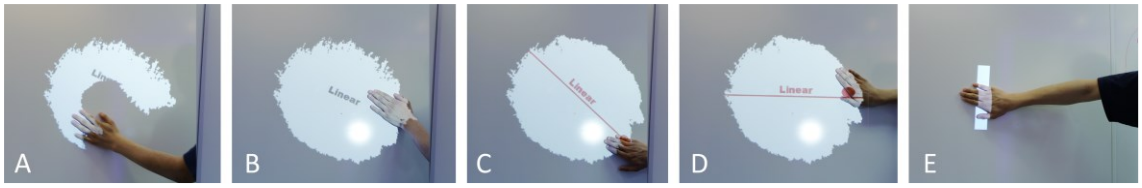

Figure 18 WorldKit showing the user 'painting' a surface and creating a slider control (Xiao et al., 2013), reprinted with permission from ACM

Opportunistic Controls (OCs) (Henderson and Feiner, 2008) extended ubiquitous interaction similar to Light Widgets by using physical elements on available surfaces in the immediate environment as the base for vision-based AR controls. Developed as a method of system navigation for an AR instruction aid for mechanics, the system would use pre-determined components near the user to navigation the system, e.g. a bolt could be used as virtual dial (Figure 19). As the OC AR system was developed for existing work environments, the controls were predetermined and could not be authored in-situ as required. In addition, like the other tangible systems before it, Opportunistic Controls was

not extensible beyond the fixed mechanical instruction tasks it had been designed for, limiting the true ad-hoc applicability of it. As such, the investigation described in this dissertation realises the potential of OCs by addressing their limitation by focusing on the design of a system to support the in-situ creation of novel controls in a *heterogeneous environment*, as opposed to pre-authored controls in a single *homogeneous system*.

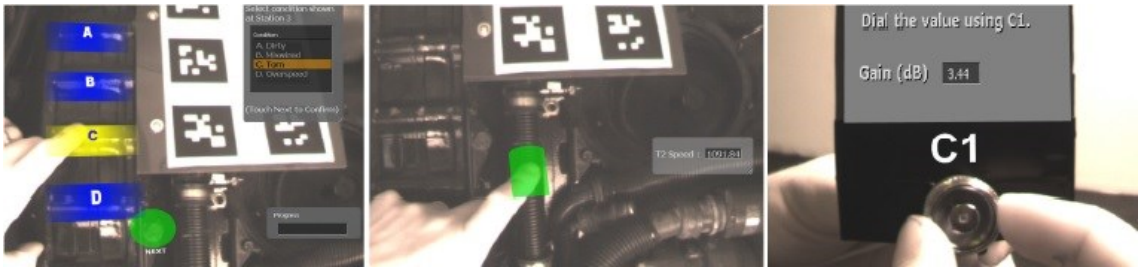

Figure 19 Opportunistic Controls with digital overlays showing buttons (left), slider (middle) and dial (right) (Henderson and Feiner, 2008), reprinted with permission from ACM

Intel's OASIS (Ziola et al., 2010) focused on interaction with ad-hoc objects via feature detection. The system could detect specific features of objects presented, allowing dynamic object-based systems to be developed for set scenarios. For example, the project allowed children to interact with projection and sound-based additions to their games by using feature recognition, e.g. building a 'dragon' would allow them to set a 'house' on fire. A more real world application was in the kitchen, allowing the system to identify and count ingredients on the counter, projecting instructions for relevant recipes. The system allowed the user to create digital proxies of real world objects, for example an orange could be captured so a projected one used for recipe selection instead of a real one.

Figure 20 OASIS identifying grouped objects due to proximity (left) with a digital proxy of the orange able to be stored in a virtual drawer (right) (Ziola et al., 2010), reprinted with permission from Ryder Ziola

2.2.3.1 Middleware

Kjeldsen et al. (2003) proposed a framework for a vision-based input system for traditional TUI applications where the vision system was separated from the application. Using middleware, an application can request a specific feature (e.g. a button) which the middle and vision-layers can then implement based on the current environment. The main benefit of the system identified by the authors is the ability for the system to be moved to

different surfaces with the system inputs adjusting automatically. This also enables the middleware to combine inputs across multiple physical surfaces, as well as combine a single action into multiple events for different applications.

Papier-Mâché (Klemmer et al., 2004) continued abstracting the physical inputs from the software events. Their high-level framework abstracts passive (vision-based) and active (electronic, RFID, etc.) inputs to a high enough level that different modes of interaction can be substituted for one other, e.g. allowing an application to be developed using vision-based events, but deployed using RFID events. This approach should be noted as treating the inputs as generic 'black pucks' as described by Ishii (2008), where inputs are abstracted to a high enough level as to lose the very affordances which make them unique as tangibles and thus losing associated benefits.

The iStuff/iRoom project (Ballagas et al., 2003) worked towards functionality to support ubiquitous, pervasive computing, whilst arguing that ubiquitous computing will not appear overnight, instead being a slow transformation as more devices are added over time. As a result, their main focus stems from asking "how does one design a system that may be augmented with future devices and services whose nature or feature set cannot be predicted in advance?". In order to support this interaction with future devices whose nature and features are not known, they argue for a patch panel. Patch panels utilise a flexible middleware component to dynamically map data and resources from one component to another, similar to how original telephone calls were routed by an operator using a physical patch panel. The patch panel used (Ballagas et al., 2004) supports the integration and communication of different devices regardless of the individual communication methods, using event *producers* as triggers to pass data to event *consumers*, with an event-consumer pair classed as a *mapping*. Data transformations can be manually defined with coded equations by the user, with mappings edited within a GUI (Figure 21).

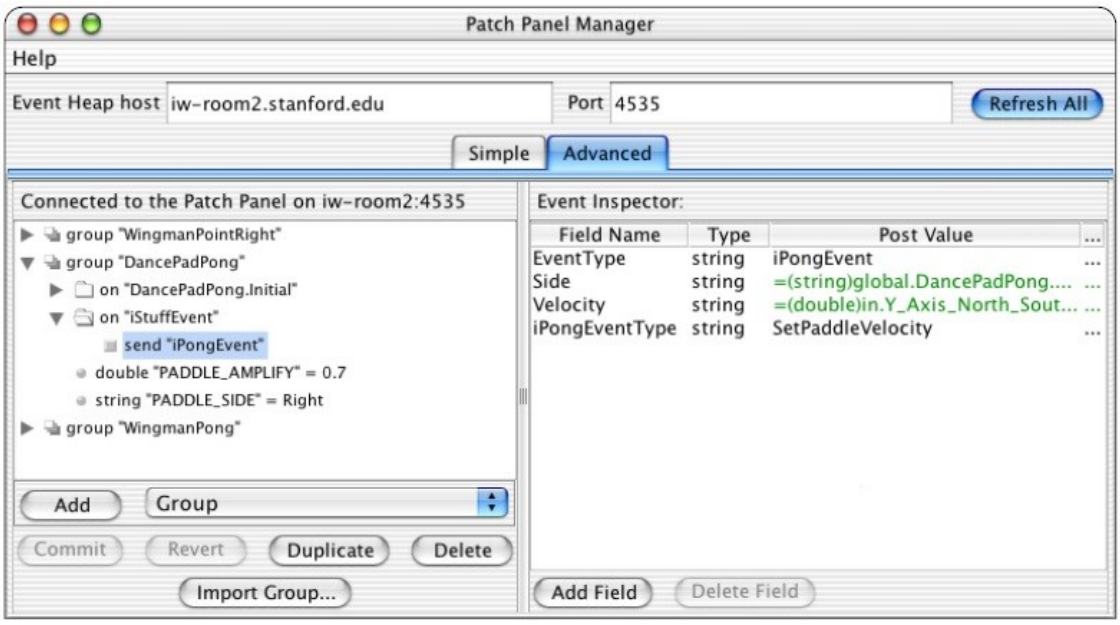

Figure 21 The iStuff patch panel GUI (Ballagas et al., 2004), reprinted with permission from IEEE (© 2004 IEEE)

Any transformations must be manually coded for each pairing, meaning a slider generating events in the range 0-1 must be manually transformed to provide values for an output that takes values 0-100. Finite State Machines (FSMs) can also be created, creating different modes for input devices. Given FSMs can require multiple actions to move between states, the authors note the need for collections of events that are performed sequentially, as such noting the need for an atomic and discrete event. Whilst the process of creating a mapping is labour intensive, it is only required when the mappings are changed. As such, iStuff is focused on the designer/developer rather than the end user, as even once devices and applications can all communicate with the patch panel, macros must still be written for each mapping. Using a similar approach for on/off device control, Fukuchi et al. (2009) proposed physical 'push-pins' that are inserted into event generators (i.e. a switch) and an event consumer (i.e. the device). By pushing the physical pins into small sockets present on the devices (in the format of 3.5mm audio connectors), devices could easily be paired together using software.

More recently, Marquardt et al. (2011) looked at providing abstraction from sensor hardware for detecting proxemic events within and between objects, as opposed to developing a dedicated system. By providing a number of generic tangible events (Figure 22), system developers can entirely abstract away the complexities of what interactions an object supports, or how to detect them with the available technology. Information and events were classified into three categories based on:

- individual entities,
- geometric relationships between objects, and
- pointing relationships between objects,

and used the distance, orientation, movement, identity and location of the objects. Despite supporting over 27 properties, only six different data types were required (Booleans, doubles, 2D points, 3D points, arrays, and strings). Given the limited resulting data types from such a comprehensive set of tangible interactions, such interactions could easily be automatically associated with functions that take in such parameters, however this was not explored.

		Property name	Description	Data type	Distance	Orientation	Movement	Identity	Location
A Individual entity	I1	Name	Identifier of the tracked entity	string				■	
	I2	IsVisible	True if entity is visible to the tracking system	bool				■	
	I3	Location	Position in world coordinates	Point3D					■
	I4	Velocity	Current velocity of the entity's movement	double			■		
	I5	Acceleration	Acceleration	double			■		
	I6	RotationAngle	Orientation in the horizontal plane (parallel to the ground) of the space	double		■			
	I7	[Roll/Azimuth/Incline]Angle	The orientation angles (roll, azimuth, incline)	double		■			
	I8	Pointers	Access to all pointing rays (e.g., forward, backward)	Array [ ]		■			
	I9	Markers/Joints	Access individual tracked markers or joints	Array [ ]					■
B Relationships between two entities A and B	R1	Distance	Distance between entities A and B	double	■				
	R2	ATowardsB, BTowardsA	Whether entity A is facing B, or B is facing A	bool		■			
	R3	Angle, HorizontalAngle, ...	Angle between front normal vectors (or angle between horizontal planes)	double		■			
	R4	Parallel, ATangentialToB, ...	Geometric relationships between entities A and B	bool		■			
	R5	[Incline/Azimuth/Roll]Difference	Difference in incline, azimuth, or roll of A and B	double		■			
	R6	VelocityDifference	Difference of A's and B's velocity	double			■		
	R7	AccelerationDifference	Difference of A's and B's acceleration	double			■		
	R8	[X/Y/Z]VelocityAgrees	True if X/Y/Z velocity is similar between A and B	bool			■		
	R9	[X/Y/Z]AccelerationAgrees	True if X/Y/Z acceleration is similar	bool			■		
	R10	Collides, Contains	True if the two volumes collide, or if volume A contains volume of B	bool	■				■
	R11	Nearest	The nearest point of A's volume relative to B	Point3D	■				■
C Pointing Relationships between A and B	P1	PointsAt	Pointing ray of A intersects with volume of B	bool		■			
	P2	PointsToward	A points in the direction of B (w/ or w/o intersection)	bool		■			
	P3	IntersectionDegree	Angle between ray and front facing surface of B	double		■			
	P4	DisplayPoint	Intersection point in screen/pixel coordinates	Point2D	■	■			
	P5	Intersection	Intersection point in world coordinates	Point3D	■				■
	P6	Distance	Length of the pointing ray	double	■				
	P7	IsTouching	A is touching B (pointing ray length ~ 0)	bool	■				

Figure 22 Proxemic information and events generated by the Proximity Toolkit (Marquardt et al., 2011), reprinted with permission from ACM

2.2.4 Programming by Demonstration/Example

For as long as user interfaces have been in existence, developers have created systems to aid in their creation (Myers, 1995). Whilst the majority of these systems are designed for the developer, another approach is to support the run-time definition of functionality by the user, for example using visual programming languages. Such systems have been found to allow non-programmers to create fairly complex programs with minimal knowledge (Halbert, 1984). However, designers need to carefully consider how end users are kept in the loop, as the temptation is high to create automatic, adaptive systems purely for the sake of the challenge (Coutaz, 2007).

Bill Buxton coined 'programming by demonstration' as a system that requires the user to specify every individual system state (Myers, 1986), allowing them to work through a specific problem. These systems were further characterised by Halbert (1984) as "do what I did", whereas inferential systems are "do what I mean". These inferential systems were

later described as ‘programming by example’ (PBE) (Dey et al., 2004). For a system to perform an action described by the user as “put that there” (Bolt, 1980) while gesturing to a map, the system needs to correlate eye and hand gestures with the deictic expressions “that” and “there”, requiring a level of intelligence in the system (Maybury, 1999). Whilst PBE systems allow the user to create complex, context sensitive interactions that would be too time consuming to build otherwise (Dey et al., 2004), the logic they create is both complex and unstructured (Myers, 1986).

Basic programming by the user already exists as a limited form of choosing options (e.g. toolbar buttons) to be an integral to the application functionality (Halbert, 1984). Programmers regularly work in abstract, users less so. However most everyday users can describe the actions required to achieve a goal, such as driving to their house. It is easy for non-programmers to think in the physical realm, as the “methods used to manage our space are key to the organization of our thought patterns and behaviour” (Tang et al., 2003). Therefore to make programming easier, the system needs to reduce the level of knowledge a user must possess before programming whilst also reducing the cognitive overhead in creating the theoretical plan used when programming. As a result, Halbert (1984) asks “How can we let the user program with the knowledge he already has?”. The answer is to let them write programs in the world in which they are already familiar (the physical). Whilst additional commands might be required, the majority of the time the instructions being used will by the ones they use normally. Despite certain additional commands being required, the majority of the instructions used will be the ones they normally use based on their normal actions.

Whilst PBE systems can be used to aid the user in utilising existing UIs, they can also be used as a major component in the definition of new ones. As previously identified (beginning Chapter 1), the user already has the knowledge regarding how they want to interact with the system and what they want it to do. It is up to the system to decipher what the user is wanting to achieve. As summarised by Paley (1998) in discussing user interactions, “we'll have to swallow our egos when they ‘don't get it:’ It's never the user's fault… They know what they're trying to do, and it's my job to build tools to help the computer figure it out”.

In looking at PBE systems, Douglas et al. (1990) developed the QUICK UI design toolkit to enable UI design by novices using visual programming, allowing the user to interact with the system in the same way they interact with the real world, by directly moving objects and monitoring the results, and making adjustments. Here, adjustments such as

moving or rotating a control communicates to the system how that control should be used. This approach was similar to Inference Bear (Frank et al., 1995), which defined before-and-after snapshots of the UI and had the system detect the associated changes in/between components. This system was extended into Grizzly Bear (Frank, 1995), which supported conditional behaviours and interactions based on sets of controls. In addition to this, both positive and negative reinforcement could be used to train the system. More recently, Li and Landay (2005) proposed Monet as a system to allow the user to quickly prototype interactive GUIs using similar methods. New controls are drawn and illustrated by the user to demonstrate how each control functions over time (Figure 23).

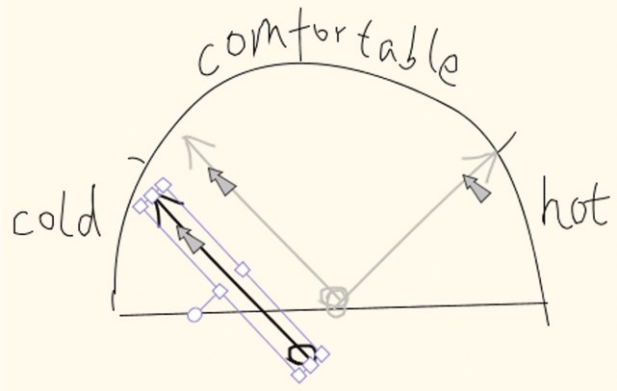

Figure 23 Demonstrating how the value of a user-defined control changes in Monet (Li and Landay, 2005), reprinted with permission from ACM

In applying PBE to game design, McDaniel (1999) highlighted the non-intuitive nature of design time versus run time for users without any development experience. In addition to this, he notes that a fundamental limitation of such system design is that the system cannot know any more about the system than the user does, requiring the explicit transfer of such knowledge. Even for simple programmatic operations such as if-else, the user must demonstrate each branch individually, back-tracking to define each branch as required.

Schafer et al. (1997) looked at using physical programming by demonstration to program manufacturing robots, for sorting objects on conveyor belts. Instead of defining rules virtually using code (as was typically done), the user could physically move objects from conveyor belt to conveyor belt to define the processing requirements and teach the system, allowing users without any prior experience to program the robots by using a language with which they are already familiar – their physical actions.

2.2.5 Evaluation

As part of their comprehensive survey of tangible interaction, Shaer and Hornecker (2010) note that the primary evaluation technique used in fixed systems has been the use of comparative studies (the 'with and without' technique), where the user first performs a task using existing methods before attempting the same task with the new methods. This provides a direct quantitative comparison between the two, serving as a performance benchmark. However, this approach does not hold when the aim is to evaluate the effectiveness and efficiency by which the system supports adaptation. Standard evaluation frameworks are not appropriate, as they do not adequately support adaptation (Stary and Totter, 1997). Evaluation by standard user interactions also becomes difficult, as the concept of a 'typical' user or use case is contradictory to the definition and goal of reconfigurable user interfaces (Paramythis et al., 2001).

For adaptive user interfaces, where the system automatically changes the interfaces based on some kind of artificial intelligence, Paramythis et al. (2001) suggested evaluating each component of that adaptive system individually, i.e. components monitoring actions, versus those identifying need for change and what the change should be. However, when the adaptations are initiated and performed by the user, this approach is no longer applicable.

In evaluating Gamut (McDaniel, 1999), a PBD game creation system, McDaniel notes that since the basic test for evaluation of the PBD system was to verify whether non-programmers could use the system, the evaluation was whether users were able to create non-trivial behaviours in the system. No statistical results were collected, instead the various functionality of the different tasks required was marked off as each user achieved it, showing expertise in it. The number of attempts it took them to achieve a given behaviour was also recorded. Using this, instead of evaluating the fact that the system was adaptive, the study evaluated how well users were able to leverage that adaptivity. All participants within the study were able to achieve the required goals using ten different interaction techniques available within the system. Interestingly, depending on the approach used by the user, not all techniques had to be utilised to achieve the required goal.

2.3 Psychology of Interaction

Despite our high cognitive abilities, "physical actions facilitate mental activity, making it faster and more reliable" (Fjeld et al., 1999), defining one of the primary advantages of

tangible interaction. However, whilst many interactions performed by the user do not directly achieve anything, they do often reveal new information and affordances that would not have been discovered through normal interactions. These kinds of interactions are classified as epistemic versus pragmatic interactions (Kirsh and Maglio, 1994). Such interactions need to support trial and error (Sharlin et al., 2004) to support the user's exploration, ensuring that the cost of their speculative exploration is low (Fjeld et al., 1999).

The continual cycling back and forth between epistemic and pragmatic interactions forms the foundation of Action Regulation Theory (ART) (Hacker, 1994), a condensed version of Action Cycles previously proposed by Norman (1988). ART defines how a user attains a goal based on a cyclic routine of planning, performing an (observable) action, receiving feedback and then updating their goals based on that feedback (Figure 24). ART can be directly applied to the design of interfaces, such as with the BUILD-IT system, where Fjeld et al. (1999) identified the need to bring goal-related cognitive activity together with associated motor functions in order to help facilitate the cognitive functions. The relationship between ART and ad-hoc systems (such as those explored in this dissertation) cannot be underestimated given the explorative nature of such systems.

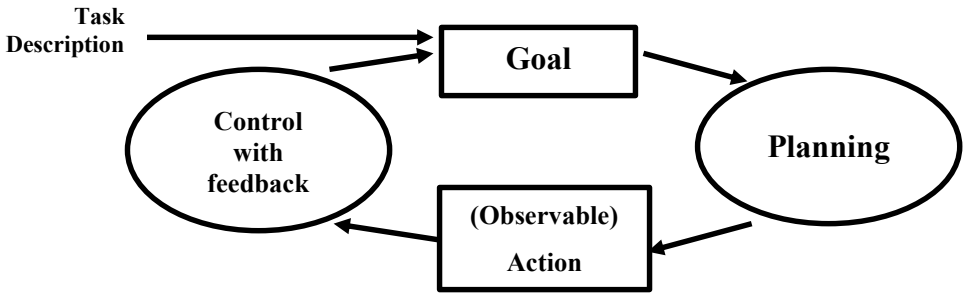

Figure 24 Action Regulation Theory activity cycle, adapted from Hacker (1994)

In obtaining feedback from the system, traditional interaction leverages subtle cues from the visual, tactile and other sensors (Sharlin et al., 2004). However as previously noted in this chapter, the distinct separation of the input from the output in digital systems creates uncertainty within the user about the system state and whether they should trust the state of the input device, or the state of the output device most. Following the development of HCI systems over the years, users have learned to identify the current system state from visual feedback as the output (Sharlin et al., 2004).

In addition to the ambiguity about system state, non-developers regularly utilise and leverage the functionality offered by abstract concepts beyond their comprehension,

without being directly aware of their use. Concepts such as shallow versus deep copies of objects and the use of inheritance are regularly utilised by end users in a system without any understanding of them. Whilst the concepts have a high pedagogical value, they often cause issues for users with no previous concept of them, for example inheritance is not intuitive for novice users (Rekimoto and Saitoh, 1999). Users unaware of such concepts are surprised by the results of interactions involving them (Travers, 1994). As a result, the incorporation of such concepts by a system, especially when used in a user-defined nature such as within user-defined and customisable systems, must ensure appropriate guidance and support for the user.

2.4 Interacting Digitally in the Real World

Whilst TUIs primarily focus on the use of objects as a form of input, the requirement for output from the system used to exist in the physical realm also exists. Augmented reality provides a medium where digital information and functionality is presented and physically registered to the real world (Sutherland, 1968), creating a mix between the real and virtual environments as identified by the virtuality-reality continuum (Milgram et al., 1995) (Figure 25).

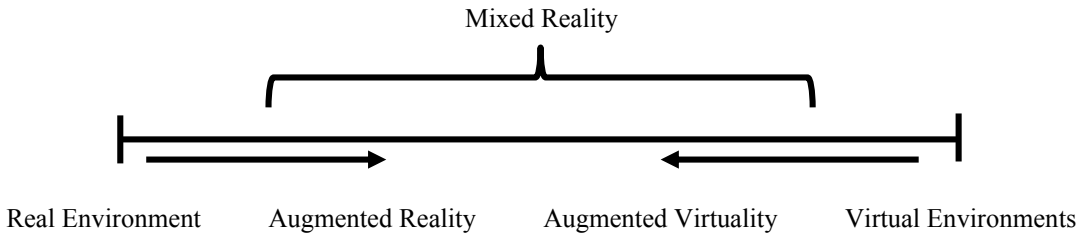

Figure 25 Milgram's virtuality-reality continuum, adapted from Milgram et al. (1995)

AR provides a number of benefits. Virtual content (and thus functionality) can be presented alongside real world content, registered to the user's viewport. Additional context-sensitive functionality can be provided, allowing the user to exist and interact within both the real and virtual worlds simultaneously. A number of different technologies exist to support AR; head-mounted devices, mobile devices and projection-based systems. The benefit of the first two is that virtual context can be shown in mid-air, as the user is viewing the world though a flat display. However projection-based AR, known as spatial augmented reality (SAR) (Bimber and Raskar, 2005), must display the virtual content directly on physical objects (and cannot display in mid-air). Whilst the requirement for projectors often means SAR is less mobile than other methods, SAR naturally supports large, collaborative environments, negating the requirement for each individual user to have their own AR hardware as the AR is present in the real world.

However, this benefit comes at the cost of being able to display AR content for individual users (given the projection is shared between users) and cannot render AR content in mid-air. Given the collaborative, large scale and more natural interactions offered (Marner et al., 2014), this dissertation focuses on the use of SAR to provide feedback to the user for their interactions.

This use of projectors displaying onto real world objects requires some form of mapping from the projectors viewport to the location of the physical objects, a process known as projection mapping. Raskar et al. (2001) presented Shader Lamps as a projection mapping system to display realistic augmentations on plain models. Later work has taken into account projecting dynamic content on dynamic models (Deepak, 2001, Marner et al., 2009).

Whilst not directly thought of as SAR, tabletop systems that utilise a projector above the interaction area are in fact using SAR systems augmenting the users' experience, identified as tangible augmented reality (TAR) systems (Kato et al., 2000). Early systems such as the Digital Desk (Wellner, 1991, Wellner, 1993) are in fact AR systems, providing additional information, functionality and context-based information through digital content displayed registered to the real world on the user's desk. With such an example as the Digital Desk, additional virtual functions (such as a calculator) were provided on the user's workspace.

Underkoffler's Luminuous Room (Underkoffler, 1997) envisioned pervasive projection displays and camera systems, allowing the system to project and track interactions in any location a room. People would use a check board pattern to play chess with virtual pieces, data could be stored 'in' objects, with the data projected onto those objects. There was no element within the room that could not be associated with virtual functionality. This concept implementation was similar to the Office of the Future (Raskar et al., 1998), where light bulbs were replaced with camera/projector pairs to augment entire office spaces.

2.5 Discussion

The advantages of tangible interaction has been well established, with user interfaces that mimic real-world behaviour likely to be intuitive (Petersen and Stricker, 2009). Given how our hands influence our concepts (Schafer et al., 1997), systems need to allow for "space-multiplexed, rapidly reconfigurable, specialised, input devices to accommodate the individual needs of users engaged in a variety of different tasks" (Fitzmaurice et al.,

1995). With proxemic interactions, designers can create systems that allow people to exploit their understanding of proxemic relations with their digital systems, facilitating more natural and seamless interactions (Marquardt et al., 2011). As previously discussed in this and the beginning of the preceding chapter, assigning interactive roles to objects is crucial for ambient computing, for example "by uttering the sentence 'this spoon is Street Michel-Ange' while pointing at the spoon, Bob couples the interaction resource with a particular digital object known by the system as Street Michel-Ange" (Coutaz, 2007).

As highlighted by the long term research visions of the Digital Desk and Luminous Room (amongst others), the ultimate tangible UI is ubiquitous computing. It is proposed that in the future, all tables and walls will eventually function as digital displays (Rekimoto and Saitoh, 1999), and thus support for interactivity is required. First steps have already been taken, for example with Peripheral TUIs, Edge (2008) acknowledged the need for "episodic engagement with tangibles, in which users perform fast, frequent interactions with physical objects", "to create, inspect and update digital information".

However, users should not be required to learn the 'language' used by these systems (Matthias and Morten, 1998), as currently much of the user's mental capacity is used adapting to the system, rather than using that capacity for the current task (Fjeld et al., 2002). This is especially important, given the majority of users of the systems are experts in the task domain, but only have minimal knowledge of the computing system (Hutchins et al., 1985), and as such, these users need to be taken into account as much as possible (Luyten and Coninx, 2001). Despite not being system designers, these non-programmers still have skills and ideas that would be useful in creating interactive applications (McDaniel, 1999).

Halbert (1984) noted that even though non-programmers are not used to creating programs in an abstract manner, most people can still follow and give directions with little trouble, calling for a PBD/PBE system that can be given to the user so that they can build and configure an application to do what they want (Dey et al., 2004). As a result, the task for system designers is to develop a suitable system interface that can also serve as a possible mental representation for the user (Matthias and Morten, 1998), allowing them to 'act out' their system as they see fit.

Previous systems have not been generic or extensible, both in terms of supporting new functionality and input interactions and modalities (both defined by the user and utilising

heterogeneous environments), limiting their applicability in the real world. In order for tangible systems to see wider adoption, generic systems must be available and support adaptation to individual users' use cases, like commercial software is now. Whilst previous systems (such as WorldKit, Light Widgets and Opportunistic Controls) have shown there is a need for extensible ad-hoc tangible interaction, they have stopped short of actually enabling it. As of yet, no generic approach has been developed to enable the in-situ creation of ad-hoc tangible controls and logic. Every approach until now has been a closed system. Given even trained system designers cannot see how software will be used by large varieties of users, and since people only utilise a small subset of functionalities at any given time (Stuerzlinger et al., 2006), I seek to support the development of system interactions and functionality as required by the user, allowing them to develop such systems without purpose-built hardware, enabling users to 'come as you are' (Wachs et al., 2011), using their normal interactions within the system to create new functionality, *authoring by interaction*.

Chapter 3. Theoretical Framework

The end goal of this investigation is to enable the ad-hoc, run-time extensibility existing systems with TUIs. One form of extensibility is the addition of new rules or logic into a system (AH-Logic), with one common example with existing non-tangible systems being the creation of rules for sorting emails. Extensibility for tangible interactions however can be demonstrated by the previously mentioned (beginning Chapter 1) scenario of a teacher utilising ad-hoc functionality to create a teaching aid to demonstrate how different elements and chemicals react with one another, as has previously been created as a dedicated AR system (Fjeld and Voegtli, 2002) and tabletop system (Patten et al., 2001). To utilise ad-hoc extensibility requires the teacher to be able to identify individual physical objects around them to be used as different chemicals. Once they have given chemical names to these objects, they can then create any number of different rules defining how different reactions between different combinations might occur by moving the objects in relation to each other around on the table into desired configurations. In addition to this, the teacher might link each reaction with a smart-board, opening links to Wikipedia to provide additional information to students whenever they discover a reaction.

However extensibility does not just mean adding new functionality, but can also refer to utilising existing functionality in new ways. If the user knows what functions a system is capable of, they will see new applications for those functions. For example one might say Microsoft Excel is more often utilised for tasks other than accounting. As such, if the functions of an application can be exposed to the real world, these functions can be controlled and utilised in new ways. However in order to be able to control an existing function requires the user not only knowing that such a function exists, but also the parameters (if any) the function requires (i.e. its definition/signature). Functions might be nullary (not requiring any parameters), require an explicit value (e.g. any integer) or even require values that are across a given range, for example. Whilst the teaching example created logic-based interactions, it also leveraged existing functionality in using a web browser to open a web page with information as trigged by a physical interaction (AH-UI). As such, in order to be able to utilise such functions, the user needs to be able to create physical controls that generate any values required as parameters. For example, the user could create a slider or dial that utilise the physical affordances of available objects, utilising the Token and Constraint (TAC) paradigm (as was discussed in Chapter 2), to

control the audio volume. Basically, physical interactions can be used to specify values that can be passed to external functions.

The third and final component to this (after AH-Logic and AH-UIs, as defined in section 1.3) is that existing functionality involves instances of data within a system. For example, the previously discussed (Section 2.1) and highly cited marble answering machine system (Crampton Smith, 1995) associated voicemail messages with tagged marbles that were 'released' as messages were received. To play a message, the marble was placed in a special cradle. In this case, the function that existed was not just 'Play Message', but 'Play Message X' that was passed an instance of a message to play as a parameter. As such, ad-hoc interactions must not only account for new interactions by defining logic, but also for new interactions that utilise existing functions that require both values (AH-UI) and data (AH-Data), and the combinations therein (Figure 26). *Truly reconfigurable systems can be realised through the manipulation and utilisation of application functions, data instances, and logic.*

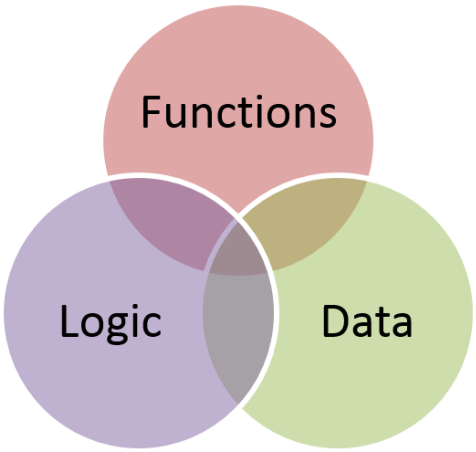

Figure 26 The distinct and overlapping interactions involving application functions, data and logic

The result of the increasing ubiquity of digital systems means interactions are no longer limited to a single device or platform. Users now exist in a heterogeneous digital world, and as such, there needs to be a way to integrate the functions provided by these different systems. The extensibility of tangibles is not just limited to what can physically be done with the objects, but how those tangible interactions can be integrated with and control existing systems. This is not just limited to incorporating any 'output' functions of other systems, where a tangible interaction might trigger a function on an external system (such as opening the web browser in the teaching example). New input modalities can rapidly become available. For example the sudden availability of an inexpensive depth sensor with the Microsoft Kinect caused a revolution almost overnight in enabling gestural and

depth camera-based interaction. As such, external systems need to also be able to generate inputs on behalf of the tangible ad-hoc system, allowing new sensing and interaction modalities to be incorporated into the system post-development. However, these external systems need to be transparent to the user, who should only be aware of interacting within the one reconfigurable system. They do not care how such functionality is supported, only that it is supported. Heterogeneous environments need to appear to the user as being homogenous, with the reconfigurable tangible system being the glue that links everything together.

The creation of ad-hoc functionality within existing external, fixed systems (hereafter referred to as "external systems") after the external systems' design and development creates a requirement for the external application functions to be exposed in a standardised way to enable control by future ad-hoc functionality. As such, there are three theoretical components in play:

1. Pre-existing hardware/software systems: any external systems providing their associated functions,
2. The new ad-hoc system: the reconfigurable system (this is not the ad-hoc functionality itself, but the system that enables such functionality), hereafter referred to as the "ad-hoc system", and
3. The new UI/functionality: the new ad-hoc functionality built using the ad-hoc system that may link to and incorporate existing functions from the external, fixed system.

As such, support for unknown functionality creates a requirement for the abstraction of new functionality through a standardised interface to isolate any new capabilities, enabling a 'plug and play' approach to new functionality. The external application functions do not know what the ad-hoc capabilities will be or how the external application functions will be used, but the use of standardised interfaces ensures the new ad-hoc functionality and the external systems can communicate. As highlighted above, adding new functionality post-development means abstracting new functionality into its own component, and having that component integrate with both the ad-hoc system and any external application functions using the standardised interfaces. *There has to be a standardised way for existing external functions to be exposed in order to support the automated integration with unknown controls.*

It's not only new functionality that is dynamic, but also how that new functionality links to and triggers external application functions. Since any ad-hoc functionality is by

definition dynamic, the linking/association of physical interactions performed by the user to trigger any associated functionality must also be dynamic by definition. The use of a patch panel (as defined in Chapter 2) enables the required isolation of new functionality from the existing capabilities of external systems. The patch panel is the dynamic 'glue' linking everything together, with “inputs” defined as user or system-generated events that trigger an associated “output”, defined as a function in the patch panel that modifies the ad-hoc system in some way or executes functions of external systems.

This general approach constitutes the basis for the Tangible Agile Mapping (TAM) architecture presented in this dissertation and chapter, and allows for the isolation and post-development modification of the input and output capabilities of the reconfigurable system (Figure 27). The data/event flow from the user performing a physical interaction with an input control, through the patch panel to the outputs (and through to any external application functions) is obvious. However, the output may also generate a response back to the user (e.g. a light turning on, an image appearing, etc.). Relating this back to the three theoretical components identified earlier, any external systems are represented as the external functions, with the ad-hoc system encompassing the creation and management of the input/patch panel/output relationship, and ad-hoc functionality defined as different types and instances of inputs and outputs.

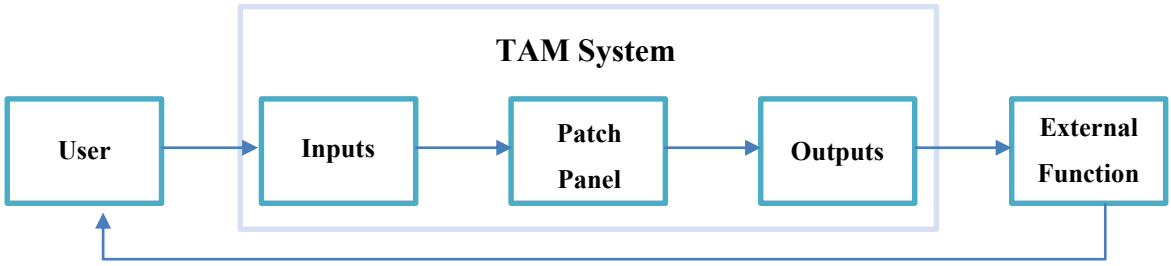

Figure 27 Conceptual structure and flow of events and data using the patch panel

Whilst different inputs and outputs may be incorporated ad-hoc by the end user, there will always remain a core set of capabilities provided by the ad-hoc system that provide the dedicated application functions as well as enabling the ad-hoc functionality. The ad-hoc system cannot be completely ‘empty’, as by definition some fundamental interactions and functions must always exist to allow the user to adapt the system as they see fit, as the user needs some way to interact with and utilise the reconfigurable functionality.

Despite the fact that both interactions sensed as input as well as output functions generate and require different parameters respectively, they can still be abstracted through corresponding interfaces. Instead of just defining that there is an input or an output, the definition of such a function needs to also provide what kinds of values/data are

provided/required. This allows different types of system inputs to be generated based on the definition of the outputs, with any association between the two handled dynamically by the patch panel middleware, which is responsible for providing output functions with the required parameters from relevant input controls. This need to develop and incorporate different inputs and outputs in isolation is crucial to ensuring extensibility through support for heterogeneous systems. Using this approach, the integration of different classes of tangible user controls (AH-UI), data (AH-Data) and logic (AH-Logic) manipulation are explored.

3.1 Supporting Tangible Controls (AH-UI)

In addition to using simple touch-based controls (such as touching a virtual line on a table for a slider, or a region for a button), physical interactions with objects can be interpreted as valid inputs for existing application functions (such as spinning a pencil as a dial). Users should be able to define how they want to use certain physical objects to control those known functions. Looking at dedicated proxemic toolkits such as the Proximity Toolkit (Marquardt et al., 2011), a number of different types of interactions with objects are possible, for example proxemic events that generate nullary (e.g. is an object visible), range-based (e.g. the interpolated position of a slider between two extremes) or explicit values (e.g. ensuring the user has three oranges on the table for a given recipe, as used in the WorldKit system (Xiao et al., 2013)). To support the ad-hoc creation of such interactions, a number of requirements exist:

- Arbitrary objects need to be used within the system, creating a requirement for their identification and definition to the system.
- Support for properties that can be used to describe unique attributes of the objects (identifiers, augmented colour, etc.).
- Define rules or interactions of some kind that can make any number of changes to object properties.
- Support sequential interactions, e.g. interaction A must occur for interaction B to occur.
- Interactions within the system must follow the natural progression for how users would describe the same scenario to another person. The ultimate system would be one where the user could describe a scenario as if describing it to another person, being able to use any modality or structure to communicate.
- Interactions need to be generalizable, allowing for objects to be assigned as members of different classes or conceptual ‘groups’, allowing for interactions to

then be created using those class/groups of objects, i.e. do not define the same interaction for 20 different objects, instead define one 'group' of 20, then one interaction using that group.

- No distinct development/usage stages, given the use of such stages has been shown to be unintuitive for end users (McDaniel, 1999), as previously discussed (Section 2.2.4 and Section 2.2.5).

In addition to the above requirements, a number of goals exist for enabling the generation of tangible UI controls by the user:

- Enable the integration between the tangible ad-hoc system and any external system applications providing additional functionality, whilst ensuring their distinct separation, e.g. a patch panel.
- Allow functions from any number of different systems to appear side-by-side, i.e. the user should be unaware that a) any external systems are being utilised to provide the functionality, and b) what individual system is being used to provide each output function. As such, functionality involving external systems needs to be leveraged in an identical manner to the native functionality.
- Support a mix of modalities through touch and tangible-based controls, as well as other NUI modalities.
- Allow external systems to provide inputs into the system to that can be utilised by the user as they see fit (supporting new types of inputs and modalities).
- Enable the authoring of controls entirely within the user's current modality (e.g. a PC is not required to create controls).

Given users only utilise a small number of set functions when they use a system (McGrenere et al., 2002), the system only needs to focus on supporting the user in creating a small number of controls for the current feature subset. Whilst previous systems have enabled the run-time creation of controls and the mapping of controls to functions in the user's environment, they have been limited in a number of ways:

- The user must often exit the current context, go to a PC, and author controls in the environment using the PC using a different modality than they were previously using, before then returning to the original context to use the created control, e.g. the Light Widgets system (Fails and Olsen, 2002). Controls are authored in an external environment, using different interaction modalities, i.e. not ad-hoc.

- Systems are limited to supporting a small set of pre-defined functionality that cannot be extended, e.g. the WorldKit (Xiao et al., 2013) and iCon (Cheng et al., 2010) systems.
- Users are required to write code to define how a user control maps to application functions (i.e. offline, using a different modality), e.g. (Ballagas et al., 2004).

In summary, given the speed, improvised nature, and the range of tasks possible in everyday interactions, there is a need to enable the ad-hoc creation of controls leveraging the interactions possible within the current modality used the user.

3.2 Supporting Interaction with Data (AH-Data)

The prerequisite that some application functions require different values as parameters also extends to the ability to include instances of application data in those parameters (i.e. pass in a reference to a data object). For example, the user might want to create an interaction in a tangible video editor to delete a given video clip. In this case, the delete function is not a nullary function, as it requires a reference to a data object as a parameter for which object to delete. Whilst programmatically supporting references for data instances is no different to just supporting any other data type, a number of other considerations must also be taken into account:

- How to define the different types of data objects that can exist.
- How instances of data of different types can be associated or what relationship between them are possible.
- How to pass instances of data to external functions as parameters that might have certain constraints (e.g. data type, number of objects, etc.).
- Enabling support to create those associations or relationships involving the data.

For example, for a tangible photo tagging application where the user can tangibly associate any photo with a text tag, the ad-hoc system would need to support at least two different data types, images and tags, and support the one-to-one association of an image to a tag. When the user uses the ad-hoc system to associate an image with a tag (using whatever physical interaction is used to make that occur), that action/trigger is sent back to the normal photo tagging application, along with the objects involved (that are essentially just acting as the parameters for that function), for the external application to perform the association/tagging.

3.3 Supporting Logic (AH-Logic)

Supporting user-defined logic within the system is very similar to supporting tangible user controls, but, for example, instead of a single object being used as a slider between two positions, the Boolean property of an object being in a given position relative to any other objects defines a more complex user control involving multiple objects. Instead of having interaction support for a dial or slider, custom logic is supported by allowing the system to record/capture given rules defined by the user, and monitor future interactions to match the captured one. If the interaction has been matched, then an associated function can be called, perhaps even generating parameters for that function based on which objects were used or their configuration (e.g. distance between objects). As such, this means the support for tangible ad-hoc logic needs to allow:

- Users to create simple, domain independent, logic/rule-based interactions within the system using arbitrary objects without the system requiring any prior knowledge of the task or content being used by capturing configurations defined by the user at given times.
- Allow the creation of those rules to be generalised to avoid repetitive entry.

Whilst defining logic for an entire application is a significant task, it does not mean that smaller interactions cannot be created as part of a larger system. The use of logic does not restrict whether explicit rules or artificial intelligence/machine learning is used. Like supporting different external input controls, all that is required is a component-based approach that enables different user controls to be created. In many cases the control being created involves a more complex process of matching any number of objects to a known configuration, collectively enabling a set of logical operations.

3.4 Resulting Architecture

Up to this point, this chapter has described the features and capabilities for the different classes of interaction required to enable extensible, tangible ad-hoc interactions. This section serves to outline the TAM architecture that was designed to support such features. TAM consists of six main components as shown in the UML diagram in Figure 28: InteractionObject, Property, Interaction, Action, PropertyMap, and PropertyMapping. Aside from the six components, the rest of the system, acting as the host for such functionality, is identified as the “Tangible Application Host”. The remainder of this chapter serves to describe the architecture, introducing and describing each of the six components as they relate to the requirements for ad-hoc interaction. Later chapters explore the application of TAM to the different aspects of ad-hoc interaction (AH-UI,

AH-Data, and AH-Logic respectively), providing three implementations focusing on different aspects of interaction, demonstrating the application of such a structured architecture to support radically different scenarios.

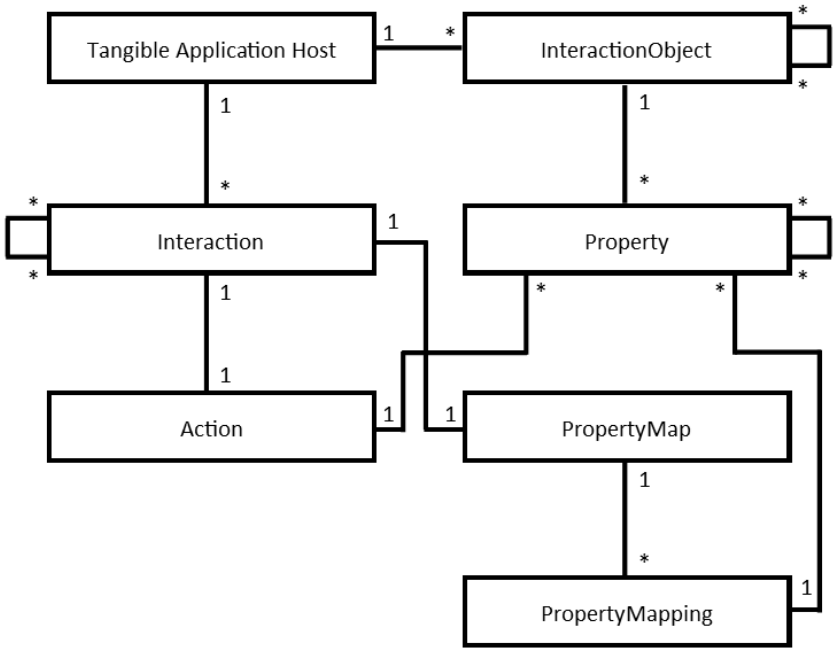

Figure 28 Relationships between the core components of the TAM architecture

3.4.1 Interaction Classification

In looking at tangible interactions, there are a number of different combinations of interactions and associated outcomes that can occur (Table 1). Multiple interactions can be distinctly separate, involving different physical objects (i.e. *isolated interactions*) or involving some or all of the same physical objects (i.e. *overlapping interactions*). In addition to this, the results of those interactions may result in changes that affect two distinctly different sets of output (*isolated updates*) or create a state of conflict if updating the same output (*common updates*). It is important for the system be able to natively deal with each scenario as to avoid any possible disconnect rising from the ambiguity of how the system should interpret a scenario.

	Isolated Updates	*Common Updates*
Isolated Interactions	Scenario 1 *e.g. "two different interactions affecting two different outputs"*	Scenario 2 *e.g. "two different interactions affecting the same output"*
Overlapping Interactions	Scenario 3 *e.g. "two different interactions involving some of the same objects, but affecting two different outputs"*	Scenario 4 e.g. *"two different interactions involving some of the same objects and affecting the same output"*

Table 1 Interaction scenarios

These scenarios can be represented according to the relationship between the inputs, outputs, and any associated mapping between them, as seen in Figure 29, which shows both the physically embodied (bottom overlap) and digitally assisted (top overlaps) nature of different interactions. Controls that exist in the bottom overlap between inputs and outputs can be identified as physically embodied UIs, such as an abacus. Traditional TUIs represent the top overlaps between inputs and software, and the software and the outputs. When an input occurs, software interprets the input and generates an associated outcome. Perhaps the most interesting component of such interactions, is where embodied interactions allow the user to not only have the direct physical affordances utilised as output, but also have these interactions leveraged as input for other outputs as well (i.e. the overlap of all three regions). Systems such as the Illuminated Clay project (Piper et al., 2002) provide such interfaces, where embodied interactions are leveraged with the addition of digital functionality. By developing a framework to support basic 'input creates output' interactions, support for embodied interfaces will be implicit.

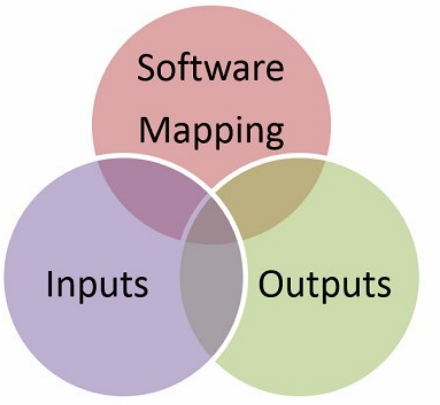

Figure 29 Showing the relationship between inputs and outputs, with embodied user interfaces located in the middle overlap

Using this representation, and as previously discussed (in this chapter and Chapter 1), a number of features of the system can be identified. First, the input must first be known (defined), from which there can also be any number of associated outputs that occur, either as the result of naturally embodied interaction, or with the assistance of an external system. The incorporation of external application functions requires not only the definition of the output function itself, but also the software patch panel that maps the input to the output. As per the goals and requirements for generalised interactions, the architecture developed was designed for generic applications, making no assumption regarding what kind of tangible systems were going to be created, or in which domains those systems would be applied. The TAM architecture developed to support such features is described in the following sections, outlining how the different functions and capability of ad-hoc systems can be grouped, encapsulated and abstracted through standard interfaces.

3.4.2 Supporting the Incorporation of Physical Objects

Like utilising an object in real life, before the system can use a physical object, the system must first know that such an object exists, along with the associated properties for that object. As such, each physical object used within the system is defined as an *InteractionObject*. The InteractionObject class defines the basic, required attributes for every object, the most basic of which is a user-friendly name and system identifier, defined as a string. Whilst an object's name may or may not always be used, it ensures the system always has a user-friendly tag with which the user can identify each object if needed.

The various different unique properties associated the InteractionObjects are represented using a generic *Property* class (note the capital "P" to denote the programming class as opposed a conceptual property as an attribute). Given any number of different types of properties may need to be associated with an object (e.g. numbers defining a position, colour, or texture describing the object), the Property class is inherited from by other Property sub-classes that provide the functionality and data storage required for each type of property associated with an object. For example, the core Property class would be inherited from to create PositionProperty, ColourProperty, TextureProperty, etc. classes, each storing their own corresponding data, but accessible through the universal functions declared in the core Property class. The core Property class provides a user-friendly name, along with stub methods for setting and getting values – using the Property type as the

parameter and return type respectively (which would be changed/overwritten by the individual inheriting classes with the appropriate data type).

Aside from implementing the individual functionality of each data type stored, there needs to be some way in which the type-specific data could be stored in a generic, non-binary format. This would enable the system to be able to provide some level of accessibility and functionality based on the values of different Property objects. For example, if defining a board game, the player piece may be required to stay within a given area, and as such, the data stored in a PositionProperty might be need to be somehow mapped to the object's ColourProperty, allowing the colour of the object to change as the object moves further from its origin. Whilst previous approaches such as the iStuff patch panel have addressed this by having the user manually write a piece of code that acts as an adaptor between the two, true ad-hoc interactions should enable the automation of such tasks, removing the need for the user to author such code. To achieve this, data must be stored in a generic manner where possible, as to ensure it can be interpreted and reused by other components in the system. This is enabled in two ways.

The first is the creation of a Property to store numerical, or more specifically integer, values. The *IntegerProperty* class, aside from inheriting the core Property configuration, provides a private attribute storing a single integer value. Whilst this provides a generic abstraction for storing a basic data type (an integer), there still needs to be some way to associate an IntegerProperty with a more complex Property.

Complex Properties can be realised by enabling a Property instance to contain/encapsulate references to any number of other Properties, e.g. to allow a PositionProperty to actually encapsulate any number of IntegerProperties to store the individual X, Y and Z axis positions. This creates a hierarchical structure within the Property class, as can be seen in Figure 30, showing the basic UML structure of these components using example Property types. A LocationProperty, might actually encapsulate a PositionProperty and an OrientationProperty, each of which encapsulates their own IntegerProperties. As such, the core Property class stores a collection of pointers to other 'sub' Property objects, supporting the creation of such a hierarchy. To support assigning and retrieving values from such a hierarchy, the Set(Property*) function performs the assignment recursively, providing a deep copy assignment from one Property to another. For example setting one ColourProperty to another would actually assign one ColourProperty instance to the other, and then copy each sub-Property contained in that instance, which might, for example, be three IntegerProperties (for the

RGB colour channels). Such assignment of one Property to another does require both structures to be somewhat compatible, e.g. assigning a non-hierarchical IntegerProperty to a ColourProperty may have unexpected results, with the recommended handling behaviour to be to assign whatever possible Properties can be assigned, with the remainder skipped. This is not largely a problem, as the direct utilisation of such functionality would most likely be created by and constitute the core functionality of the system, and thus be designed from scratch, and not at run time by the user. However, this structured approach to storing object Properties is still required as to ensure all data is available to external application functions, as is discussed later.

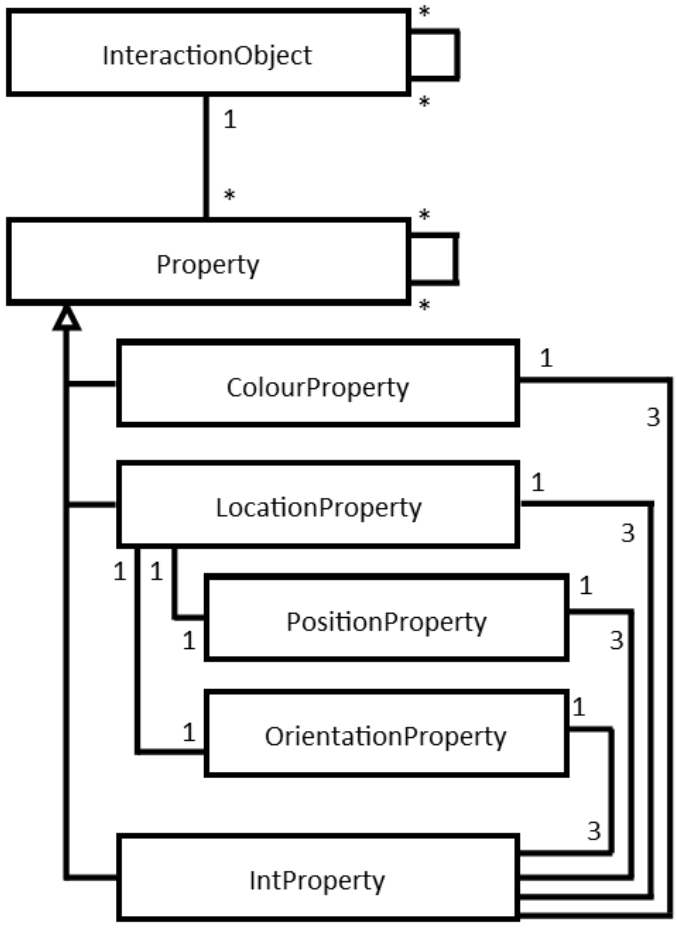

Figure 30 UML of the InteractionObject and example Property classes, showing integer values stored within other encompassing Property types

Whilst intangible properties such an object's position and orientation can now be stored, there is as yet no way for Properties to render any representation on the real world. For example, a ColourProperty can be set to a value, however even if the ad-hoc system supports augmenting an objects colour (e.g. using a projector), there is no way for the system to know that this is how this collection of Property values should be interpreted (and rendered). As a result, the core Property class contains an empty Render() method

to be overwritten by each specific Property sub-type that has a representation to the user. For example, in a SAR-based system this might be involve using OpenGL commands to render objects a certain colour (using the associated position/size Properties to render the object), or if the physical objects used by the system contained electronics, Render() might contain some way to change the colour using LEDs. For the OpenGL example, in order for a Property to access the corresponding position/size Properties for an InteractionObject to which the Property is associated, the Property contains a reference to the parent InteractionObject (most likely set as part of its constructor), from which other Properties can be accessed.

It is important to note that the architecture does not dictate how the Property is rendered, rather just that Property is the component responsible for rendering, as within the system, Property is the only object that is aware of the correct context/method to which its value(s) (if any) should be applied. Whilst the system could support a core number of different fundamental Properties as part of the InteractionObject itself, with the system rendering all Properties (instead of each Property rendering itself), this would limit the extensibility. In the future, other Property types may be introduced, e.g. one supporting texture or sound. These would then not be understood by the system, and thus could not be rendered. One could even remove the need to have the name/identifier of an InteractionObject stored directly with the InteractionObject, and instead provide a StringProperty (for example) to store/render such data. Such a Property is explored later in Chapter 6 when enabling interaction with application data.

In summary, InteractionObjects provide an instance of a physical object being used within the ad-hoc system, and are defined by any number of hierarchical Properties that contain a user-identifiable name, stub methods for managing basic get/set functionality, along with an implementation-specific and Property-type-specific render method.

3.4.3 Enabling the Definition of Types/Groups of Objects

Given the possible assignment of cognitive roles to physical objects, interactions based on those roles are possible. For example nullary interactions such as “when any one of these objects is in this area, do X”, should not have to be created as multiple times for each object. Instead, the system should support the creation of a single type/group of objects, allowing any one of that type/group to trigger the interaction, enabling the generalisation/substitution of objects.

To enable the substitution of one InteractionObject for another, an InteractionObject can contain a collection of references to other InteractionObjects that it encapsulates. This allows an actual physical object to be described to the system as 'containing' a number of other subsequent objects, with a simple example being to describe a car as having doors and wheels. Aside from support for this kind of 'is-a-part-of' relationship, intangible InteractionObjects can be created that encapsulate any number of other tangible objects. This allows the parent InteractionObject to act as an encapsulating 'group'. Given any number of different objects can be contained within an InteractionObject, the use of an appropriate name for the parent object provides a method for defining a common attribute between the member objects. This approach is similar to how drawing objects in desktop publishing application work, where any number of objects can be 'grouped', however groups may contain other groups, etc. Essentially, all physical objects within the system must inherit from the same InteractionObject type, and provide a collection of other objects of that same type which could be classified as part of the parent instance/object.

For example, in the chemistry reaction example, the two hydrogen atoms could be used to interact with a single chlorine, with only one hydrogen atom required to trigger the reaction. To create such an interaction, the two hydrogen objects are defined individually, and then defined as being part of another separate InteractionObject containing them both. When the interaction is created, it is created using the encapsulating InteractionObject, and thus whenever any contained object matched the required configuration, the interaction would occur. Because the InteractionObject contains only references to the other InteractionObjects, they can be members/part of any number of other groups. Aside from the encapsulating InteractionObject containing references to the child objects, the children also maintain a reference to any parent InteractionObjects. Using this approach, each physical child object is defined once, and then essentially encapsulated/grouped within the conceptual parent object, and the interaction defined using the parent InteractionObject. The application of such functionality is discussed in more detail in Chapter 7.

3.4.4 Supporting Interactions

Now that the system has structures to track and describe the objects used within it, interaction between these objects can now occur. As such, to enable interactivity, the system needs to be able to:

- monitor the interaction space to detect when a desired interaction occurs,

- make any number of changes to objects that occur as a result of the interactions, and
- execute existing functions (within the ad-hoc system or externally) that (possibly) take parameters based on physical interactions.

Each requirement will be addressed separately in the following sections, describing the sensing of interactions, modifications of objects, and the execution of external functionality.

3.4.4.1 Monitoring for Interactions – Defining the Inputs

As previously noted (Section 2.1), whilst Foley et al. (1984) identified only six fundamental forms of input for graphical systems, any number of different interactions can be performed with any object in the real world (e.g. squeeze, stroke, toss, push, tap, pat, etc.). Given this, the designer of the system must decide which actions should be interpreted as having meaning (Foley et al., 1984). Even recent tangible toolkits, such as the Proximity Toolkit (Marquardt et al., 2011), demonstrate a wide array of possible interactions. However, whilst there will always be some fundamental limit to what the system can interpret, the method used for supporting different types of input should be easily extensible.

To achieve this, the sensing of interactions within the system is encapsulated entirely within a core *Action* class, with a different subclass of Action created for each individual interaction to be monitored/natively supported by the system. Instances of Action are passed all the information known about the current context (e.g. InteractionObjects in the scene, current touch interactions, etc.) to a corresponding Evaluate() function. The Action instance then evaluates all of the contextual data, looking for a given input interaction. The individual parameters for the evaluation function are implementation specific, and are based on the native input modalities of the system, however they would involve the collection of known InteractionObjects as a minimum. In addition to the evaluation function, it is recommended Action include an attribute describing the physical interaction it is monitoring (e.g. a string set to "Proximity"), and an optional description of what the specific interaction is for this Action instance. For example, since the system knows objects' names, the description could be "when X is within 5cm of Y" (for a proximity-monitoring Action type).

An ObjectVisibleAction's purpose to monitor an object's visibility, for example, but needs to know which object(s) to monitor. As such, the core Action class provides a Set()

method, that is overwritten by individual Actions and allows for the customisation of each Action instance at run-time. For example since, the ObjectVisibleAction needs to know which object it should be checking the visibility of, the Set() method takes same parameters as the Evaluate() function as all available context data, but is only passed the data that is required to 'set' this Action. For the ObjectVisibleAction, this would just be a reference to the InteractionObject to which ObjectVisibleAction should respond when the InteractionObject is visible. References and relationships involving such objects are then stored internally within the Action class and used for comparison in future Evaluate() function calls. When Set() is called, the description for an Action can be updated to specifically address what that instance the Action is watching for, e.g. "when object A is visible". As just mentioned, the name of objects can be accessed and thus used in the description given the Action is passed a reference to the corresponding InteractionObject.

The logic encapsulated within an Action type is not just limited to simple logic. To enable the creation of new types of objects and to define how those objects interact with each other (as would be required to create the chemistry example), would involve some kind of PBE/PBD Action class that looks at the current state of any number of objects in scene and evaluates if they match a required configuration. Whilst the use of the Action class allows for the isolation of capabilities for detecting individual types of user interaction, it does not impose a limit on how complex that detection can be.

In the case of the chemical reaction, any hydrogen object could be used to trigger the reaction, and thus if the system was going to augment the objects used in the reaction to indicate something has taken place, the system needs to know which object(s) to augment. To enable this, the evaluation function of an Action type returns a collection of entities (InteractionObjects, touch instances, etc.) that were involved in triggering the Action. The specific objects returned would be dependent on the system's implementation and the types of interaction modalities the system supports, with the possible return types being any of the piece of data passed to the evaluate or set functions. In the case of the hydrogen-chlorine reaction example, the parent InteractionObject for the two hydrogen atoms would be used for the Set() call (given any objects part of that logical InteractionObject group can be used to trigger the Action), but it will be one of the individual InteractionObjects that is returned by Evaluate().

In addition to this, objects can fulfil different roles. If an object is part of a group, and that group along with that specific object are both required for an interaction, the result of the interaction needs to identify what role each object had in the interaction. For example was

the object involved fulfilling its own role, or only as part of a group the object was in? As such, to match which individual objects fulfilled different roles, InteractionObjects are returned with a corresponding InteractionObject instance, identifying the role they played in the Action. In the hydrogen example, the role would be a reference to the parent hydrogen InteractionObject instance. This way, the system not only knows which objects were used, but also what roles each of those objects had to fulfil in the interaction. In this case that the individual hydrogen instance that was returned as being involved in the interaction was returned with the hydrogen parent InteractionObject instance (since the single interaction was triggered based on any object in the parent). Using this approach, the system can enable generic substitution based interactions based on what object(s) partook in the interaction, and what role it fulfilled.

Whilst it is up to the individual Actions regarding what constitutes the desired interaction, the default recommended behaviour is that for interactions requiring more than one object, InteractionObjects can only fulfil one role within that evaluation. This means that if an Action instance was created requiring two or more InteractionObjects from the 'hydrogen' group, the Action will not evaluate to find one object as fulfilling the required role, and then check for objects fulfilling the second role, and match that same object. Any one object can only fulfil one role.

3.4.4.1.1 Monitoring Action State

Some interactions need to have some state representation to the user. This is different to the output, as the representation just provides information about the interaction state, for example, touch-sensitive buttons, sliders and dials all need to be visible to the user, something previous systems (such as Light Widgets) omit completely. Given all sensing for an interaction is encapsulated within the Action class, Action is the only component aware of:

- what interaction is actually being monitored, and
- what that interaction explicitly involves.

As a result, the creation/rendering of any representation for that Action must also be performed within that Action class. Therefore, Action is assigned a Render() method to render its state in some form. Much like the Property class' rendering function, the low level method with which this is achieved is implementation-specific. In implementations created during this investigation this is performed using OpenGL with a projected display, however in other implementations this might be to turn on LEDs, update a mobile phone display, etc. Again, the function is not to create the output of the interaction, rather it is

just to provide state feedback for the interaction, e.g. rendering a flat button, showing as indented when the Action is occurring (i.e. the button is being pressed).

In one of the sample implementations (described later), different button, slider, dial and 2D Touch Pad Actions created read in the object and touch locations in the scene, evaluating if any of them are within a user-defined region for the control. If so, the Actions update their internal state, triggering in the process, and renders controls with those values using OpenGL (Figure 31).

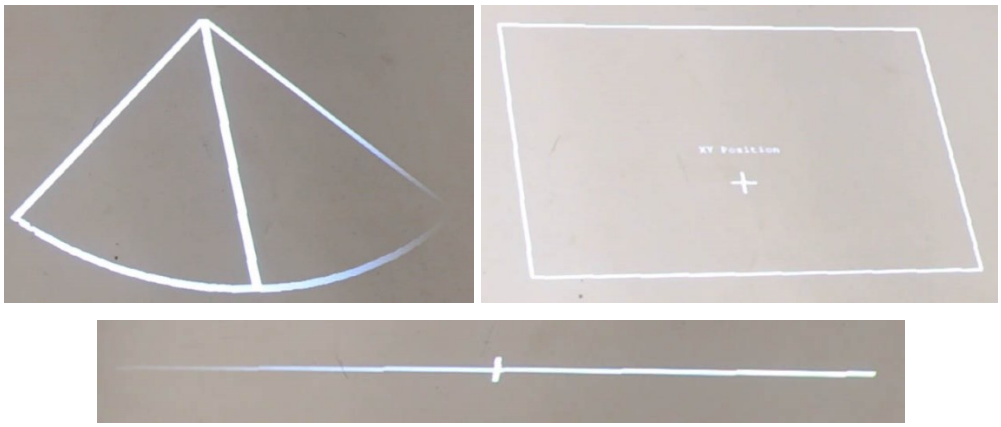

Figure 31 Rendering of the touch-based (partial) dial, 2D touchpad, and slider controls for the example implementation

3.4.4.1.2 Supporting Interactions that Generate Values

The Action class presented until now is 'triggered' by an interaction whenever Evaluate() returns instances of data for objects/touches involved in the interaction. However, not all interactions are nullary or binary. Valuator controls (e.g. slider or a spinner counter) generate dynamic values – the interaction they support is not simply 'occurring or not'. As such, to support valuator-style interactions where values resulting from the interactions can be used throughout the rest of the system (i.e. passed to output functions), there needs to be some method to disseminate these values. As with the state rendering, given the Action class is the component responsible for evaluating user interactions, it must also be the component responsible for generating any resulting values for the actions involved. As a result, a single Property reference (the same Property type used to describe InteractionObjects) is added to the core Action class as a "Value" attribute. For interactions that result in some value, an IntegerProperty could be used in the Value field, storing an integer value (or any other Property type) for the resulting interaction. Whilst any integer value can be stored, in these implementations for range-based interactions (slider, dials, etc.) the value is explicitly defined as being across a predetermined range

used by all valuators, i.e. 0-1000. Whilst this range is fixed, the range of values actually used as output for external functions can be transformed, as discussed later.

Depending on the nature of the interaction being monitored, more than one value may result. For example a virtual touchpad Action class needs to provide two (or three) different values (across the same predetermined range used by all valuators) as values for the X and Y (and possibly Z) location of a touch (or object). Given Properties can be hierarchical in structure, any number of other Properties can be encapsulated within the single Value attribute. For the touchpad example, they can either be encapsulated in a PositionProperty, or explicitly stored as three individual Properties using an instance of the base Property class for the Value the attribute. This choice of structure would be dependent on the implementation of the system, and how these values are used throughout the rest of the system, e.g. are they only used internally by the system (i.e. a flat structure), or does the user need to be able to navigate the collections of Properties to select them at run time when mapping a UI control to an application function (i.e. a hierarchical structure)?

Aside from providing an explicit value for the result of the interaction, the Value Property can also be used to provide additional information about the interaction. For example, whilst the pressing of a virtual button is communicated through the value returned from the evaluation function of the Action class, additional Properties might be used to indicate metadata about the event (e.g. the duration of the press, size of the object touching it, etc.).

3.4.4.1.3 Incorporating External Inputs

Whilst the use of the Action class enables new native functionality to be easily added without modifying large parts of the rest of the system, it does not address the need to incorporate input and interactions from external systems. To enable heterogeneous extensibility, new input modalities need to be able to be integrated as if they were native. To achieve this, a VirtualAction class is used. VirtualAction inherits from the base Action class and acts as a layer of abstraction between TAM and external systems. Using VirtualAction, the ad-hoc system can receive data from an external source, which is passed directly to a VirtualAction instance, where the data is processed and emitted through the VirtualAction's value Property, as if the VirtualAction natively sensed the data directly from a physical interaction. This allows data from external systems to be processed and propagated using the same pipeline as data that was sensed natively.

In order for VirtualAction to receive data from external sources, a number of attributes are required for the data type received:

- Name (string): User friendly name to identify the input.
- Identifier tag (string): Some unique identifier that is also passed in with the associated trigger data so that the data can be associated with the correct VirtualAction instance.
- Values (depends on implementation): Define what values are going to be sent into the system, so the associated VirtualAction can create corresponding Properties for each of them.

Using this information, when the user selects an external function to control that requires X number of parameters, only the native system Actions and VirtualActions that produce X number of values can be used as the input control for it. Using this approach in the example implementations, VirtualActions are defined in a single XML file, with only a single node required per external input (Figure 32). Parameters provided are required to be integer values.

```
<input name="Cursor Position" tag="mouseXY" parameters="2" />
```

Figure 32 Defining external inputs using XML in the example implementation

In the implementation, external applications can pass data to the VirtualActions via a network connect or executing by a program, SendData.exe, with the input's tag and associated parameter values as a space-separated string. Using this approach, all that would be required for an existing system that senses user interactions to be incorporated into to the ad-hoc system would be:

- one line of XML to define the function to the ad-hoc system, and
- one line of code in the associated application that passes the information to the ad-hoc system via SendData.exe (or via TCP).

Using a shell application and TCP connection in the implementation means no libraries or system-specific APIs have to be known user or supported by the software platform. Using this approach, external macros, scripts and almost any other executable code can easily be incorporated into the ad-hoc system without having to modify any code or recompiling of the ad-hoc system. Novel input technologies and modalities supported by external systems can be integrated into the ad-hoc system as if they were natively supported, with the user unaware of the abstraction that is occurring as they interact with multiple systems.

Given the forms of interactions are now only limited by what the end user's technology supports (and not what is available when the tangible system is designed), new input modalities such as speech, gesture, pressure, light, etc. can all be incorporated and used within the ad-hoc system as if they were originally supported. Active electronic devices (e.g. a mobile phone's touchscreen or sensors) can also be easily integrated. Given all that is required is one call to pass that data to the ad-hoc system using a variety of means, the focus for the external system's development remains on the electronic/active device itself, and not how to integrate the device to support tangible ad-hoc functionality.

Once the external input has been defined to the ad-hoc system, the integration of multiple systems is completely transparent to the end user. This allows the ad-hoc system to not just enable interactions for a single, specific set of native functions, but provides a flexible framework that can incorporate new technologies and modalities. This is required to not only adapt to the specific usage scenarios of the system, but also enable extensibility as new interaction modalities become available post-development.

3.4.4.1.4 Complex/Sequential Interactions

Whilst the core reasoning behind the encapsulation of interaction-specific logic in Action was to enable the incorporation of different types of interactions without modifying the core system, this implicitly implies that each Action class only support one specific physical interaction method, e.g. one Action supports object proximity, another Action supports rotation, etc. Whilst any amount of logic can be incorporated in Action to support multiple input methods, or even to use PBD/PBE, the different elements within possible physical interactions leaves a large space for possible combinations, e.g. rotation plus proximity. As such, short of developing an entire PBE system as a single Action where the user is unsure of how their actions are actually being interpreted, the user should be able to explicitly tell the system what interaction each Action type should sense. Whilst simple scenarios utilise only a single type of interaction, more complex ones also need to be supported.

In addition to this, scenarios three and four in Table 1 defined the use of 'overlapping' interactions, where multiple interactions may utilise common objects, or in some cases require the creation of sequential interactions. As such, systems need to support sequential interactions, allowing not only instances of an Action A and an Action B individually, but having Action A as a part of or as a prerequisite to Action B. For example, complex chemical reactions might involve water (requiring three objects to define/create H_2O first) plus another chemical that reacts with the water. However instead of having a separate

complex interaction that checks both if water exists and if water is reacting with other chemicals, the complex reaction might just have the water interaction as a pre-requisite, enabling its logic to be exclusively based on how the water reacts with the second chemical.

In order to ensure the isolation of different functions within TAM, an additional class, *Interaction*, is used to manage a single interaction that has been defined by the user. Interaction contains a reference to an associated Action instance that is used to evaluate if the interaction has occurred. In addition to this, the Interaction class also has an evaluation function with the same parameters as Action's evaluation function. To enable physical interactions to have pre-requisites, the Interaction class also contains a list of references to other Interaction instances which must have occurred before the Action associated with this Interaction will be evaluated. Essentially, an Action instance is stored in an Interaction instance, with the ad-hoc system only calling the Interaction's evaluation function (and not Action's). When evaluated, the Interaction instance runs through the prerequisite list, checking if each prerequisite Interaction in the instance has occurred, and if so, only then will it call Action's own evaluation function with the same parameters to check if this Interaction has triggered.

To avoid Interactions triggering multiple times when they are evaluated once by the ad-hoc system, and then once for each subsequent prerequisite check which they are part of, Interaction contains a simple Boolean "executedThisFrame" attribute, storing if it has been evaluated as true this frame. The use of a separate attribute means Interactions can be evaluated in any given order, without triggering or evaluating any Interaction more than once for a single frame.

3.4.4.2 Changing the Outcome – Defining the Patch Panel and Outputs

Whilst the Action class allows the system to detect when something has occurred (the input), the system needs a way to trigger a corresponding output function or perform a given response, either internally with the reconfigurable system itself as dedicated functionality, or to control additional functionality added by the end user by means of external systems. As such, from a programming standpoint, the interactivity within the system could conceptually be represented as a basic If-This-Then-That (IFTT) function (Code Excerpt 1).

```
void MonitorInteractions()
{

	//For a simple nullary function
	if (interactionCondition)
	{
		functionToExecute();
	}

	//For a simple valuator interaction
	if (interactionCondition)
	{
		leftSideOfAssignment = rightSideOfAssignment;
	}
}
```

Code Excerpt 1 Code sample showing conceptual If-This-Then-That functions for nullary and valuator functions

The individual Action classes act as the condition element (evaluating if an interaction has occurred) whilst also optionally providing a function call or some assignment of properties based on the interaction. However, as yet there is no way to disseminate the result of an Action instance to any specific function internally within the reconfigurable system, or to external functions. As such, a PropertyMap class was developed to manage the assignment of one collection of Properties instances to another collection of Properties instances. Each individual mapping of one Property to another is stored as an instance of a PropertyMapping class, storing references to two Properties as the 'source' and 'destination' of the assignment respectively. Whenever the PropertyMap is told to update by Interaction, it goes through the individual PropertyMap instances it contains, assigning the target Property as the value from the source Property. Using this approach, Property instances used to describe InteractionObjects can be updated based on the values of Properties of different types of Actions, fulfilling the second type of resulting valuator-type assignment that was in Code Excerpt 1.

Figure 33 Showing the flexible patch-panel approach of mapping n inputs to m outputs, as used by the PropertyMap

Whilst the core Property class defines the get/set functions as providing/assigning a Property reference, these may have been overwritten by type-specific Properties, e.g. an IntegerProperty class that actually assigns an integer value that can only be provided by another IntegerProperty. As such, it is up to the ad-hoc logic responsible for creating the

individual PropertyMaps to ensure the mappings generated are of compatible Property types. However, this is explicitly required anyway as the interaction logic will have to know which Properties in the system should be used for the source/destination in the mapping.

For example, a specific interaction for a board game might involve keeping a playing piece in a given area. In this case an InteractionObject would exist for the playing piece, with a corresponding Action that was set to monitor that object in a defined region. The resulting PropertyMap would contain a sole PropertyMapping, copying the InteractionObject's PositionProperty to its ColorProperty. Both of these Properties constitute three other IntegerProperties, however since the Set() function of Property is recursive, only a single PropertyMapping is required from the PositionProperty to the ColorProperty. The assignment of the individual IntegerProperties is performed recursively by the Set() function. To assign a constant value in a ProperyMapping, a dedicated Property of the given type would be created, assigned the required constant value, and used in the mapping. Using this approach, as the playing piece moves from an origin, its colour is augmented to reflect its movement.

3.4.4.2.1 Managing 'Toggled' Interactions

Given the assignment of one Property's value to another, the previous value is lost, meaning that even when the Interaction ceases to occur, there is no way for the previous value to be reset. Once a value is set as the result of an interaction, it remains until changed again by another interaction. Many of the interactive systems that could be created however allow the user to explore different configurations to solve a goal. This use of ART (Hacker, 1994) means that the interactions should support trial-and-error exploration (Sharlin et al., 2004) through epistemic and pragmatic interactions (Kirsh and Maglio, 1994). As such, interactions in the system need a simple way to be reversible or 'toggled', where the result of any interaction is only present as long as the interaction is occurring. If the user were to stop their action, the outcome should be reset to its prior state. In the chemistry teaching example, when chemicals are moved together they react, but stop when moved apart, with the default state resetting. This involves some form of 'undo trigger', as well as the ability to store the previous state of Properties before they are changed.

In order to support such functionality, the core Property class needs to also include:

- A 'history' collection storing the same class type as itself.

- A StoreCopy() function that returns a deep copy of the Property.
- PushHistory() and PopHistory() functions that either call StoreCopy() and store the result in a stack, or pop the top element off the stack, setting the Property value's to the recently popped value.

The core Property class may also require a GetNewInheritedProperty() function added that returns a new, empty Property object of the same type. This is required as given all Properties (regardless of their specific inherited type, e.g. ColorProperty, etc.) are treated as generic Properties, there is no simple way for the host system to know what specific type they are. GetNewInheritedProperty() is a virtual function overwritten by inherited classes, allowing the function to return a new object of the correct type. This can then be passed to StoreCopy() to retrieve a deep-copied duplicate of a Property, and its sub-Properties. PropertyMap and PropertyMapping also have push/pop functions added to store/reset the collection of Properties stored within each. The need for such a function would be dependent on the programming language features of the implementation.

Now that the system has a way of storing and retrieving deep-copies of object Properties, there needs to be a way to utilise this functionality. As such, the Interaction class is modified to add a Boolean "Toggled" attribute. When set to true, the Interaction object evaluates the associated Action, and if the Action's physical interaction had occurred, call PushHistory on the associated PropertyMap to copy the existing configuration before executing/applying the PropertyMap and thus overwriting current values. Interaction also keeps track of whether it was executed in the previous frame, as if it was executed in the previous frame but not in the current one, Interaction must call the PopHistory function to restore the original values of the Properties involved. E.g. if the user has moved the individual chemicals apart, they should reset to their original, non-reactive state.

Since the system is now aware if the Interaction was triggered in the previous frame, another Boolean attribute of Interaction, "Discrete", allows the Interaction to execute the PropertyMap either once when the Interaction first occurs, or continuously on the first and every subsequent frame during which the Interaction is occurring. By storing the history of Properties and the associated Toggled and Discrete settings, the system can provide flexible support for any number of different configurations. A learning and exploration system might require the use of toggled interactions to allow the user to explore the interaction space (such was previously discussed in this chapter), whereas other systems, such as that used in next section describing an 'infection' scenario where

entities in close proximity permanently affect each other, might require vales to be persistent.

3.4.4.2.2 Communication with External Applications

Whilst the use of a PropertyMap enables values internal to the system to be updated, they cannot as yet be disseminated to external systems, resulting in just another purpose-built system. As discussed previously, new functionality needs to be able to be incorporated in what is essentially a 'plug-and-play' approach. As such, in an ideal case external application functions would appear as if they were native functions of the ad-hoc system, with the end user unaware they were in a heterogeneous environment. In order to achieve this, a similar approach to how enable external inputs were enabled, with a VirtualProperty class created and utilised in a similar approach to the VirtualAction class. VirtualProperty passes values assigned to it to from PropertyMappings to external systems, and is defined according to five fields:

- Name: a user-friendly identifier for the function.
- Function Identifier: the technical data required to execute a function (e.g. an API function name, program name, etc.), along with the definition of any parameters required.
- Rate Limit: The delay (in milliseconds) between subsequent executions of this function. Can be zero for no delay.
- Scaling: Define any scaling that might be necessary, e.g. mapping a set 0-1000 range of valuator controls to a different range or scale. Can either be set to RANGE, CAP_MINIMUM, CAP_MAXIMUM, or EXPLICIT (any value).
- Scale Values: Any values (i.e. a minimum and/or maximum) required for the above scaling function to be applied.

In the example implementation systems, (discussed later) VirtualProperties are defined much like VirtualActions using XML (Figure 34). Functions are grouped arbitrarily according to relevant groups of functions (e.g. placing all audio-related controls together, etc.). Identical function entries can appear any number of times across different groups. Whilst their definition is implementation specific, the use of XML is appealing as it allows the various parameters to only be defined if required. Using this approach, IntegerProperty values replace the "%i" tokens in the parameters for a given application, allowing values generated within the ad-hoc system by user interactions to be disseminated to external systems.

```
<group name="Audio">
	<output name="Left Channel" ratelimit="0" execute="SetAudio.exe left %i"
		min="0" max="1000" scale="range"/>
	<output name="Right Channel" ratelimit="0" execute="SetAudio.exe right %i"
		min="0" max="1000" scale="range"/>
</group>
```

Figure 34 Output mappings for dual-channel audio control

3.4.4.3 Putting it Together

Now that the system has a way to check if a given interaction has occurred, and a separate component that is capable of executing changes for output, the Interaction class provides the encapsulating logic control between the two, checking if a given Action has occurred, and updating the PropertyMap as required maintaining references to associated Action and PropertyMap instances, acting as the glue between them that forms the actual patch panel 'link'.

Using this architecture, physical interactions can occur within the ad-hoc system using native capabilities (within an Action instance) or be sensed by an external system and passed in via a VirtualAction. Data values from these inputs can then be sent to external application functions via a VirtualProperty instance, which may then optionally actually send that information back in via a VirtualAction to create a feedback loop (Figure 35). This allows the integration of external logic, e.g. to support finite state machines.

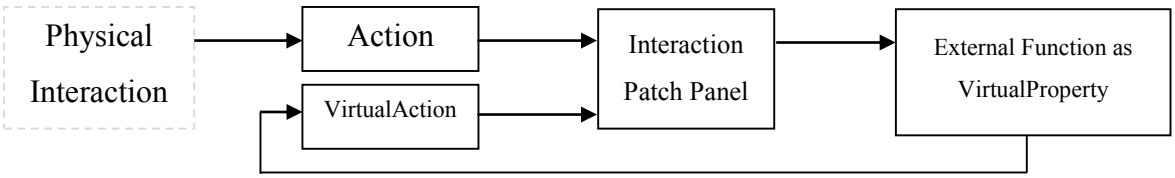

Figure 35 Information feedback loop

Essentially, Interaction's evaluation function is called by the system each frame, where it evaluates its associated Action (assuming any prerequisites have been met), and if triggered, executes the associated PropertyMap. As a result, the original IFTT interaction from Code Excerpt 1 statement is now akin to Code Excerpt 2 when described/implemented using the corresponding architecture functionality.

```
void EvaluateInteraction(InteractionObjects*, UserTouchData*)
{
	result = Action->Evaluate(InteractionObjects*, UserTouchData*)

	if (result)
	{
		PropertyMap->Execute(result); //Property map assigns Property A to B
	}
}
```

Code Excerpt 2 Code showing a basic If-This-Then-That function using TAM

Whilst the architecture to this point is suitable of support scenarios one and three in Table 1, the system cannot yet deal with conflicting updates in the system, with support for such scenarios discussed later.

3.4.5 Enabling Control of Data-Related Functions

The ability for the user to create controls for existing functionality so far has focused on creating controls that generate numeric or nullary output that is used to control external functions. When these controls are mapped to current GUI controls, they appear to enable support for a comprehensive set of tangible equivalents. However as previously stated at the beginning of this chapter, application data needs to also be controlled. Traditional GUI components such as a list do not generate numeric data, rather they enable the selection of one or multiple instances of data from a set, perhaps calling an associated function on the instance(s). As such, to enable the creation of tangible controls to perform similar functions, a number of capabilities need to be supported:

1. TAM needs to be able to monitor and track any number of different data instances that exist in an external system.
2. To enable tangible interaction, data objects need to have some kind of representation to the user (physical and/or touch-based).
3. There needs to be some way to associate/disassociate non-physical data objects with physical objects as proxies (when objects are available).
4. The user needs to be able to define mappings for any number of types of relationships within-and-between data objects as required by those functions, for example the application function might require a pair of data objects, a list, a hierarchy, etc.
5. External application functions requiring data objects as parameters need to be able to be triggered using tangible interactions (e.g. *delete data object X*, *associate data object X with data object Y*, or *merge data objects X, Y, Z*).

The first point needs to be supported at a core level within TAM, enabling it to track and store information about the data objects. Points two and three relate to tangible interactions that need to be supported within the system, with the last two points relating to support for data objects in the patch panel.

The previous descriptions of TAM have completely ignored any concept of application data or the run-time generation of additional content or functionality for use within the system. As a result, the TAM architecture needs to be modified to not only support the concept of data objects and their associated types, but also their dynamic nature and the

fact that the tangible representations are just proxies, and that changes to the data may occur outside the ad-hoc system (i.e. within the existing host application itself). The ad-hoc system may not have exclusive rights regarding what operations are performed on the data and when, for example data might be manipulated within the existing application and the tangible ad-hoc system simultaneously. As such, at most the ad-hoc system can only be a proxy, providing a tangible shell for data that is entirely managed within the external host application.

Much like controlling external functions, in order for TAM to enable the control of application data, the different types of data must be explicitly declared, describing what data object types can exist, as well as how those data objects can be used (i.e. what functions can be performed with them). In order to support such functionality given the dynamic nature of the application data, messages are exchanged asynchronously between the data host application and TAM, each notifying the other of any events involving data that have occurred (Figure 36).

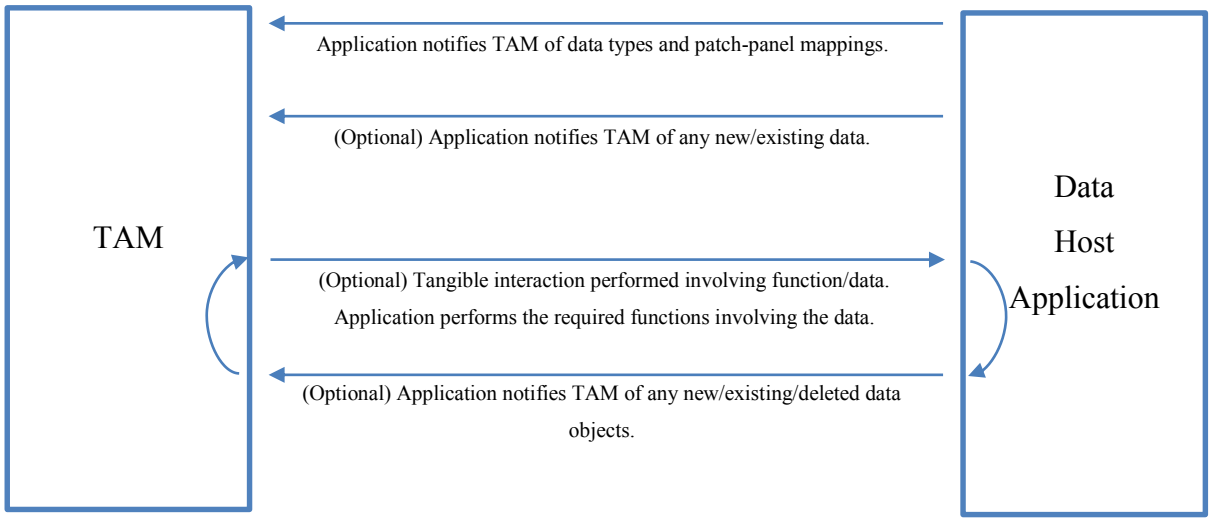

Figure 36 Timeline of interactions between TAM and the host application

Initially, the host application must notify TAM of the different data types that can exist, and any of the associated functions that can be performed using them (e.g. a delete object function, an associate function, etc.). To describe the different data types that will be used, two pieces of information are required:

- a unique *data type* identifier, and
- a user-friendly name to identify that type to user (if required).

For the previously mentioned tangible photo tagging application (Section 3.3), the types used would be an image and tag data type.

Once TAM is aware of the data types available, there needs to be a way to allow external functions to receive data instances based on interactions in the ad-hoc system. This is done by modifying the existing string that is passed to execute external functions. For example, when describing how interactions were supported, the string "SetAudio.exe left %i" was used to describe an external function to execute, noting the "%i" that indicates a single integer value was to be substituted in that position. Whatever the specific approach used is, it must be extended to support additional types. For example, the example implementations of TAM use the string "%data-typeX" to indicate that the function requires an instance of data of "typeX" as a parameter, with typeX being the one of the type identifiers that was previously passed to TAM. Whilst this describes what functions can be performed using the data, support for the physical data-based interactions that execute those functions are described later in this section.

Since TAM now knows:

1. the types of data can exist, and
2. what functions can be used with that data,

it can be notified of instances of data that exist within external systems. For each instance of data that exists, TAM needs to be notified of:

- a *type* (as previous defined) to identify the type of object,
- a *unique data object identifier* to identify the specific object instance in the host application,
- a user-friendly *name* to identify the object (optional), and
- some *representation* (e.g. an image) to allow state feedback and identification about the data objects to the user (optional).

Whilst the type and identifier are required by the system, the name and representation are only required to allow the user to identify one data instance from another. The use of a representation for the data object may either be passed in a common binary format (e.g. a JPEG image), or a URI to an external resource in the same format. It is important to note, much like the initial integration of external applications, the architecture does not dictate how the messages are propagated between TAM and the external system, as that relies on the specific implementation. However, this could utilise the previously mentioned methods for external communication, i.e. network connection or inter-process communication (IPC). An example of notifying TAM of a new instance of data is seen in

Figure 37, as used by the example implementation where a simple string is sent to TAM notifying it of the new data instance.

new *datatype, identifier, "Entity Name", newRepresentationURI*

Figure 37 Example of a new data instance message

Upon being notified of a new data object, TAM creates a representation of the data in the real world for the user to control. Given it cannot be guaranteed that an appropriate tangible object will be available to use a physical proxy, or even if one is whether the user wishes to use the object as a proxy, the representation will most likely be presented to the user in a non-tangible form (i.e. digitally displayed). Whilst the data structure used to store such a representation is implementation specific, it is not recommended to use an InteractionObject to store an intangible entity. This is because if at a later point in time the user wishes to associate a piece of data with a physical object as a proxy, the physical object will already have its own InteractionObject representation, and thus they must be merged or one deleted, possibly affecting/orphaning existing Interaction instances. As such, it is recommended to create a separate entity (e.g. using a DataObject class) that can be interacted with using only non-tangible means (e.g. touch).

As interactions involving data occur (either using the tangible ad-hoc system or external application), the external system notifies TAM of any updates, deletions or new objects that have occurred either explicitly by the host application or as a result of tangible interactions performed by TAM. These updates may change the user-identifiable name and/or representation for existing data objects, using whatever communication methods are used to link the systems. The individual messages passed between the systems must include:

- the identifier of the data object to update,
- (optionally) the new name for the data-instance, and
- (optionally) the new representation for the new data instance.

Object identifiers by definition cannot be changed, as any change should be performed as a deletion and a creation. An example of an update message implementation can be seen in Figure 38, where like the notification for a new object, a simple string is sent to the TAM system notifying it of any changes to its name of representation.

update *dataObjectId123, "New Name", newRepresentationURI*

Figure 38 Example message of an external system updating properties of data objects known by TAM

For new data objects that have been created as a result of a function call from an interaction performed by the user, the external system can also provide the name of the related function in the patch panel that was the cause of the new object as a third parameter. Using this, TAM can try and place the new data object relative to the location of that function in the system. For example, if the user were editing a video and used a button to call a 'split' function to split a video clip at a given position, the new data objects resulting from that split may be passed to TAM along with the name of the split function. With this, TAM can set the initial location of these objects to be near the split button control that generated them. Using this, the function names can be used as a hints regarding what Action a data object has resulted from, and thus where the data object should be located to give added meaning based on its context. The use of hints is only supported for new objects and is not guaranteed given data instances may be associated with physical objects, and randomly changing the position of existing data instances from their existing locations as laid out by the user creates the possibility to confuse the user.

When a function involving the data is executed by TAM (e.g. a 'pairing' interaction between two data instances), TAM notifies the external host system using the same approach as used for previous mappings, however instead of passing any value(s) as the parameters, the identifiers of the two data instances involved are passed. For example such an implementation could be as in Figure 39. This message is then interpreted by the host application and those two data objects passed to the existing function within the external application that performs the association.

associatePhotoWithTag *photoId123, tagId456*

Figure 39 Example message of TAM notifying the external system to execute a function involving data objects

Using this approach, only two pieces of application-specific logic must be written inside an external application that wants to enable tangible control using TAM:

- One is the notification to TAM for any new/updated/deleted data objects. These calls are simple located where the new/update/delete events occur inside the application (most likely in the associated dedicated functions).
- The other is a single function that receives messages from TAM, before interpreting them, accessing the referenced data objects and passing them to the existing functions to perform the required task (e.g. passing a data instance to a delete function).

Whilst the use of a globally unique identifier for data objects initially appears to imply an overhead on the host application, it is actually a simple solution in itself. External systems can use the memory location of a data object as its identifier. Aside from being unique to the external system and removing any overhead in creating and mapping a custom identifier, it also means that when a function inside the external system is called with a data object's identifier as a parameter (which is actually just the object's memory location), the system can simply de-reference that object and pass it to the required function. If multiple external systems are being used, an external system-specific identifier could be prepended to the identifier to mitigate the remote risk of multiple external systems using identical memory locations for identifiers of multiple data instances.

3.4.5.1 Interacting with Data

To allow the ad-hoc system to detect interactions involving instance of data, the evaluation function of the core Action class is modified to not only take in information about physical objects and touches within the scene, but also data objects of which TAM is aware exist. This may be in a number of different ways depending on the implementation. Data may be associated with physical objects as an attribute of InteractionObjects, or exist as a touch-based entity in the system, or as an entirely new type. If the implementation allows for data objects to be associated with physical objects or exist solely as touch-based entities, it is recommended to have them as an attribute of each physical object (InteractionObject) or touch instance type, instead of their own 'data object'. This allows any existing physical or touch-based entity in the system to be associated with a data object as required. Using this approach, the signature of the evaluation function does not change, as it is still only accepting physical object and touch data as its parameters, with the instances of data being attributes of the physical object and touch data.

Once the evaluation function in Action has a way of being notified of data objects within the scene, little else is required. Individual Action classes are then written to support the different types of physical interactions that can occur with data instances. For example, a SpatialDataAction might check if any two or more data objects of the required type(s) are next to each other, and if so, trigger that Action and allow them to be passed to an external function, e.g. to associate them. However, in order to do this, the data objects identified by Action need to be able to be passed through the patch panel to the receiving VirtualProperty that actually executes the function.

No operations on external data are assumed to exist at all as a design decision. Whilst basic data operations such as new, delete, copy, etc. could be assumed to be universally applicable, the ad-hoc system cannot make the assumption that a piece of data should always be able to be created/deleted at any point in time. Given the complete lack of context or state awareness by TAM about the data instance and what the data actually represents, TAM cannot assume any level of functionality about it. In addition to this, if the system were to support a number of basic operations, it would also mean supporting universal tangible interactions to perform them, which may not be desirable given how the user is wanting to interact with the system. For example, what physical interaction can universally be used to mean "new object of type X"? Instead, even primitive functions must be explicitly defined and controlled by the external system using the patch panel. By declaring functions in the patch panel that take single instances of data of a specific type as a parameter (discussed next), the user can then create those primitive functions as required, using the desired physical interactions to perform them. This ensures that if some operations should only be supported for some data types but not others, the patch panel, and ultimately the external system, retains control. Even if a new data object function is called by TAM, the external system can still make the choice to not create it based on the external system's internal state. This does not affect the TAM system, as any 'output' or response to that function is merely received as any number of distinct new/update/delete data messages that are received at any time after the function is called. If nothing is received, nothing changes, and interactions continue as normal.

3.4.5.2 Supporting Data Objects in the Patch Panel

When exploring how external functions should be defined in the patch panel, there is a tension in the balance between what should be supported in the ad-hoc system versus what should be relegated to the external application logic. For example, should the ad-hoc system merely notify the external application whenever two or more data instances are next to one another? Or should the end application say "only notify me when you have a collection of X number of objects, with some of type Y and some of type Z", or different combinations thereof? On the first extreme, the tangible ad-hoc system merely just becomes an abstraction for sensing tangible interactions, with the external application still required to monitor every frame and interaction performed by user, filtering them for valid input configurations. The other approach however offloads such tasks to TAM, enabling the external application to only be notified when the ad-hoc system believes it has a valid configuration of data objects. As such, in order for the ad-hoc system to identify what data objects can be used for different functions, TAM requires the data type

identifiers in the function declaration, which allows the host application to request a collection of object(s) of types X, Y, Z for a function. This then allows the application to only be notified when TAM believes it has a valid set of data instances on which to operate.

As such, TAM takes a middle ground between the two extremes, where a host application can request either a single data object of a given type or a collection of data objects of a given data type. This first approach allows an external system to request that a given function in the patch panel be passed the identifier of a single data object of a given type. A simple mapping from the sample TAM implementations uses XML to achieve this, as can be seen in Figure 40, where the "%data-typeX" is replaced by the selected data instance's identifier at run time.

```
<output name="Delete Object" execute="NotifyApp.exe DeleteObject %data-typeX" />
```

Figure 40 Patch panel declaration to receive a single data object

To pass the identifier of data object to the patch panel, another fundamental data-storage type was defined within TAM. Previously, the use of the IntegerProperty class allowed controls such as sliders and dials to generate values that could be passed to external application. To support the identifiers used by data objects, a StringProperty class is used. StringProperty sub-classes the abstract Property class, just as IntegerProperty does however stores a string value instead of an integer. This approach may vary by implementation. For example an integer could be used to represent a data object's memory location (depending on the memory reference size). However, the use of a generic string type ensures universal support, especially when multiple external systems using multiple platforms at utilising the same ad-hoc system.

Using this approach, the VirtualProperty class does not need to be modified aside from enabling the "%data-typeX" token to be replaced with the assigned data instance identifier.

3.4.5.2.1 Collections of Objects

Where collections of objects are required, either a hierarchical structure of VirtualProperties can be used (with the entire hierarchy assigned a hierarchy of object identifiers from a relevant Action), or multiple data object identifiers structured/concatenated into a single StringProperty. For example, a video editing application may require any number of video clips for a function. To support this, different parameter types are used in the function mapping, for example instead of having

a “%data-typeX” parameter, it would be “%dataList-typeX”. Depending on implementation, the source data-based Action might generate a list of data objects involved as a single concatenated string, which is assigned to the VirtualProperty and used to replace the “%dataList-typeX” parameter in the execution string.

As just mentioned, since Properties can be hierarchical, a hierarchy of data instances generated by an Action class can be mapped to a hierarchy of VirtualProperties, allowing a hierarchy of data objects to be communicated via individual output functions to an external system. In addition to this, different collections can be created by encoding the identifiers of multiple objects in a single StringProperty (or implementation equivalent). For example a DataListAction might emit a single StringProperty that contains multiple data object identifiers as a comma-separated string values (CSVs), enabling a single VirtualProperty to emit data about multiple data instances that can then be processed by the external system. More complex structures aside from CSVs, such as defining a hierarchy using pairs of identifies in a CSV format. However, the individual mapping used can still force the creation of different collections in different ways. For example, a host application could force the creation of a list one object at a time, by having the function mapping accept only one data object. This means the user could only add one object at a time, with the list actually being created and stored by the host application and not TAM. This gives flexibility to the role that TAM should have if the host application actually wants greater to influence how certain operations are performed.

3.4.6 Resulting Architecture

The final resulting fundamental architecture (as was shown earlier in Figure 28) comprises six core components/classes that enable the creation of extensible ad-hoc systems that enable support for ad-hoc interactions. The Tangible Application Host object in Figure 28 represents the implemented tangible ad-hoc system, which is required only to maintain references to the InteractionObjects used in the scene, and the Interaction instances that have been created.

To summarise the final architecture, InteractionObjects represent physical objects in the scene and are described by any number of Properties, whilst encapsulating any number of other InteractionObjects. The core Property class is inherited from to create specific Property types, allowing for the storage and optional rendering to the user of object attributes. In addition to this, the core Property class provides a hierarchical structure within which more complex Properties can be structured.

Subtypes of the Action class are passed all available data about the current scene and evaluate if specific interactions have occurred, providing a single unit within which all interaction-sensing logic is contained. Values and data generated as a result of the interaction are stored as associated Property instances within the Action. An Interaction class then utilises an Action instance to monitor for specific user interactions based on the Properties of various InteractionObjects or other interactions in the scene. If triggered, the Interaction's associated PropertyMap is executed assigning each pair in individual PropertyMappings from a given Property instance.

Entire interactive systems can be created using these components, without relying on external functionality, as can be seen in Appendix B – Simple Application/Interaction in Code.

3.5 Discussion

This chapter has provided a high level description of a patch panel approach that enables the ad-hoc creation and incorporation of novel input controls as well incorporating inputs from external systems, allowing for the definition of internal application logic whilst allowing external systems to be controlled. Different requirements were identified in order to support the tangible manipulation of application logic, functionality and data. Extensibility is enabled through the TAM architecture that enables the abstraction and modularity (using the Action class) of system inputs and outputs, allowing the patch panel (using a generic Property type and specific sub-types) to enable the user-defined, run-time mapping between them. The use of VirtualActions and VirtualProperties allows for external systems to be integrated, whilst appearing to be a native part of the system from the user's perspective. Using this approach, extensibility for heterogeneous environments that is completely transparent to end users can be supported, as is described in further detail in later chapters. This allows users to define novel user controls involving values, data and logic within their current context and modality.

Chapter 4. Pilot Studies Exploring Ad-hoc Interaction

To explore how users would ideally like to interact with tangible ad-hoc systems, two pilot studies were conducted at various points throughout the investigation. These studies were conducted before any system design or implementation was conducted for the corresponding classes of interaction, and as such, were used to provide direction for subsequent development. The first study explored how users would create tangible controls for a known set of existing systems functions. The second study explored the ad-hoc definition of application logic to a tangible context. The primary goal of both studies was to explore how users would go about creating different ad-hoc elements, as well as to identify the kinds of interactions and methods they would utilise to create them. This chapter describes both studies, their goals, and outcomes.

Whilst the use of voice (and thus voice recognition) to describe such systems would be expected as one of the primary means of interaction, both studies prompted users for alternate methods in addition to voice. Despite voice recognition having made great advances in recent years to the point of seeing adoption in consumer devices such as Apple's Siri[3] and Microsoft's Xbox[4], unstructured interactions still pose a problem, and as such alternative inputs modalities are still required.

4.1 Pilot Study Exploring Ad-hoc User Controls

This section provides an overview of the study into how users would create user controls from arbitrary objects available in a home/office environment. The study was designed to elicit not only how users would create ad-hoc user interfaces, but the actual kinds of UIs they would develop. Would they want to use gestures, touch interaction, physical objects, etc., or a mix of those techniques? The process of creating and editing the UI, as well as the types of controls suggested by participants could then be used for subsequent design and development. As such, this section provides an overview of the design of the study and the final results that guided the future development of a system to support users in creating the kinds of UIs described in the results.

4.1.1 Design

The design of the study was similar to that proposed at the conclusion of work by Cheng et al. (2010), and a study conducted by Henderson and Feiner (2010) for their work on

[3] http://www.apple.com/au/ios/siri/

[4] http://support.xbox.com/en-AU/xbox-360/kinect/speech-recognition

Opportunistic Controls. As the focus of the work on Opportunistic Controls was on aiding mechanics, the selection and usage of environmental controls was all authored prior and offline. There was no ad-hoc component, rather the system allowed designers to pre-author the controls that leveraged the affordances of physical objects (e.g. a bolt as a dial) that the designer knew would be accessible. However at the conclusion of the work, Henderson and Feiner designed a study where users were given a number of different devices (TVs, VCRs, etc.) and asked to select how they would control the different functions associated with each. This study was designed using the same approach, designed to elicit how users:

- get the system's attention when they want to do something,
- select what kind of control to create,
- select a function to control,
- edit or delete controls once they have been created, and
- the kinds of controls users would make from everyday resources.

Participants were seated at a table covered in writable butcher's paper, with a whiteboard alongside, attempting to 'surround' them with writable surfaces to enable interaction on any surface. The table had a collection of random items (both active electronic and passive) on it that one would find in any home/office environment – pens, mugs, mobile phone, Post-It notes, scissors, rubber bands, chewing gum packets, etc. This was the same process used by Cheng et al. (2010) where photos were taken of users' desks to identify what objects were available 'in the field', with objects of possible use for later interactions identified. Users were asked to create controls for the following applications using both nullary and valuator-based inputs, using both absolute and relative values:

- a PowerPoint slideshow (start, next, go to slider number X, etc.),
- a digital video editor (clip, join, play, order clips, etc.), and
- a generic 3D game (joystick, accelerate, etc.).

Participants were asked how to control systems both within their reach (e.g. the video editor) and outside of their reach (e.g. the PowerPoint presentation) for three types of systems; omnipotent ('all-seeing'), tangible-only and mobile device-only. For example, participants were asked to create a control for navigating the slides in a slideshow using only physical objects. They were prompted on:

- how they should get the system's attention to create a new control,
- how they would tell the system about which object(s) they wanted to use,

- how they should be able to describe those objects to the system, and
- how those objects would be utilised to control a given function.

Once the participant had created a tangible control, they were then then prompted how they would control the same function but only using touch-based controls, only a mobile device, etc. The progress of the study was not structured and was based on the individual controls created and the dialog with the participant. Any controls or system content displayed was created on-the-fly using the writable surfaces and/or available materials, using a Wizard-of-Oz (Maulsby et al., 1993) (WoZ) style approach. Participants were continually prompted for their thoughts and reasoning, e.g. "how would you get the system's attention for X" or "how would you select which application function you want to control", as they described their process for creating the associated controls.

4.1.2 Results

Three participants took part in the pilot study, with only one having a background in computer science. An example of the study setup is shown in Figure 41. The majority of approaches were suggested individually by all participants, suggesting strong support for them.

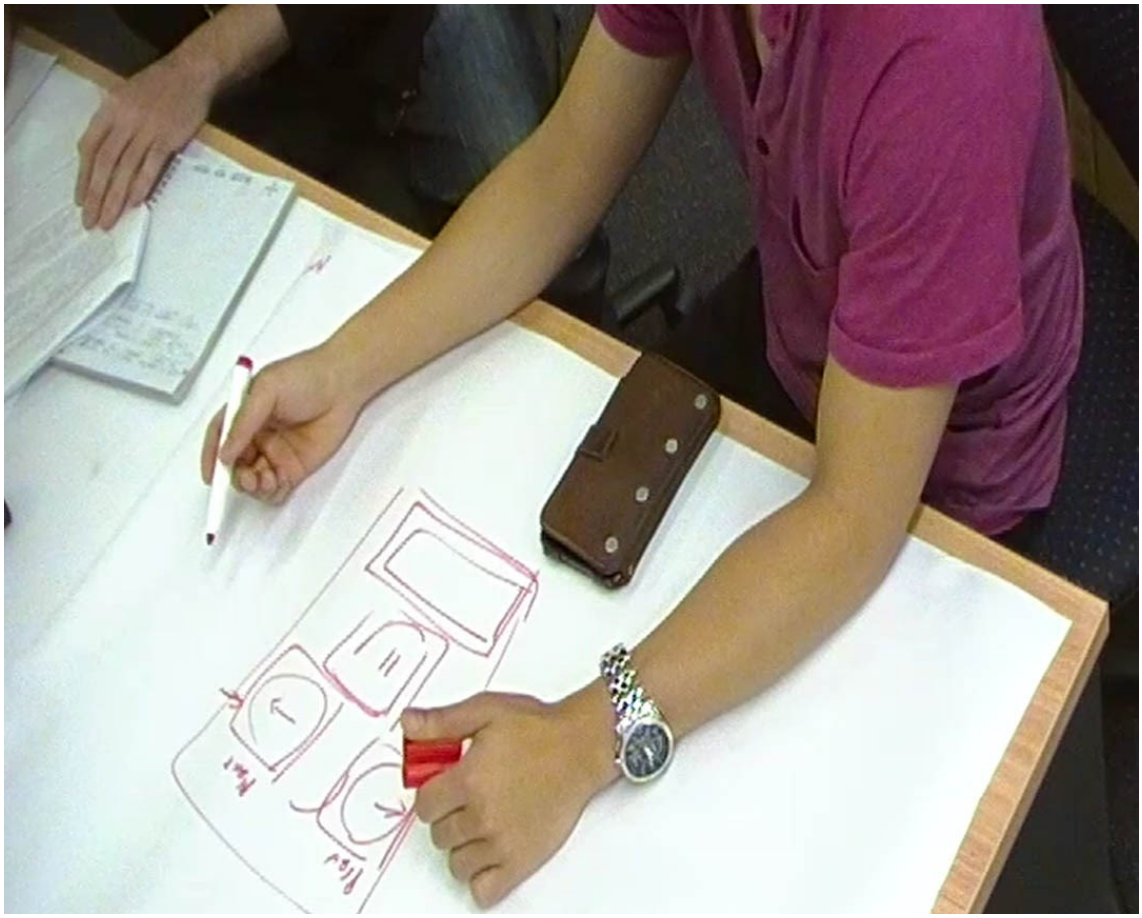

Figure 41 Video frame showing a participant taking part in the study

4.1.2.1 Process Order

All participants created UI controls using the same process. The participants first selected an 'output' function to control, before then selecting the input control for it. When prompted why, it was said the selection of what function they were going to control would change the types of inputs that could be used to control. In addition, one participant phrased it as though thinking about "what they wanted to do, not how to control it", so it made sense to "offload" that information into the system as soon as possible. The use of a wizard was suggested, guiding the user through the possible choices at each step.

4.1.2.2 Getting System Attention

Participants requested a different mode of interaction to "get the system's attention" to create a control, versus that being used for normal interactions as to avoid confusing the system. A number of different approaches were suggested to initiate the creation of new controls:

- Waving.
- Using a 'sweeping arm' across the table to define the "area where the interaction will take place".
- Placing an open palm on table, "making contact" with the system.
- Tapping on the table as if waiting for the system to do something.
- A big, 'record-style' button (which participants preferred over voice or gesture).

The use of a physical button was suggested by all three participants, with one noting that the system should have a button the user can press so the system can "watch/learn from" the user's subsequent actions. One participant noted the use of an idle countdown timer to reset the system/exit out of the existing task if the user interacting using it, alleviating the direct need for any form of 'cancel' control.

4.1.2.3 Selecting the Output

Participants wanted to select an output function to control using a only two different approaches; saying the name of the function they want to control (one participant) or selecting it from a set of options displayed by the system (all participants). When asked how to select from large sets of functions, participants said they should be grouped according to their related functions.

4.1.2.4 Types of Input Controls

When asked how the user should be able to select the type of control they want to create, the use of a virtual button was suggested, with participants noting that the buttons should

have labels such as “Add Dial”, “Define Gesture”, etc. Upon selecting the type of control to create, the system should provide feedback regarding the corresponding user input required to create the control. Aside from the creation method, a number of different input controls were created by participants for each of the omnipotent, tangible and mobile-based scenarios.

4.1.2.4.1 Omnipotent

For an omnipotent, all-seeing system (where any user action could be sensed and correctly interpreted), a number of different input controls were used depending on the type of control being created:

Nullary Controls

- Clapping
- Waving
- Arms opening/closing
- Pointing
- Virtual Buttons
- Voice recognition

Valuator Controls

- Use an object as touch-based valuator (e.g. pick up a pen and slide finger across it).
- Virtual touch-based slider and dials.
- Move an object within a defined area.
- Use the distance between the user’s hands, for example set an initial ‘unit’ distance between hands, then move them inwards to define value relative to starting distance.

4.1.2.4.2 Tangible

Like creating controls for the omnipotent system, a number of different controls were created:

Nullary Controls

- Place a given object in a given location – akin to “placing the key in the ignition”.
- Have a glowing “swipe to start” option, like recent mobile phones.

- Have a single object as the 'select' object, and place it into different regions to select the corresponding region or associated function
- Associate an object with a function that triggers the function when touched by the user.

Valuator Controls

- Rotate objects left/right to select a value "like a control knob".
- Define a scrollbar using two objects, then touch in between them to set a value.
- Display/project a number-pad on the surface.
- Utilise some affordance of any object to scroll, e.g. touching a position on a pen.
- Touch-based slider on the table.

Given one of the main problems in tangible interaction is discerning purposeful, discrete interactions from accidental ones (as previously highlighted by Bowman et al. (2001)), it was suggested that interaction using objects only be valid if being held by the user when the interaction occurs. This would assist in avoiding false-positives for interactions when the user is simply reconfiguring that layout of objects in their workspace.

4.1.2.4.3 Mobile Device

When asked to use a mobile phone (an electronic and only means to interact with the system), a number of interactions were present:

Nullary Controls

- Shake device.
- Tap device screen.
- Wave with device held in hand.
- Flip phone over on table.
- Knock phone on table.
- Press on-screen button.

Valuator Controls

- Virtual slider on the screen.
- Virtual dial on phone, "like an iPod wheel".
- Move phone left/right or up/down to set value using the phone's accelerometer/gyroscope.

4.1.3 User Controls Summary

A number of user controls across different modalities were suggested by participants. The majority of interactions suggested utilised gestures, regions of interest/buttons on the surface, rotating/sliding/touching/tapping objects, and using virtual controls on digital objects, similar to the types of controls suggested in the OC study. There was a mix of touch and table controls, where physical objects were placed in pre-defined regions on the table to trigger associated functionality. The most preferred method for creating controls was to use a button to first get the system's attention, before selecting the function to control from a number of relevant groups, and then selecting a user control suitable for controlling the function.

4.2 Pilot Study Exploring Logic for Ad-hoc Interactions

As previously mentioned (Chapter 3), support for ad-hoc interaction involves more than simply assigning roles to the objects used in the scenario. Interactions between these objects can occur, creating mental overhead in tracking the current state of each object involved. To explore how users would create such systems digitally, a second exploratory pilot study evaluated how users would, ideally, like to interact with such system – i.e. how would they assign roles, communicate logic, states changes, etc. This section describes the design of the study, its implementation, along with the results.

4.2.1 Design

Different users interact in a multitude of different ways. Some think out loud, some describe problems explicitly using logic, some describe a scenario by acting it out, etc. Existing digital systems only support a small subset of the interaction modalities used by humans, whilst also requiring strict adherence to the required communication protocol. To describe a problem or scenario to another entity, person or otherwise, the user must first translate the problem into a structure familiar to the receiver, and then communicated to the receiver using a supported modality and format/protocol. In an ideal scenario, the user would be able to describe a scenario to the system as if they were describing the scenario to another person, i.e. any modality, language, gesture, etc. would be correctly interpreted and understood.

To understand how users would describe scenarios to a third party, participants were asked to create a tangible version of the Fox, Duck and Grain (FDG) problem. The FDG problem is a logic based, 'river-crossing' puzzle where a fox, a duck and a bag of grain are located on one side of a river, with a boat that can only hold one item at a time. To

solve the puzzle, the player must get all three across the river without violating any of the following rules. If at any point in time:

- the fox is left alone with the duck, the fox will eat the duck, and
- if the duck is left alone with the bag of grain, the grain will be consumed.

Moving only one object at a time across the river, the player must successfully move all three items across the river. Whilst such a problem can be solved mentally, it is often beneficial to 'play it out', creating a mock river with tokens to represent each item. By trying various combinations of moves, the player can eventually solve the puzzle (Table 2).

Step	Move	Bank A	River	Bank B
0	Game Begins	Fox, Duck, Grain	Boat	
1	Duck to Bank B	Fox, Grain	>>> Boat	Duck
2	Empty boat to Bank A	Fox, Grain	Boat <<<	Duck
3	Fox to Bank B	Grain	>>> Boat	Fox, Duck
4	Duck back to Bank A	Duck, Grain	Boat <<<	Fox
5	Grain to Bank B	Duck	>>> Boat	Fox, Grain
6	Empty boat to Bank A	Duck	Boat <<<	Fox, Grain
7	Duck to Bank A		>>> Boat	Fox, Duck, Grain

Table 2 Solution to the Fox, Duck and Grain puzzle

In creating such a system, a number of tasks would have to be implicitly carried out by the participant:

- Selecting abstract physical objects to fulfil a given role.
- Assign roles to those physical objects.
- Create a number of tangible (and possibly intangible) objects or regions as the two river and banks, describing both positive and negative outcomes.
- Describe a number of simple rules to the system involving physical objects and possibly intangible objects.
- Describe rules based on quantities and classes of objects, i.e. only one item on the boat.

Whilst participants might have been directly aware they were making decisions regarding the use of different modalities and types of input, the large array of possible options meant that the results could still be used to describe what they found to be most intuitive or 'best' to use.

The configuration of the study environment was similar to the previously described study. A desk was covered in writable materials, with 'everyday objects' placed on top of it like the first study. Participants were asked if they were familiar with the FDG problem, with the rules explained verbally regardless of if they had heard of FDG prior.

4.2.2 Methodology

All participants followed the same methodology:

1. Participants were seated at a large desk, covered with paper surfaces along with a variety of materials found readily around any office or home environment (markers, staplers, phones, blocks, etc.) to be used for the puzzle.
2. The rules of the puzzle were then explained verbally regardless of if the participant had FDG prior knowledge of the problem. No solution to the puzzle was provided, however the basic premise was described, e.g. "the first move would be to move the duck since the fox will not eat the grain".
3. Participants were then asked to describe the rules of the puzzle using available objects as proxies for the required roles as if being watched by an 'all-seeing' machine. Where needed, responses required by the system, graphical or otherwise, were improvised at a low level with the available resources (using a WoZ approach), as has been leveraged previously for exploring multimodal interaction (Salber and Coutaz, 1993). Participants were told they could employ anything available as they saw fit.
4. Based on the current feasibility of the modalities/functionality of the 'all seeing' system that were being used by the user, users were prompted for 'second-best' alternatives after describing their first choice modalities. For example, if a user were heavily reliant on voice-recognition, they were prompted for a second form of input to use if voice recognition was not available. Even if the methods they chose were feasible, they were eventually prompted for alternatives, as to explore the complete interaction space.
5. Once the participant had created their implementation, they were then asked to solve the puzzle using their implementation. As they played it out, they were prompted for what kind of feedback they should receive from the system when rules were broken or to indicate that the puzzle had been solved.

Participants were recorded on video during the session and asked to think-out-loud as they worked through the scenario, describing the different goals they were trying to achieve, how they were going to achieve them and why. In addition to the video, notes

were recorded by the supervisor following prompts to the participant throughout the session.

4.2.3 Results

Six participants (four males, two females) took part in the study, two of which had a background in computer science. Results were collated by task/focus, before identifying any common/unique themes present in the results. An example of the setup for the study can be seen in Figure 42.

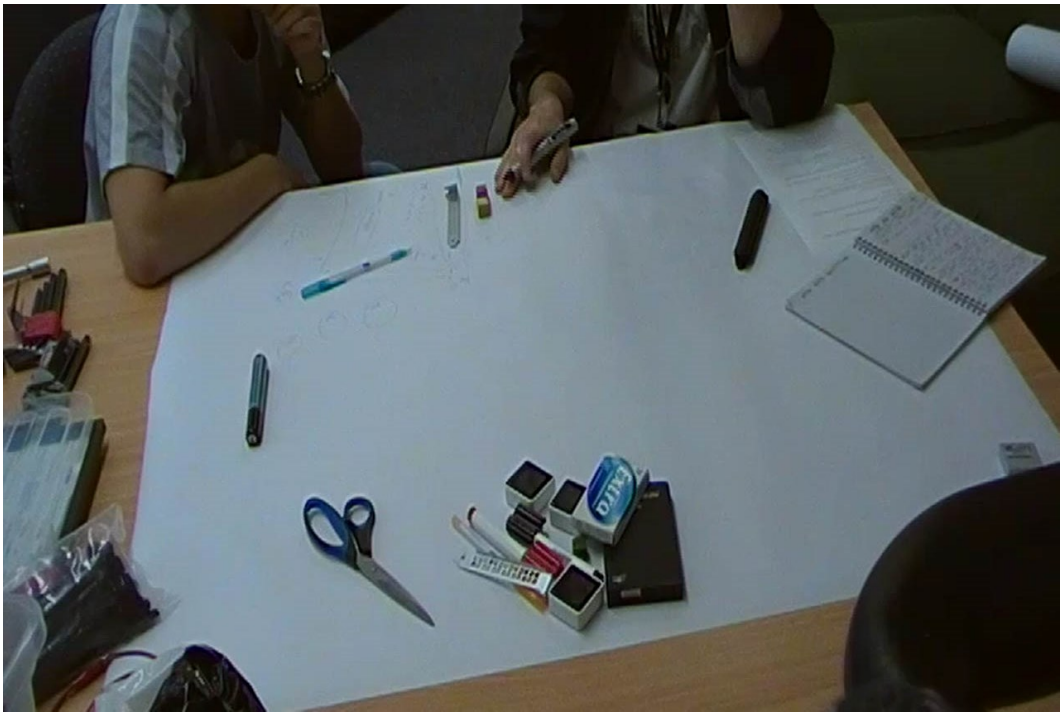

Figure 42 Video capture of a participant in the study

4.2.3.1 Object Choice

Participants used a mix of physical objects and virtual regions to define the banks of the river, with the majority (four) using virtual regions. Of those four, three drew lines on the table to indicate the banks (some with arrows on the ends to indicate the regions extended infinitely in each direction), labelling the bank on the 'land side' of the line, the other using rectangles.

For selecting objects to use to fulfil the roles in the puzzle, all participants chose physical objects to use. However one participant used physical objects for the fox, duck and grain, but used a movable 'virtual boat' that they drew by hand and expected to be displayed by the system.

4.2.3.2 Assigning Roles

Surprisingly, only one user used voice to define object roles, pointing to an object and saying "this is an *X*". When prompted about alternative methods for definition, the participant wrote the names on the surface. Whilst being a second choice for this participant, all other users used either short abbreviations or names to identify objects as a first choice. Participants would write names/roles for objects them (four participants), writing the name in front of them before 'dragging' that name to the object (one participant), or physically writing on the objects (one participant). One participant said the system could show a keyboard as needed. When prompted about the use of initials for objects, one participant commented that although they were using them for this scenario, they would not work for larger systems which would require the object's full name. Aside from writing the name to identify the object, participants said regions could be defined to introduce/delete objects in the system, not unlike the approach used in DuploTrack (Gupta et al., 2012) where different regions are used to manage objects.

In addition to using the name, three participants suggested the use of projected icons to help identify objects in the scene. One participant even suggested the system use Google Images[5] to search for the object's name, and either just pick the first available image or allow the user to pick from a small subset of the results. In addition to this, one participant suggested having a small 'drawing region' where they could sketch things that could be used as textures in the system – essentially a "digital etch-a-sketch[6]". This was supported by another participant that wanted to write on the objects with "virtual ink".

4.2.3.2.1 Repetitive Creation

Given the requirement to manually introduce each object to be used, participants highlighted the need for the system to reuse existing information to assist in the integration of new objects. This came down participants saying the system should either use some form of physical detection on the object to see if it had similar features to previously introduced objects, or to manually prompt the user if the object was based on an existing one. As phrased by one participant, "(the) system should suggest as much as possible". When introducing objects, participants noted that they should be defined according to common features first, before then defining the differing features.

[5] http://www.google.com/images

[6] http://www.ohioart.com/brands/etch-sketch

4.2.3.2.2 Groups/Types/Classes of Objects

After being told that they would have to instruct the system about every permutation or combination of valid/invalid moves for the objects, five participants asked if the system could detect if objects are of the same type based on their shape – basically wanting to define types of objects based on their physical properties. These types could then be used to simplify creating the interactions. Interestingly, one of the participants without a computing background referred to these as "different classes of objects".

The use of colour was paramount to participants in identifying the different objects and their roles within the system. Four participants used colour to convey similar roles/properties – e.g. both sides of the river were the same colour. One participant used colour in conjunction with icons, using icons to identify the type/group of the object. Two participants overloaded the use of colour to also be used to identify the types of object.

When asked how else the system might support the identification of common object types, a number of different approaches were suggested. One participant wanted to define groups based on object proximity, another by tapping all of them, another by drawing a circle encompassing them all, another by physically picking all the objects up, cupping them in their hands and gently shaking them up and down.

4.2.3.2.3 Deleting Objects

To remove objects from system, participants all essentially created 'recycle bin regions' on the table, where objects could be placed, and if not immediately removed from that region, would be removed from the scene. This was similar to the approach used by Ziola et al. (2010) in removing tangible tracked objects from a scene.

4.2.3.3 Identifying/Selecting Objects

To identify of select objects in the scene, users just pointed to or picked up objects to select them.

4.2.3.4 Logic

To define the logic for FDG, four of the participants created regions on the table to define objects that were 'allowed together' versus those not. These were created by first drawing a box and labelling it as "allow/not allow", "yes/no". One participant gestured this by placing their hands together before then moving them apart (similar to "opening a window") to define each region. The other participant moved objects together and wrote either a tick/cross alongside to indicate the validity. One interesting approach used by the final participant (without a computer, or science background) was to write equations for

interactions, e.g. ‘Fox + Duck = X’, with X implying the interaction was not allowed. Whilst they were happy defining interactions for FDG, when asked about how they would use this for larger numbers of objects, they said it would not scale, instead using a similar approach to the majority by defining regions.

For defining the different rules of the game, one (non-computer science) participant wanted a log displayed on the table, showing what the system was thinking was happening (e.g. showing a message about an illegal move being detected), or as phrased by the participant, so they could see “what worked and what didn’t”.

4.2.3.4.1 Constraints

To define the boat’s path, five participants drew a line between the banks, moving the boat along the path to reinforce the constraint. The other participant simply moved the boat between banks without any line.

To limit the number of objects allowed on the boat at any point, there was a mix between writing the maximum number allowed next to it, and placing different combinations of objects in the allow/not allowed regions with the boat. When asked about how they could avoid having to enter every permutation of objects on the boat, the previously discussed (Section 4.1) concept of defining objects as having common aspects reappeared. However prior to this, one, non-technical participant actually noted that when defining such an interaction, the system should have a ‘generalise’ button so avoid repetitive entering of rules.

4.2.3.5 Navigation

A number of different methods were suggested in order to support system navigation. Given the continuous nature of tangible interaction, discrete events are required to confirm, cancel and navigate the system. Five participants wanted to use a physical button or region on the table that could be pressed to change between interacting with the system normally versus authoring new content. One participant wanted to use the ‘thumbs up’ gesture, which was what was actually used by the others as the ‘confirm’ action. In defining a confirm action, one participant noted that the system should allow them to swipe their arm across the desk to do a “reset” (as if they were pushing everything off the desk), saying the action would function “kind of like an etch-a-sketch” toy.

The most novel approach by one participant was to use a six-sided die as a ‘control cube’, with different functions on each side, changing the mode of the system based on the up-facing side.

4.2.3.6 Process Order

In evaluating the order of tasks for creating such a system, all participants said the system should allow them to define the objects being used, then the rules between them, and then finally play the game. Participants were asked if an order of introducing required objects, defining associated properties, and then defining the interactions used in the system was appropriate, all participants responded positively.

Prior to defining any objects, two participants wanted to first define the 'regions' they used to define valid/invalid logic, delete objects, etc., noting it is "like getting your tools ready", and that they "want to define areas for things I do regularly, like putting them on your desktop … toolbox on the side, canvas in main area". Despite wanting to customise their space, it was suggested that the system should have a default layout that can be later modified, instead of always setting everything up from scratch.

4.2.3.7 System Feedback

In exploring how the system should provide feedback regarding valid/invalid moves, there was a mix between those that wanted feedback received in a global context, versus those wanting feedback local to the objects involved. Two participants said it should be in the global context, as it should be akin to "tilting a pinball machine", where the system locks up as a form of response. Another wanted a large 'X' in the middle of the display, as they were worried that if feedback were localised to the objects involved, they may not see the feedback if they were focused on another region. A different participant wanted the display area to flash, with a sound indicating a bad system state, conflicting another participant who wanted the individual objects to flash, showing red on objects to indicate an error state.

The use of sound was supported by two other participants, wanting sound to play both when correct and incorrect actions are performed, playing a 'buzz' when things go wrong. Participants noted, "it's good to know you're doing the right thing, but it's more important to know when you're doing the wrong thing", with another suggesting the use of active vibration, as feedback should "use all senses".

4.2.4 Logic Study Summary

As was to be expected, participants each created their own uniquely individual systems, using their own approach. However, common themes did exist across systems. Participants separated both the psychological and programmatic aspect of creating interactions/logic in the system into two different groups; those classed as being 'good'

(legal) and those being ‘bad’ (illegal). Participants would create interaction rules for leaving the fox and duck alone on the river bank as being ‘bad’, asking the system to highlight the objects to convey an error state. However for valid interactions, they wanted a different operation, even though the only difference might be that the objects are highlighted in green (e.g. when the FDG all reach the far bank). Unsurprisingly, there was no separation as to the *interaction* versus the *output*. However, surprisingly it was a non-computer science participant was the one to suggest the system should have a generalise button to avoid entering every permutation possible for interactions.

For programming interactions, participants created regions for ‘accepted’ and ‘unaccepted’ interactions, moving objects into regions assigned to each. The other approach was to assign valid or invalid states directly next to where objects were located in the scene. Suggestions for feedback for valid and invalid interactions was mixed across participants between wanting feedback in the global scope and those wanting it to be localised to the objects involved. Limitations for quantitative constraints (i.e. only one object allowed on the boat), were defined by the user first performing the interaction involved, i.e. placing an object on the boat, before writing the number of legal items allowed near the boat.

To identify objects, names, abbreviations, colours and icons where used. However this use of colour was later overloaded to define types of objects with common properties. These common properties should, ideally be suggested by the system based on the physical properties of the objects, or else manually created by the user using a variety of techniques: cupping all the objects, drawing around them to encompass them, or placing them in close proximity.

In switching between interacting with the system versus editing it, a mix of gesture (‘thumbs up’) and a virtual region/button was used. One especially important result from the study was the resulting control flow resulting from the study was to introduce objects (based on common properties), define their individual properties and then define the interactions involving those objects. Given the goal of allowing the user to naturally describe the given scenario as if describing to another person, the system needs to support the description of the scenario in the order and format expected by the user.

4.3 Summary

This chapter has served to provide insight into how users would, ideally, interact with a tangible ad-hoc system, creating user controls using the affordances of available objects

as well as defining logic-based interactions between them. Subsequent chapters outline the design, development and implementation the TAM theoretical architecture to support the development of such systems.

Chapter 5. Controlling Existing Functionality Implementation

As previously discussed, the requirement for users to be able to leverage external application functionality using tangible ad-hoc user interface controls is required. This chapter explores an implementation of the TAM architecture that allows users to create novel tangible and touch controls for a set of known external application functions that have unknown controls, focusing specifically on short-term use (minutes to hours), enabling the user to be able to tell the system *what to do, not how to do it*.

The remainder of this chapter is as follows. First, the background for the tangible user controls is described, outlining the current direction and the types of interactions and functionality such systems should support. The implemented system is then described, providing a larger context in which the TAM architecture can be interpreted. A number of example applications built to demonstrate application functionality are then discussed. The chapter concludes with a discussion regarding how the system design and implementation relates to previous work, before a summary.

5.1 Background

Whilst previous work in adaptable controls has looked at the modification of UIs from some existing base configuration (e.g. Stuerzlinger et al. (2006)), this is work looks at the complete creation of a tangible UI from scratch based on the user's current context and available resources. Whilst modern systems have a near endless set of functions to accommodate for their generic development and usage, most people only use a small subset of those functions at any time (Stuerzlinger et al., 2006). As such, it makes sense to allow the user to develop tangible interfaces catering to that subset, leveraging the affordances of the available resources.

A number of different examples scenarios exist to demonstrate such functionality:

- Using a camera/projector pair (as is now common in lecture theatres supporting recording), there is no reason why a lecturer should not be able to walk into a room, and create controls to navigate their presentation based on which direction they point with their marker (left or right), perhaps even pointing to the projector to start/end the presentation. The user knows what functions the system is capable of, but only needs to define a UI that matches the current task requirements and context, allowing them to be mobile whilst giving the presentation.

- Geologists meeting to explore a recent field survey in a meeting room begin exploring depth scans of regions of interest. Despite the use of 3D systems to represent the data, participants have only brought their laptops to the meeting, creating difficulties for navigating the dataset in 3D. Using a whiteboard marker available in the room, a geologist makes an improvised joystick, mapping the left/right, up/down orientation of the joystick to the existing, keyboard-based navigation function. Using such a constructive control, the geologists can now easily explore the dataset using a more natural control with affordances appropriate for the current task.
- Designers of an industrial control system are working through the design and layout of the control room. Using a working software simulator of the system (e.g. a nuclear reactor), the team prototypes different user controls using improvised controls made from passive materials (some prototyped with 3D printing), as well as active electronic controls. With these controls, they run through different layout configurations of individual controls and whole control panels, and they can associate the controls with the simulator's functionality. Using this approach, they can evaluate the selection and location of controls, and actually use those controls to work through different use case scenarios with the simulator. For example does this current configuration support everyday use as well supporting optimal access for emergency and auditing requirements? No code has to be written, with designers creating and evaluating controls without leaving their current context.

This chapter explores such concepts, culminating in an implementation of TAM that allows users to create interactive, physical and touch-based UIs constructed from passive materials available within the current context, whilst incorporating active electronic input devices. Whilst such controls may allow for the generation of novel peripheral TUIs, entire systems can be controlled using such an approach. Each of the examples provided above are examples of actual systems that can be supported by the resulting implemented system.

5.2 Using the System

Whilst the primary contribution of this dissertation is the design of a system to support tangible ad-hoc interaction, it is easier to understand the features and design decisions when viewed in context with the end application. As such, this section describes the implementation of that architecture in an example system before the core architecture itself is then described in the following section.

5.2.1 Configuration

SAR-TAM is an implementation of the TAM architecture, providing an example system that enables the run-time creation of novel tangible user controls. SAR-TAM is written in C++, running under Ubuntu 12.04, and is built on top of an existing SAR framework developed and used within the Wearable Computer Lab at the University of South Australia (Broecker, 2013, Marner, 2013). The SAR framework provides data structures and functions for simple 3D manipulation and scalable projection mapping.

A table-top mounted projector is used to augment the user's workspace (Figure 43). A Microsoft Kinect is used along with a Natural Point OptiTrack[7] motion/object tracking system. The Kinect is collocated with the projector above the workspace, facing downwards on the table to detect touches and objects on any visible surface. The Kinect provides a 640x480 pixel depth image at 30Hz. Updates for the Kinect are threaded, meaning the system does not have to pause to wait for updates from the Kinect, instead only grabbing new frames as they are available.

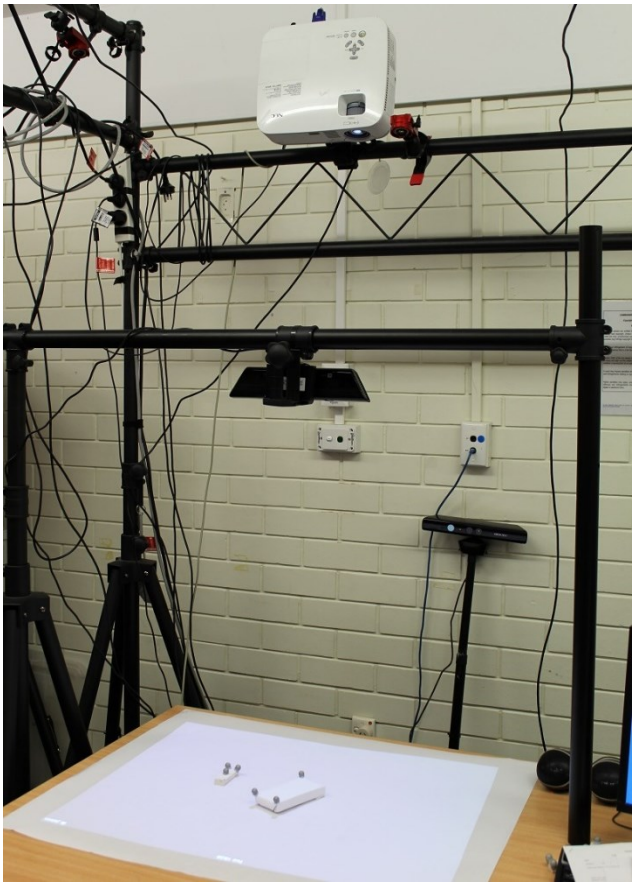

Figure 43 Basic tabletop configuration showing the projector, one OptiTrack camera (red) and downward facing Kinect

[7] http://www.naturalpoint.com/optitrack/

The OptiTrack system, running on a separate Windows-based PC, is used to identify and track objects used in the system in 6DOF between frames. Tracking objects requires the addition of (at least) three, retro-reflective markers be attached to each object (Figure 44). The tracking system uses the VRPN protocol[8] to provide tracking data across the network at approximately 60FPS. Originally black and white fiducial markers were thought to be suitable for tracking objects, however issues with finding flat surfaces on objects as well as always ensuring the marker was visible following obfuscation when the object was handled. However, optical markers can be placed arbitrarily on organic objects, with any number of additional markers (beyond the minimum of three) available to ensure more comprehensive tracking of the object, especially when being handled by the user.

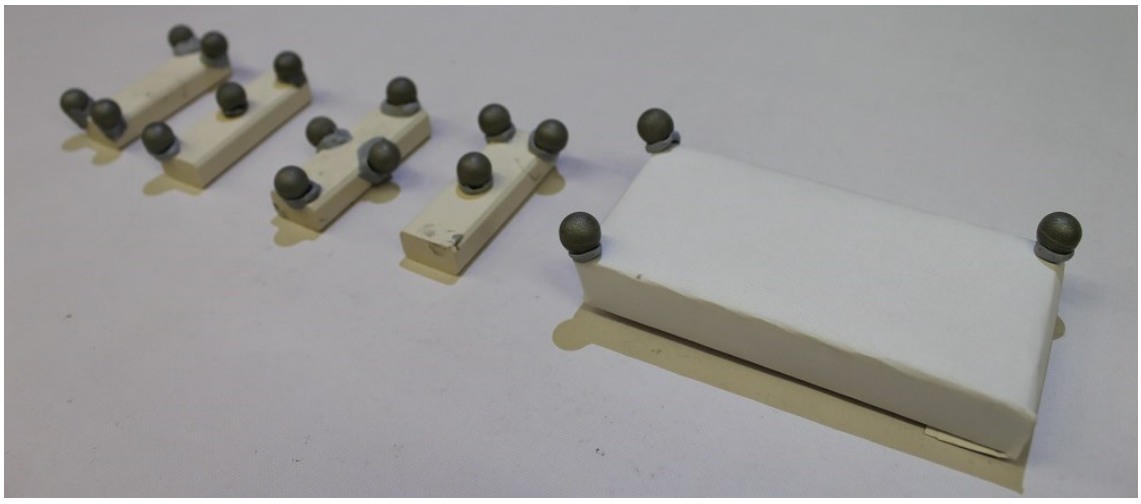

Figure 44 Generic objects with attached OptiTrack markers

In the future with advances in systems such as Kinect Fusion (Newcombe et al., 2011) and other optical tracking systems not requiring depth cameras, such as Pradeep et al. (2013), it is envisioned the requirement for the OptiTrack system could be entirely replaced by computer vision algorithms, providing the same level of tracking as currently afforded. In the system, user controls based on touch are created using only the Kinect sensor, with tangible controls using the OptiTrack to provide information about the position/orientation of the object and Kinect to provide any required size information.

The SAR-TAM system runs at approximately 55FPS (frames per second) on a 2006 Intel Core 2 6300 clocked at 1.86GHz. Given the minimal rendering of graphics content, the primary limitation for frame rate is the CPU. Whilst updates from the Kinect and OptiTrack do no limit the speed of the SAR-TAM system itself, their use as an interaction

[8] http://www.cs.unc.edu/Research/vrpn/

device means interactions are limited to their refresh rate. However the current rate of 55FPS is well above the 10FPS required for a system to feel interactive (Card et al., 1983).

Both the Kinect and OptiTrack systems must be calibrated so that each device is using the same coordinate space as the SAR system. Calibration involves calculating a simple coordinate space transform based on known points collected in both SAR and tracking system space, i.e. selecting the SAR origin, then defining known points on each axis (four points total – three axes plus origin). This creates a transformation matrix that is applied to subsequent frames to convert from the individual coordinate system to the SAR system's coordinate space.

A Griffin PowerMate[9] button (Figure 45) employs a push-button, rotation sensor and blue LED, and is used as the 'bootstrap' and confirmation action for the system, however the rotation sensor is not used. Such a configuration has previously been used by Edge (2008) in peripheral TUIs. To provide feedback to the user regarding the contextual applicability of the button, the button's LED glows at 1Hz when the button can be used. Whilst other modes for 'bootstrapping' the system exist (such as that used in WorldKit where a user 'paints' a surface with their hand to define the interaction area), that despite being mobile, could interfere with normal interactions as false positives. Consistent with user feedback from the two pilot studies, it was noted that the bootstrap function should utilise input of a different modality to that used for system interactions, and as such the button was a suitable control.

Figure 45 The Griffin PowerMate button, with LED visible as the subtle blue glow around the base

[9] http://griffintechnology.com/support/powermate/

5.2.2 Creating Controls

The implemented system allows users to create novel, touch and tangible controls ad-hoc as required to control existing application functions. To create a UI control mapped to external application functionality, the user follows a simple state machine (Figure 46), through four separate states/steps:

1. Select Function Group: Select a group of functions containing a desired function to control.
2. Select Function to Control: Select the individual external function to control.
3. Select Control Type to Create: Select the specific input control to use to control the selected external function.
4. Create Input Control: Create the selected input control using guidance from the system.

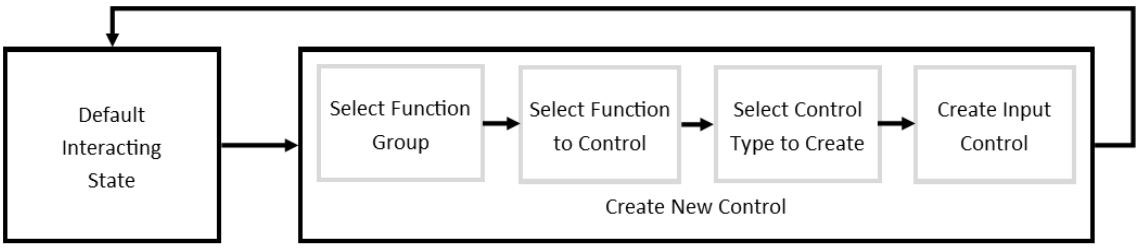

Figure 46 State machine for creating a new input control

To do perform this sequence, the user first presses the large, physical PowerMate button. The system then presents any number of different function 'groups' as projected buttons on the surface (Figure 47a). Each button is labelled according the types of functions the group contains, e.g. "Audio Control", "Cooling System", etc. The user selects which group of functions to control by touching one (Figure 47b), with the system then showing the list of individual functions contained in that group, e.g. "Volume", "Balance", "Cross Fading", etc. The user then selects the specific function to control. Using this approach allows functions across multiple external systems to appear in arbitrary groups by related function (or external system should the user desire). By grouping functions by their purpose, heterogeneous environments can appear to the user as all being part of the one ad-hoc system, since functions from different systems appear side by side.

With the knowledge of what that the desired function to be controlled is, the system has information about the parameters that function requires. Using this, the system then presents the different fundamental input interactions that can be used to control the selected function (Figure 47c), e.g. projected button, slider, dial, use of an object in a given orientation/position, etc. as well as any VirtualAction instances available.

Depending on what control the user selects, they are then stepped through its creation using text and voice prompts (Figure 47d), either using touch to define projection-based controls on an available surface or the selection and configuration of an object(s) for tangible controls. The whole process of creating UI tangible controls for existing application functions takes only seconds, with a maximum menu depth of three (group selection, function selection, input control selection). Once authored on the table, controls can be moved anywhere in the tracking volume as required, with large scale, full room OptiTrack systems available[10].

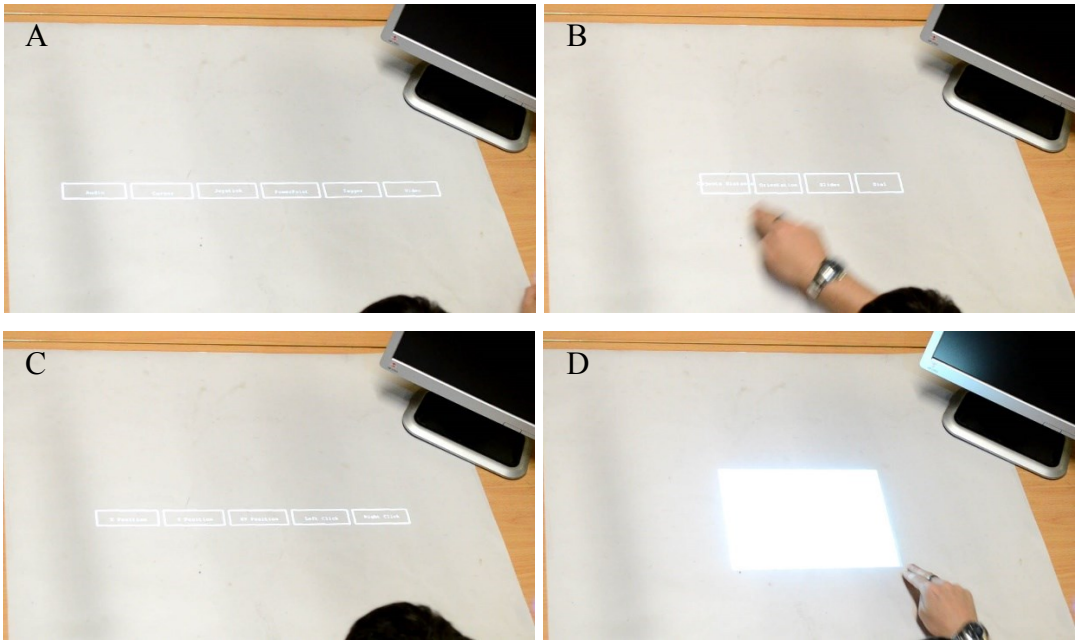

Figure 47 Selecting function-group (A), selecting a function (B), selecting a control to use (C), and then defining that control (D)

For example, for the user to create a simple volume control dial, they first press the button. The system then displays the different function groups, as defined in an XML file (described later). Upon selecting the "Audio Control" group, the system displays the individual functions that were defined in that group/node in the XML file. One such entry is "Volume Control", with the associated XML node defining that the function requires a parameter between 0 and 100. Since the system knows the function requires a parameter between 0 and 100, only input Actions capable of providing a matching single integer parameter are displayed. In this case, options for a touch dial, touch slider, object between positions, and object between orientations are displayed. The user selects the dial control, and is then prompted to draw the control. For a dial, this involves first touching the location for the centre of the dial, dragging outwards at least 20mm to define the 'starting

[10] http://www.naturalpoint.com/optitrack/products/prime-41/

angle' (Figure 48, left), before then 'sweeping' around to define a partial or whole dial (Figure 48, right). The dial 'snaps' to becoming the full 360° when it is greater than 350°. When the user removes their finger, the dial remains on the table, but can be re-drawn as many times as required until it is satisfactory, at which time the user presses the PowerMate button to confirm its creation. The system then resets to the default interaction state, with the dial's value now mapped to the volume control function.

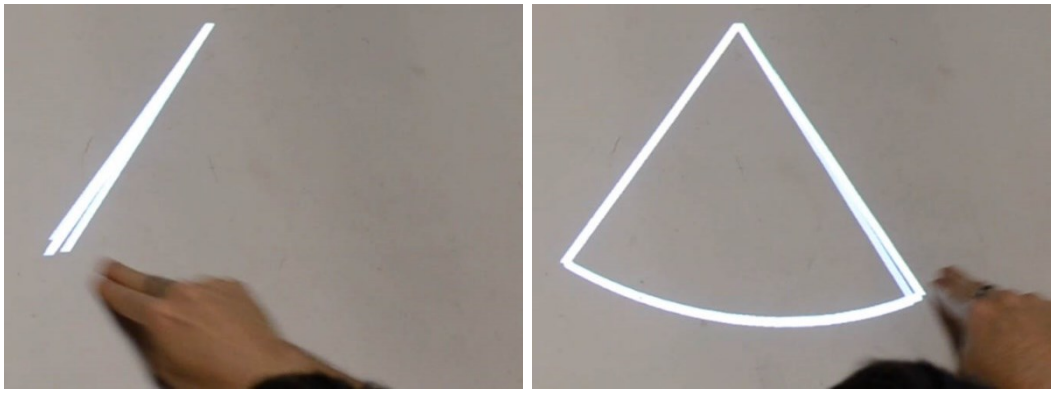

Figure 48 Creating a volume dial control by dragging 'out' to create the dial start angle, and completing the arc

5.2.3 Editing Controls

To edit/delete a control (Figure 49), the user holds the button for at least one second. Upon releasing the button, a red, semi-transparent bar appears across the bottom of the interaction area, with any projected controls wobbling as if floating on water (Figure 49, left). This is similar to the effect used when repositioning icons on mobile phones. Users can then touch-drag-release controls to reposition them, or drag them into the red bar (at which time it becomes opaque) to delete the controls (Figure 49, right). This was designed to be akin to "flicking something off the bench".

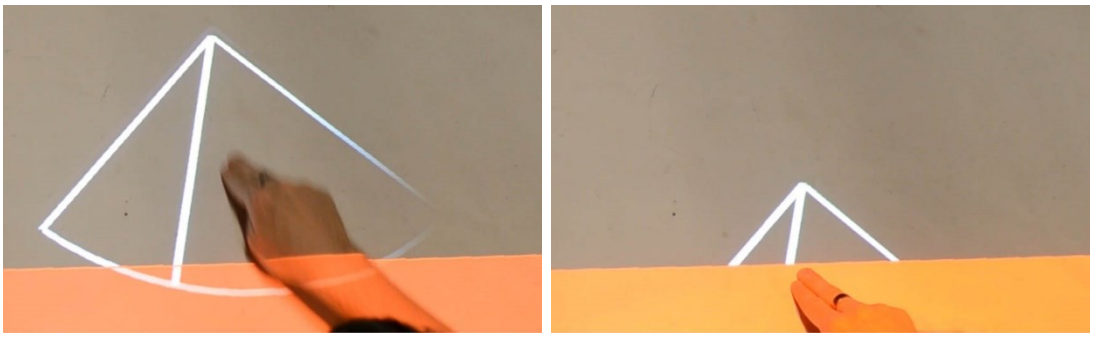

Figure 49 Repositioning floating controls (left) and deleting a control by dragging it off the edge (right)

As per feedback from the studies, the user's arms were also tracked to monitor their engagement in the current task. If the user accidentally makes a wrong selection or wants to exit out of the current state, they can just remove their arms from the table, at which

point an analogue-style stop-watch clock appears in the upper-right (Figure 50), counting down four seconds before then exiting back to the default interaction state. If at any point the user re-engages with the table by placing their arms anywhere in it, the countdown is aborted. This is similar to the approach used by video games utilising the Kinect.

Figure 50 Idle count down timer before resetting to the default state

To identify the user's arms, the system scans for contours sharing the edge of the Kinect's depth camera frame, checking for a minimum contour size, before then making the assumption it is an arm. Pilot studies with various members of the lab showed this to be quite robust, and allowed the system to cancel out of the current task if the user is no longer interacting (i.e. arms off the table).

5.3 Supported Interactions and Controls

Aside from the types of interactions identified in the second pilot study, a number of previous works, e.g. Conner et al. (1992) and more recently Lee et al. (2004a), Grossman and Wigdor (2007) and Marquardt et al. (2011), have explored the classification and interaction space involving proxemic interactions relating to their use in HCI, with the ultimate goal to provide developers with toolkit support for the 3D creation of widgets. Most relevant are the types of controls identified by Henderson and Feiner (2010) in their ad-hoc interaction study and Marquardt et al. (2011) in the development of the Proximity Toolkit. The results of the study by Henderson and Feiner showed the nature with which objects in the scene were used for various tasks 3D, 2D and valuator tasks (Table 3) as they related to machine maintenance and home entertainment control, with the results of the study supporting the types of controls developed in the second pilot study.

Task Type	Button	Valuator	Movable
Maintenance Interaction Domain			
3D Object Selection	13%	60%	27%
3D Object Manipulation	7%	87%	20%
3D Scene Control	13%	73%	27%
2D Document Visualisations	7%	67%	27%
Discrete Valuators	67%	20%	13%
Continuous Valuators	33%	40%	33%
Menu Selection	47%	53%	13%
All Tasks	27%	57%	23%
Home Entertainment Domain			
3D Object Selection	47%	40%	20%
3D Object Manipulation	20%	67%	13%
3D Scene Control	20%	53%	33%
2D Document Visualisations	20%	67%	13%
Discrete Valuators	67%	33%	7%
Continuous Valuators	47%	53%	13%
Menu Selection	53%	33%	0%
All Tasks	**39%**	**50%**	**14%**

Table 3 Frequency of control occurrence by task, reproduced from Henderson and Feiner (2010)

In creating the Proximity Toolkit, Marquardt et al. (2011) also identified a more comprehensive collection of different proxemic relationships properties, both within and between objects (as previously shown in Figure 22).

As can be seen from these previous works on classifying tangible input and proxemic interactions, there are a large number of different ways to interpret physical user input. However, there are a number of basic inputs common throughout, such as the use of object location (both position and orientation), binary inputs (such as object visibility) and the use of objects as valuators. As such, by evaluating the types of possible inputs, a subset were implemented to demonstrate a comprehensive set of UI controls that enable the user to both control existing application functions whilst also being able to create new input devices in their own right. The implementation employs the position, orientation, visibility and proximity of objects to allow physical interactions to generate discrete events. In addition to using touch-based buttons, sliders and dials, the system can support

the vast majority of interactions as previously suggested in both the pilot study, as well as in the possible interactions identified by other work. However, it is important to emphasise that these interactions are merely what has been implemented as a demonstration. Whilst their utilisation is discussed later in this section, the types of implemented Action classes are:

Nullary:

- PositionAction: monitors if given objects are in a defined 3D position. For example, the user selects a given object by placing it in the middle of the interaction area, presses the button to select it, before then placing it in the desired trigger location and pressing the button again. Any time the object is in that location, the Action will trigger.
- RotationAction: monitors if an object is in a given 3D orientation. To do this, the user selects a given object by placing it in the middle of the interaction area, presses the button to select it, before then placing it in the desired trigger orientation and pressing the button again. Any time the object is in that orientation, the Action triggers.
- ProximityAction: monitors if at least two objects are a given distance apart. The user selects the objects by placing them in the middle of the interaction area, presses the button to select them, before then placing them the desired distance apart before pressing the button again. Any time the objects are within that distance (±10%), the Action trigger occurs.
- ButtonAction: a projected button triggered by touch or objects. To achieve this, the user touches the interaction area (defining one corner of the button) and drags to define the opposite corner. This process can be repeated until the desired button region is achieved, before the user then presses the button. A rectangle labelled with the associated function name then appears and is triggered by any touch or placement of physical object in that region.
- VisibleAction: triggers if a given object is visible to the systems tracking systems. For example, the user selects a given object by placing it in the middle of the interaction area, then presses the button to select it. Any time that object is visible, the Action will trigger.
- PointsAtAction: triggers if a given object is ‘pointing at’ another object by using a 3D vector defined using the object’s coordinate space. For example, the user selects the pointing object by placing it in the middle of the interaction area and

presses the button to select it. The user then places the object to be pointed at in the interaction area, with the first object orientated to 'point' at it (defining a 3D vector in object space), before then pressing the button again. Any time the first object is orientated so that the vector is pointing at the second object based on that initial orientation, the Action will trigger.

- AlwaysTrueAction: continually triggers the Action, regardless of user actions, primarily used for debugging or if the user wants to continually execute a given external function.

Valuator:

- GraduatedProximityAction: generates a value based on the distance between two objects OR based on a projected slider control. For example, the user selects the objects by placing them in the middle of the interaction area and presses the button to select them. The distance between the objects (in millimetres) is then passed as the resulting value of the Action, triggering each frame. Instead of using objects, the user may also touch and drag to define a slider control (as a line or set of line segments) across any 3D surface before then pressing the button. Any touch or object on the slider control will then define a value between 0-1000 for the Action.
- RotationDifferenceAction: generates a value based on the rotation of an object around a user-defined 3D axis OR based on a projected dial control. For example, the user selects a given object by placing it in the middle of the interaction area, then presses the button to select it. The system then prompts for the user to rotate it between a minimum and maximum rotation before then pressing the button again. The angle of the object as a value between 0-1000 (between the minimum and maximum rotations) is then generated by the Action for each subsequent frame. Instead of using objects, the user may also touch and drag to define a dial control across any surface before then pressing the button. The user touches a point to define the centre of the dial, before dragging outwards to define the zero angle for the dial (once the dial diameter is greater than 30mm, that angle locks and cannot be changed, at which time the user 'sweeps' around to define the rest of the dial. Partial dials are possible, with the control snapping to a full 360^{o} when the user's finger nears the starting angle. The user then press the button to confirm the control. Any touch or object on the dial generates a value for that frame between 0-1000.

- RegionAction: generates two or three values based on the location of a touch (2D = two X, Y values), or object (3D = three X, Y, Z values). For example, the user defines a 2D region (like for the button control), except the Action generates two values (0-1000) for any touch location within that region, and three values (0-1000) for the axis on the table and the height of the object from the region's surface for the third.

For Actions that result in some value, an IntegerProperty is stored in the Value field of the Action instance, storing a single integer value for the resulting interaction. Whilst any integer value can be stored, in this implementation for range-based interactions (slider, dials, etc.) the value is explicitly defined as being between 0-1000. Whilst this range is fixed, the range of values actually used as output for external functions can be transformed as part of their mapping, as previously highlighted (Section 3.4.4) and discussed in detail later in this chapter.

Depending on the nature of the interaction being monitored, more than one value may result. For example, RegionAction acts as a virtual touchpad that provides provide two (or three) different values (however both in the pre-determined 0-1000 range) as values for the X and Y (and possibly Z) location of a touch (or object) in a given area, with the values encapsulated in a root PositionProperty assigned to the Action's Value attribute.

5.3.1 GUI Control Substitution

As the Action class is the component responsible for monitoring system inputs, Action is also the only component in the architecture that knows what type of interaction has occurred and the context within which the interaction occurred and thus is responsible for creating any required representation of that input. In this implementation this is using OpenGL to display context using the projector, however in other implementations this might be to turn on LEDs, or update a mobile phone display, etc. This display function is not to create the output of the interaction function, rather just to provide state feedback for the interaction, e.g. rendering a flat button, showing as indented when the Action is occurring (i.e. the button is being pressed).

The different button, slider, dial and touch pad controls created in the described implementation use the object and touch locations in the scene, evaluating if any of them are within the user-defined region for the control. If so, the associated Actions update their internal state, triggering the interaction in the process, and render controls with those values using OpenGL. Such controls were previously shown (Figure 31).

Whilst not designed to simply replace existing GUIs with tangible equivalents, the system does support such direct equivalents, e.g. replacing an on-screen slider with a touch-based one (assuming a function to set the same value is exposed to the system). Such functionality allows the creation of TUIs that utilise the equivalencies that have been identified between the GUI controls and TUI controls (Ullmer and Ishii, 1997), as well as those controls existing more a limited system in peripheral TUIs (Edge, 2008). As a result, the system allows traditional GUI applications to become more ubiquitous and appear alongside traditional input controls such as the mouse and keyboard as required by the current task, e.g. timeline and navigation controls created across the top of the user's keyboard (Figure 51), reducing the amount time the user spends time-multiplexing in the software.

Figure 51 Traditional GUI elements existing alongside traditional inputs as dedicated controls

Given the possible mappings between the input controls and existing GUI controls, it becomes possible to map which GUI controls could be created using different types of input Actions. The top half of Table 4 shows the different types of interactions possible, and how those inputs could be used to generate inputs for traditional GUI controls. The different Action classes in the system as represented in abstract by the column headers (e.g. *Object in Orientation)*. The types of inputs represented in the column headers are classified as being Boolean (two states of input) or continuous (generating a resulting value from the interaction). Correspondingly, the bottom half of Table 4 shows how such interactions could be used to generate dedicated input devices (such as a joystick) from scratch (discussed in the next section). Whilst with enough encoding of the input into an

appropriate form any input could be used to simulate any control, a number of more sensible options are possible (indicated with checkboxes).

		Physical						Touch
		Position				Orientation		
		Object Between Positions	Object in Position	Object Visible	Object Proximity	Object in Orientation	Object Between Orientations	Button, Slider and Dial
		Continuous	*Boolean*	*Boolean*	*Boolean and Continuous*	*Boolean*	*Continuous*	*Boolean and Continuous*
Controls	**Button**		☑	☑	☑	☑		☑
	Radio Button	☑	☑			☑	☑	☑
	Slider	☑			☑		☑	☑
	List Box	☑	☑			☑	☑	☑
	Spinner	☑	☑	☑	☑	☑	☑	☑
	Menu	☑	☑	☑		☑	☑	☑
	Tab	☑	☑	☑		☑	☑	☑
Input Device	**Mouse**	☑					☑	☑
	Keyboard		☑			☑		☑
	Joystick	☑					☑	☑
	Steering Wheel						☑	☑

Table 4 Mapping existing controls GUI/physical against how they can be controlled in the system (☑ Supported)

To define tangible controls, the ad-hoc system prompts the user to select the objects by placing them in front of the user in the interaction area and pressing the PowerMate button. If more information is needed about how to interpret the object's movements, the system prompts for the object to be moved through the required states. For example, using an *Object Between Orientations* would require the user to orient the object to the orientation representing the 'minimum' value before pressing the PowerMate button, before being prompted to orient it for the 'maximum value', pressing the button again. An elaborated version of some of the major controls that can be emulated/created follows.

5.3.1.1 Button

Buttons can utilise Boolean inputs to generate input events, supporting *Object in Position*, *Object Visible*, *Object Proximity*, *Object in Orientation* and *touch button*. For example, to generate button input the user can use any object and monitor it so that when it is in a set position (*Object in Position*) (Figure 52), the associated 'button' is pushed. For toggled buttons, the persistence of the object in that location provides the 'toggled' state. Equally,

an object in a given orientation (*Object in Orientation*) or the visibility of an object (*Object Visible*) could be used. A projected button could also obviously be used (Figure 53). Using two objects, the proximity of those objects at a given distance (*Object Proximity*) could be used as a trigger for a nullary event, 'pushing' the button.

Figure 52 When an object is in a given position (denoted by the target), a nullary event is generated to simulate button input (digital overlay added for illustration)

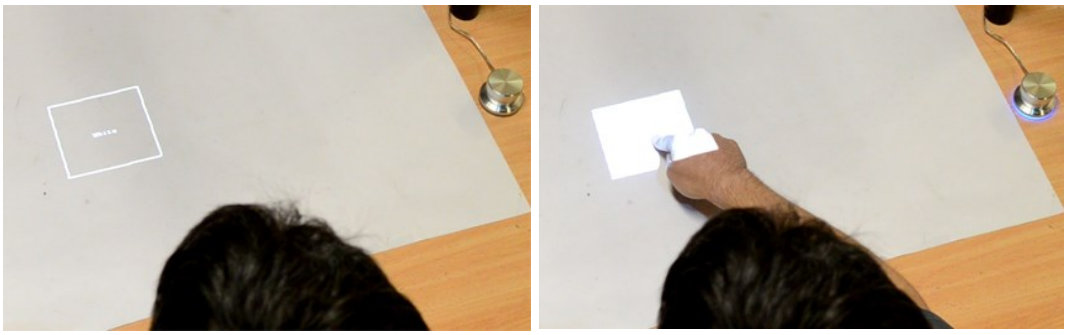

Figure 53 Simple button control when idle (left) and selected (right)

5.3.1.2 Slider

Given the requirement for sliders to generate a value from the interaction, sliders can utilise any continuous controls. This allows sliders to be created using *Object Between Positions*, *Object Proximity*, *Object Between Orientations*, and using a *touch slider* control. To create these, the user can employ objects to define the start and end positions of a slider control, using either touch or a third object to set the value depending on the distance between the objects (*Object Between Positions*). *Object Between Orientations* can also be used using the same approach, defining a maximum and minimum orientation for the object (Figure 54). *Object Proximity* can also be used to provide a continuous value for objects up to 1000mm apart (the maximum value of range-based values in the implementation). For a slider defined by touch (Figure 55), the projected path may be linear, a high order curved path or simple arc, supporting creation on both flat and 3D surfaces.

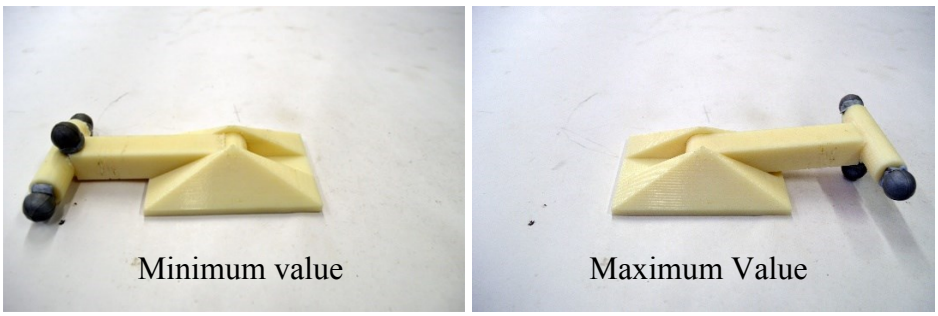

Figure 54 Physical lever between orientations to simulate a slider valuator (text added for illustration)

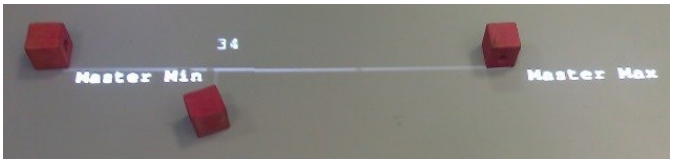

Figure 55 Objects used as a markers for a volume slider control

5.3.1.3 Spinner

A spinner can be used to generate any number of discrete values, depending on the mapping of the function. As such, both Boolean and continuous controls can be used depending on the mapping (i.e. all input controls). For a mapping that requires a parameter to function, the proximity of two objects (Figure 56) can be used to select a value that increases as the objects move further away (*Object Proximity*). Conversely, should the resulting value be persistently stored in the external application, any nullary control (such as those previously described for the button control) could be used to provide simple increase/decrease functionality.

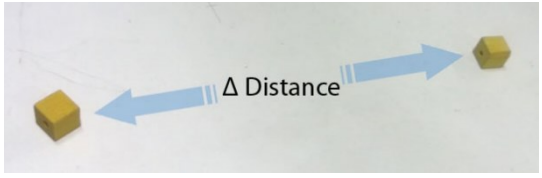

Figure 56 Using physical proximity as a valuator (digital overlay added for illustration)

5.3.1.4 Radio Button, List Box, Menu and Tabs

Despite there appearing to be a large range of distinct controls, many of the controls are similar in fundamental function, differing only in their presentation to the user and contextual usage. Radio buttons, list boxes, menus and even tab controls are all fundamentally performing the same task, allowing the user to select an option from a limited known set of options. Where the controls differ is their application and presentation to the user. For example radio buttons would only be used for a small number of choices when compared to a list box which might contain more choices. A tab control only provides a limited number of options, but the selection of those options needs to be

alongside the content to which they relate. Following previous work in presenting tangible equivalents to such GUI controls (as previous discussed in Chapter 2), a reduced set of TUI controls enable the emulation of the full set of fundamentally similar GUI controls. Given the function of selecting an option from a known set, Boolean controls can be used (with an individual control existing for each option), or continuous controls can be used (to generate a single value representing an index of a single valid option).

For example, given a set of options, the user can utilise the orientation of a given object to select an option (either as *Object Between Orientations* or *Object in Orientation*). For each different orientation of the object, a different option is selected. Given the object can only have one orientation at any time, the physical affordance ensures exclusivity for option selection so only one option is ever selected (Figure 57). This same approach enables the creation of a tangible dial control. Had the user created the selection control using individual buttons for each mapping, multiple options could be selected at once, which may or not may not be suitable for the given function.

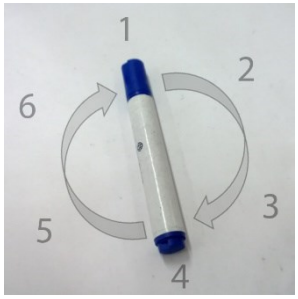

Figure 57 Object used as radio button (digital overlay added for illustration)

Similar to above, the position of an object across or between positions may also be used instead of orientation, i.e. *Object Between Positions* or *Object in Position* (depending on the use of a nullary or valuator-based mapping). *Object Between Positions* and *Object in Position* both allow the position of an object to be used to select individual values from that collection (Figure 58). Such an approach is the realisation of Ullmer and Ishii's equivalent of using objects in different positions of a tray as an equivalent to a menu.

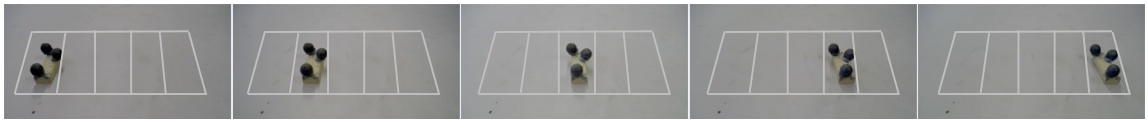

Figure 58 List box control based on an object in a given location (digital overlay added for illustration)

As just highlighted, a number of different controls can be realised depending on the associated mapping. For example, a radio button could be emulated as a marker pen in any of six different orientations on a table surface. This could be created in two different

ways depending on the mapping entry. If the mapping was for a single function that accepted a parameter between one and six, the use of an *Object Between Orientations* could be used to provide a valuator that generates the selected 0-6 value. This approach would require only one control to be created. Conversely, six different functions could appear in the mappings, each nullary and triggering a single selection. As a result, the user could create the same control, but use *Object in Orientation*, defining six different controls, each triggering a different function when the same object is in a different orientation. Both approaches generate the same end control for the user, but the method for exposing functionality in a mapping can influence how controls are created to control it. In addition to this, different input Actions can produce the same (or equivalent) user controls, i.e. as just previous described using *Object Between Orientations* and *Object in Orientation* to create a functionally identical control. However, if multiple individual mappings are used, the exclusive selection of a single one at any one time cannot be guaranteed given the user can create separate controls for each mappings, triggering multiple ones simultaneously.

5.3.2 Custom Input Devices

Aside from simply bringing GUI controls into the physical realm, dedicated physical controls can also be developed. Support for tracking objects allows users to utilise passive props as if they actually contained active electronics. For example, Figure 59 shows a prop steering wheel with added markers, that can actually be used to control a racing game, once an interaction is created that uses the orientation of an object (i.e. the steering wheel) between a far-left and a far-right rotations.

Figure 59 A prop steering wheel can rapidly be converted from passive prop to an active input device with the simple addition of optical tracking markers (top)

Whilst props can be constructed from everyday materials, e.g. just cutting a cardboard circle for a steering wheel, dedicated props can also be developed. The recent consumer adoption of 3D printing/additive manufacturing has created an entire realm of new applications for the technology, especially for those involved in design and prototyping. Using 3D printing, new physical controls can be designed or downloaded (from free 3D model marketplaces such as Thingiverse[11]), and printed on demand. Figure 60 shows a completely passive 3D printed lever available online[12], printed in about an hour, and immediately used as an active input device within the system simply by adding markers to the handle. Using the *Object between Orientations* Action, the user can define maximum and minimum orientations as the top/bottom of the lever's orientation.

Figure 60 A passive 3D printed lever that can act equivalent to one containing dedicated electronics

Users can now create effective input controls on demand, without requiring any electrical, prototyping or software development knowledge/skills, as many previous systems have required. Aside from prototyping small interactive systems, such techniques could be used in the design and development of large scale industrial machinery. As previously mentioned (Section 5.1), with a working simulator of the system, e.g. a nuclear reactor, designers can evaluate different control panel layouts, as well as the selection, sizing and feel of the controls themselves (Figure 61). Whilst previous work (Porter et al., 2010) has examined creating simple touch-controls in lieu of dedicated physical controls without linking them to any application functions, TAM supports the full range of touch and

[11] http://www.thingiverse.com/

[12] http://www.thingiverse.com/thing:278793

tangible devices, whilst also allowing them to actually be linked to application functions for interactivity.

Figure 61 Current prototyping of large scale industrial control systems with SAR, reprinted with permission from Michael Marner

Support for constructive assemblies/TACs as defined by Ullmer and Ishii (2000) are also supported. Users can create novel input devices, utilising the unique affordances of passive materials fulfilling the requirements for the current task. Figure 62 shows an improvised joystick made out of a mug, two rubber bands, a marker and some sticky tape. Improvised controls can be rapidly created as needed, with the example joystick constructed in under 60 seconds. All the user has to do is define two interactions, one defining the minimum (left) and maximum (right) orientations for the X axis and the minimum (down) and maximum (up) orientations for the Y axis (using *Object between Orientations*), and the system will interpolate between the extremes, generating the joystick's 2D value.

Figure 62 Improvised joystick showing the tracked marker (left) and constructive assembly (right)

Where the required controls are not available, novel input controls can be created on-the-fly to emulate more complicated, traditional controls that utilise active electronics. Whilst

TUIs are not ephemeral by nature, the ability for multiple objects to be used together to create short-term inputs, creates controls that are ephemeral in nature by their use together as a constructive assembly. In addition, the physical properties and constraints of objects can be leveraged to create TAC controls as identified by Ullmer and Ishii. In the case of the joystick, the size/rim of the mug creates a physical limit/guide for the purpose and function of the joystick control.

5.3.3 Communicating with External Systems

To enable the simple integration of external systems, VirtualProperties execute a given application, passing any associated parameters. This allows the individual functions of an external system to be exposed in one (or multiple) applications that can be executed using command line parameters. In the implementation of the VirtualProperty patch panel, external functions available for use are defined in an XML file. For example, to set a value for a joystick emulator would look like Figure 63.

```
<output execute="joystick.exe setX %i" ratelimit="0" scale="range" min="0"
                    max="1000" name="Joystick X" />
```

Figure 63 Example XML mapping of a single axis of a joystick

Following the results of the second pilot study, application functions can be grouped, primarily based on sets of related functions, e.g. audio control. Functions are grouped in an XML node "outputs" with a single attribute defining a user-friendly name with which the user can identify the group's functions. Given each entry in a group is its own function declaration, a single group can actually utilise functions from any number of different external systems. The use of a hierarchical menu to display the mappings to the user allows the system to leverage the limited display space on the interaction area whilst not overwhelming the user by showing all mappings at once.

External systems can also provide input to the system using VirtualActions. VirtualActions are defined in the same XML file as *input* nodes (as previously shown in Figure 32). VirtualActions appear alongside native system Actions. For example when the user selects to control a single axis for the the joystick mapping, any VirtualActions that provide a single parameter are displayed alongside the native inputs. All that would be required to incorporate new input modalities, e.g. voice or gesture, would be for the external application to define a nullary mapping in the XML file. Whenever that input is to be triggered, the external system calls an application, SendData.exe, that passes any

parameters received (the first of which being the identifier for the VirtualAction as defined in the XML) to the VirtualAction instance.

5.4 Applications

The bi-directional integration of external applications creates applications across a number of different areas. The creation of ephemeral UIs to control existing digital systems allows users to create entire UIs from scratch based on the requirements for the current task and context. For example, as previously mentioned, a user giving a lecture may walk and be able to create simple interactions to allow slide navigation by pointing their pen at different sides of the room, effectively replacing the need for a dedicated digital presenter. Similarly a video editor might create projected controls alongside their existing keyboard and mouse to create shortcuts for timeline navigation, clipping videos, etc. Such functionality removes the need for dedicated peripheral-only tangible systems such as Edge (2008).

In addition to context based tasks, the ad-hoc functionality can be utilised for prototyping. The previously highlighted (Sections 5.1 and Section 5.3.2) example of an industrial control simulator being controlled using custom controls allows designers without software development experience to develop any number of physical controls from passive materials, using them to actually interact with the simulator. Controls can be initially prototyped with crude materials, then refined using 3D printing, with the prints themselves then replacing their crude predecessor. Final versions of controls with active electronics can then also be made and integrated into TAM without any changes to the application being controlled. As such, entire functional mock-ups of control rooms can be created and used as working controls, allowing the evaluation of any number of different design decisions, all without the user having to author any code (assuming the simulator mappings have already been created). Feedback from a public demonstration of such controls can be seen in Appendix C – Public Demonstration.

5.4.1 Example Applications

A number of example application mappings were created to demonstrate the ability to create ad-hoc controls, as well as to show how such controls could leverage external systems.

5.4.1.1 Video Editor

Mappings for a simple video editor were created that, after initially selecting the file to edit, could be entirely controlled using ephemeral controls. The video is displayed on a

co-located display, with the system supporting the creation of controls timelines, play state and for cutting/joining the film. One of the most basic implementations would be a slider with at least two buttons for cutting/saving sections of the video (Figure 64). However, more elaborate tangible controls could be integrated. An old film guillotine could be used as *Object in Orientation/Position* to trigger cutting or joining the film whenever the user physically operates the prop.

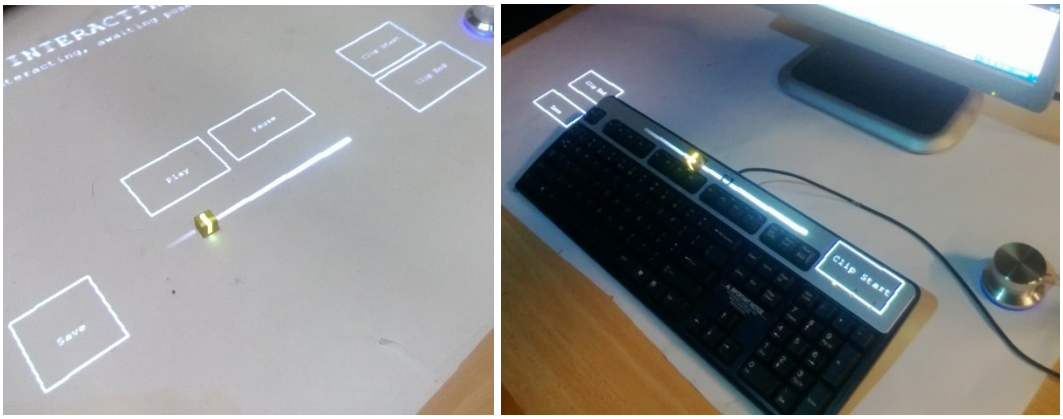

Figure 64 Ad-hoc video editing controls on their own (left) and supplementing the existing controls (right)

To create such a system, XML mappings must exist for each individual function to be controlled, in this case this means nullary functions for play, stop, pause, clip-region start, clip-region end, and save. The clip-region functions allow the user to define start and end points on the timeline to crop out that part of the video. In addition to the nullary functions, a timeline function must also be exposed that takes in a single value of a position on the timeline to navigate to.

To create the controller, the user first pushes the button to create a control, and selects the video editor function group, before then selecting the timeline function. Because the mapping for the timeline function requires a parameter to function, TAM only shows Actions that can generate the required value, one of which, GraduatedProximityAction, is presented to the user as a "Slider" option. The user selects this control and drags across the top of the user's keyboard with their finger to define the slider, before pressing the button to confirm. The user can now touch the slider control to navigate to that position on the timeline in the video on the external PC. This process is then repeated for the other functions, defining button controls for the nullary functions. Once all functions have been created, the user can navigate using the timeline across the top of their keyboard, before pressing a button alongside to define a clipping start-point, navigating to a later frame and then pressing the clipping end-point button. The user then presses save, and the video editor generates a new video file containing only the cropped section of the video clip.

5.4.1.2 Audio Mixer

The audio mixer allows the user to control media whilst adjusting individual audio channel settings, essentially creating an on-demand sound studio/DJ-style mixing board (Figure 65). Depending on what is being mixed at the time, controls can be added/removed as required for the current task. A DJ playing a whole set over a night might have different configurations for different parts of the set, as was suggested by one of the second pilot study participants who was a DJ. Controls are not limited to vertical sliders, horizontal sliders for balance, dials, levers, etc. can all be leveraged as they relate to different controls. Instead of having controls laid out in a horizontal fashion like traditional mixing boards, users could create radial layouts as to ensure all controls are within a standing-arms' reach. The use of an application providing MIDI mappings (with just the one of code line for each control) would allow the user to integrate ephemeral, tangible and ad-hoc controls with thousands of existing audio applications. This use of an ad-hoc audio controller was one example application identified by a participant in the pilot study.

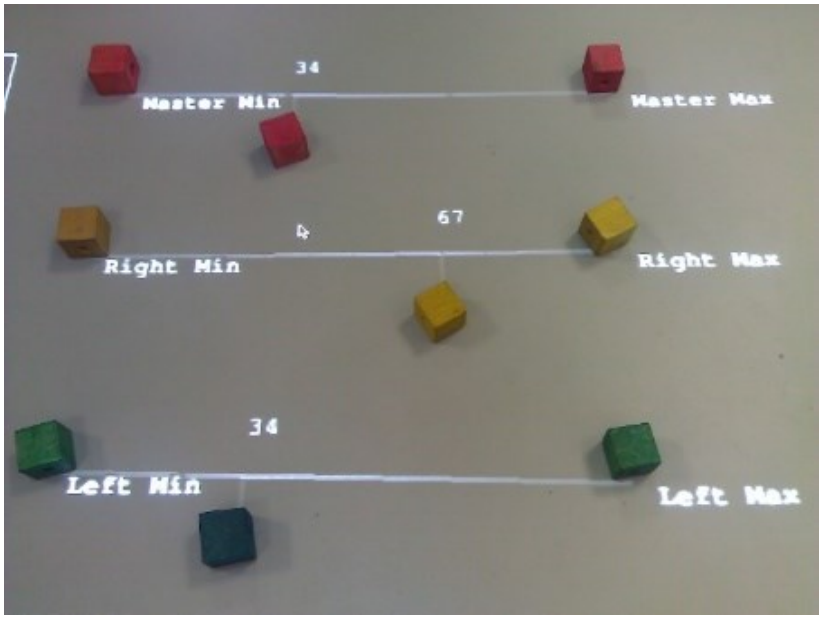

Figure 65 Basic multichannel ad-hoc audio control board, using mappings that provided values for the extremes for the maximum and minimum values of the required parameter

Different parameter-based functions are exposed in the patch panel for each audio channel available, as well as volume, balance and fade controls. Much like creating the timeline control for the video editor, the user presses the PowerMate button, selects the audio function group, selects the channel, volume, fader or balance function to control, selects the slider Action and then draws out a slider control. Using physical unit blocks (Figure

65) does not provide any additional direct functionality, but can help distinguish and identify the sliders and their current value.

5.4.1.3 PowerPoint Presenter

Mappings to control slideshow navigation and basic tasks (start/end, marker, etc.) in Microsoft PowerPoint[13] can be attached to physical actions with a marker. For example, the user can now just point to either the left or the right sides of the auditorium to navigate the slideshow. Using *Orientation between Positions*, one could even map the up/down, left/right orientations of a marker to the cursor position, allowing the user to control the cursor and write on slides at a distance.

Presenters would simple press the PowerMate button, select the slideshow function group and select, for example, the start slideshow function. In this case they want to be able to point to the projector to start, so the presenter uses the 'Object in Orientation' control (RotationAction) with and places their tracked marker on the table, confirming its selection before then pointing at the projector and pressing the PowerMate button in confirmation. The presenter can now start/end the slideshow by pointing their marker at the projector. Other mappings are then created using the same process.

5.4.1.4 Game/PC Controller

The game controller allows the user to create custom game and 3D application controls using joystick, mouse and keyboard emulation. Passive props such as a child's toy steering wheel can quickly be used as an actual control for a racing game, such as with the AR simulation by Oda et al. (2007). Users use function mappings that take in parameter as a value for the control (e.g. emulating a joystick or steering wheel control). The user selects the desired function, e.g. "Steering", and selects the "Object between Orientations" control before placing a mock steering wheel on the table and pressing the PowerMate button. They then rotate the wheel to the left and press confirm to define the 'minimum' extreme, before turning right and pressing the confirm button for the 'maximum' extreme. The system now interpolates between these two orientations, passing the associated value to the external steering wheel emulation function.

5.5 Discussion

Whilst previous systems have enabled the run-time creation of UI controls in the user's environment, they have primarily either focused exclusively on touch controls, or tangible

[13] http://office.microsoft.com/

controls. To author new controls, systems have relied on the user employing a mouse and a keyboard to create the control manually based on how the system sees the environments, e.g. drawing the control on the camera display in Light Widgets. The majority of previous tangible systems being fixed regarding functionality, such as WorldKit that allows new controls to be created, albeit for a limited set of functions. Some systems such as iStuff have acknowledged the requirement for new functionality to be incorporated into the system by the end user. However, iStuff's implementation of a patch panel approach still required the use of a PC to author any controls, relying on mappings that must be manually created for every mapping the user creates. TAM removes these limitations, as in addition to allowing both new touch and tangible-based controls to be authored by the user within their current context, they can be created without having to write any code to map individual controls to their associated functions. To overcome the limitation of purpose built systems, a simple yet flexible approach enables new functionality from external systems to be incorporated, both as inputs and outputs, allowing active electronics or software systems to communicate with TAM by simply executing a single application (i.e. using only one line of code). The ability for bi-directional communication with TAM also enables feedback loops to be created, where additional logic can be incorporated within external systems as required, and utilised as if it were native.

In exploring the comprehensibility of the methods of interactions, TAM (and the example implementation) support the three categories of physical objects as defined by Ullmer and Ishii (2000) (spatial, relational and constructive). Perhaps the most interesting is the support for constructive interfaces, where the user can utilise multiple objects together as a single control (e.g. the improvised joystick) whilst leveraging the affordances of the objects in consists of (e.g. the physical limitation of the mug's rim to define maximum angles).

The later classification of objects according to the use of tokens and constraints (TACs) is also supported. Given the user can utilise multiple objects together, they can define interactions based on the resulting control. For example the sliders and dials suggested by Ullmer and Ishii (as previous shown in Figure 8) can easily be supported using *Object between Positions* and *Object between Orientations*.

In addition to TACs, Ullmer also identified interactive surfaces. Whilst limited support for them exists (in the form of a touchpad control providing positioning information for a touch or object), another Action could easily be implemented to support detection of surface contours, generating inputs to the system based on the height at defined points,

etc. Despite not being implemented in the example system, the extensibility offered by the Action class approach means supporting such functionality would require minimal work.

Whilst the classification of the ad-hoc controls as ephemeral UIs, as defined by Doring et al. (2013) constitutes "at least one UI element that is intentionally created to last for a limited time", not only applies to the virtual controls, ephemeral UIs could also be applicable to constructive tangible controls. The creation of the improvised joystick as a constructive control (according to Ullmer) is by its very creation intentionally created to last for a limited time.

Following the results of the user study, along with using the interaction methods identified by Henderson and Feiner (2010) and Marquardt et al. (2011), a subset of those interactions were implemented in the example system to demonstrate the ease and flexibility of incorporating different interactions. Future work would include expanding interaction support to enable the full set of interactions as identified in the Proximity Toolkit.

Whilst a full evaluation of the system also remains as future work, public support and enthusiasm for the system from its limited exposure has been very positive. Aside from people seeing the application of the system to research, prototyping, and design, a number of applications outside the research domain already exist.

Chapter 6. Controlling Application Data Implementation

The need for controlling application data exists as previously discussed (Chapter 3). As such, this chapter looks at the problem of how can users interact with and manipulate the internal data of those external 'black box' applications by exposing the data to tangible ad-hoc interaction. This allows the physical objects to "become embodiments of digital information" (Patten et al., 2001). Whilst the previous chapter explored the creation of novel inputs as controls for existing application functions, the controls were limited to nullary functions or those that accept numeric values as parameters. This was achieved by treating the external system as a black box with a defined interface, through which required parameters could be provided. In this chapter, the motivation and current support for such an approach is outlined, defining the problem space, before an extension of the previous implementation is described to support tangible data manipulation. The chapter concludes with a number of example applications and related discussion.

6.1 Motivation

Interacting with application data in a tangible manner allows us to leverage our spatial intelligence as a part of our interactions with the data. Data objects can be physically grouped and stacked and then separated according to the immediate task at hand. For example, Figure 66 shows a video editor application developed where different clips from the user's computer are attached to physical surrogates, allowing the user to tangibly manipulate each clip. The icons displayed on each object are screenshots from the associated video clip.

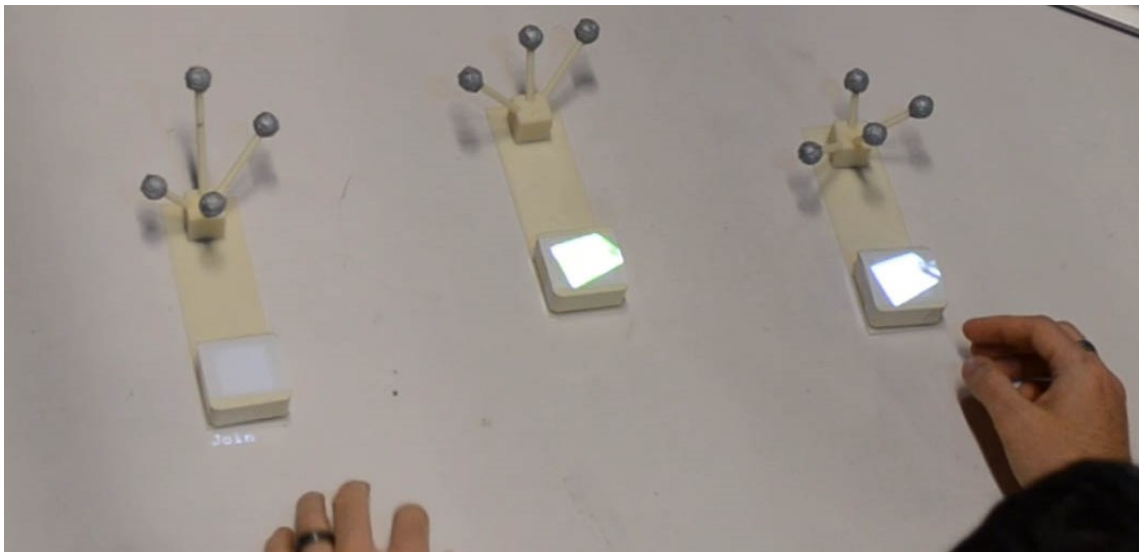

Figure 66 Video clip files associated with physical proxies

The motivation from this research came from the fact that despite previous applications allowing the user to manipulate data internal to the tangible application, there has been no level of extensibility or customisation. As such, this chapter seeks to explore an example implementation that answers the question of if limited code could be hosted within an external application, how could interaction with that application's data be supported using only the existing functions that would be required for such an external application without the tangible support? I.e. instead of looking at how to control application *functions*, looking at how to control application *data*.

Given the binary, application-specific nature of this data, some application-specific code is obviously required to perform the associated tasks directly on data objects. Following this, supporting the generic editing of data within application-specific data containers would represent an incomprehensible task, as the ad-hoc system would need to support editing every possible type of application data possible. As such, this investigation sought to explore how such a system could support defining and editing relationships between those data objects, instead of editing the data objects themselves, thus reducing the explicit need to interpret the data object itself. Using this approach, users can then freely interact with an unlimited number of data objects using physical proxies, defining relationships based on direct physical interactions. This removes the problem of how to support interacting with application-specific data, whilst still allowing users to edit aspects of the data by leveraging the advantages of TUI interaction.

Previous tangible systems have enabled the user to directly manipulate application data and the relationships between different data objects (e.g. Bricks, Build-It, etc.). However, given these systems are closed and purpose-built, specific interactions can be associated with different data operations. Up until now, there has been no way for a tangible system to be aware and support the manipulation of data from external systems for which it was not explicitly designed. Whilst standards such as XML and JSON have come a long way as to enabling universal readability, the interpretation (and thus 'rendering' and editing) of that data is still application specific.

6.2 Current Support for Interacting with Application-Specific Data

Previous attempts at exposing application-specific data to other applications have been in the form of either converting data to a common format (if possible), or through utilising a software framework that supports sharing, such as Microsoft's Object Linking and

Embedding[14] (OLE). Whilst applications regularly share primitive data (such as text and images) through mediums such as the clipboard, the data is converted into a number of different, universally supported formats (e.g. JPEG) by the source application, allowing the target application to select the format with which it has the greatest level of support. For example, copying a table of data from Microsoft Excel might result in the Excel-specific data object being copied to the clipboard, as well as text, HTML, and XML representations along with an image of the table in PNG and JPEG formats. With this, the target application can decide which format has the greatest support, and utilise that format. Pasting the data into another Excel document would user Excel's own data object (given it has the highest level of support), whereas pasting such data into Microsoft Paint would use the un-editable PNG or JPEG image representation. In addition to copying the data itself, the target application may also copy a reference to the source element using OLE.

Applications supporting OLE provide a reference to their application-specific data in a generic OLE container. This container is then inserted into other OLE-supporting applications, with the container data and rendering actually be performed by the source application. Editing of embedded data is often supported via double-click, where the source application then launched to edit the data in its own context. As such, even widely adopted standards such as OLE do not support the actual editing or understanding of binary data across applications, rather just its embedding via generic containers. As previously highlighted in this chapter, whilst the emergence of XML as a form of data-interoperability has meant the data itself can easily be shared between applications, the meaning and understanding of that data is still application specific (aside from the few specific universal file formats). As such, instead of trying to edit individual data objects themselves, this iteration instead explores editing the relationships between, and functions performed with, those objects within the system, as at most this requires minimal updates to the existing external system to call the related functions that already exist to modify those relationships. A simple example of this approach would be to be able to change the ordering of slides in a slideshow, but not change the content of the actual slides.

6.3 Implementation Design

The implementation extends the previous TAM implementation for creating ad-hoc user controls and was extended in the following ways:

[14] http://support.microsoft.com/kb/86008/en-au

- A DataObject class was created to track known instances of data that exist within the external system.
- SendData.exe and the associated message handler were modified to allow external applications to send messages regarding data updates.
- Two data-related Actions, DataSinkAction and DataListAction, were created to allow for the selection of a single data object and a list of data objects. This involved the creation of a StringProperty type to allow a single Property instance to contain a space-separated list of data object identifiers.
- The XML mappings were modified to support "%data-typeX" and "%dataList-typeX" parameter types, allowing the user to select a function including such parameters and have the system create the required Action and VirtualMapping instances for that type.
- Sifteos[15] (small active electronic devices that allow for their detection when placed next to each other), were incorporated into the system as a means to provide a physical proxy for data that can easily be detected when in close proximity with other data objects (without using the OptiTrack system).

The remainder of this section discusses each point, and how it relates to the rest of the system.

6.3.1 Tracking Data Objects

The implementation uses a separate instance of the DataObject class for each instance of data, storing the following information about it:

- the data instance type (string),
- a user friendly name, which may be blank (string), and
- an optional representation for the data object (image).

DataObjects can be associated virtual regions on the table to allow them to be interacted with using touch, as well as being able to be associated with an InteractionObject as an attribute for when a physical proxy is to be used. Both the virtual regions and InteractionObjects are passed to Action's evaluation function, allowing data to be evaluated alongside physical interactions when Action looks for a given interaction.

[15] http://www.sifteo.com

6.3.2 Data Instance Notifications

As discussed previously (Section 3.4.5), data objects are defined as having an identifier, type, name and visual representation (in this implementation this is an image). External applications can notify the ad-hoc system of data objects in the same two ways it can pass information to the patch panel, via running the SendData.exe executable with the required parameters, or via passing those same network parameters via a TCP connection. To create a new data object in the tangible realm, the external system sends a new object command (Figure 67).

"newDataObject *id, type, name, imageRespresentationURI"*

Figure 67 New data object command

Such an instruction notifies the system that there is a new data object with the given identifier, type, name and the URI (Universal Resource Identifier) of an image to use as the visual representation. This URI could reference a file on a common file system or globally accessible network URL resource.

Upon a new data object notification, a new DataObject is created and associated with a virtual control region on the table as a projected image and name (an empty name can be provided if the external system wishes to only use the visual representation). The new data object is presented in the middle of the interaction area with the image provided and name displayed below it. If an object is already in the centre, the offset for the new object is modified in a counter-clockwise, outwards-spiral. In addition to this, the external system can also provide the name of a function, that is also defined as an output node in the XML patch panel, as a hint as to which user control (if any) the new data object should be located (Figure 68). For example, for a file explorer, a new folder function might be associated with a simple button. When the user presses it, the ad-hoc system is notified of a new data object, the folder, with a hint that the 'new folder' function is responsible for it, and is thus placed alongside the button. Whilst this approach is not precise, it does attempt to provide a level of congruency between the cause and effect of the new data instance.

"newdataobject *id, type, name, displayImageURI, optionalFunctionNameHint"*

Figure 68 New data object command with a hint for location

The size of a data object as it is displayed on the surface is defined by the size of the representation image provided by the host application (1 pixel = 1 mm). A 50x50 pixel

image would create a 50x50mm physical representation on the surface. If no image is available, a white outline of a 50x50mm square if used.

Future changes to data objects are propagated to TAM in a similar way to new ones. Changes and updates to a data object are sent using a reduced command providing only the identifier of the data object to update, along with any changes to the name or representation (Figure 69).

"updateDataObject *id, name, imageRespresentationURI"*

Figure 69 Update data object command

This allows the system to update the name and visual representation of the data object with the given ID. Similarly, data objects can be deleted at run time with a simple command (Figure 70).

"deleteDataObject *id"*

Figure 70 Delete data object command

6.3.3 Interaction Support

In order to interact with instances of data, two Action subclasses were created to allow for the selection of single instance and the selection of collection of instances of a given data type. To select a single data instance, a DataSinkAction is used. Data Sinks (Figure 71) enable the user to define a physical area where data instances can be placed (using physical proxy or using touch) to be selected and passed to an external function, providing a 'sink' to pass data to external systems. As can be seen in Figure 71, data instances (in this case digital images) can be selected by being dragged onto the Data Sink.

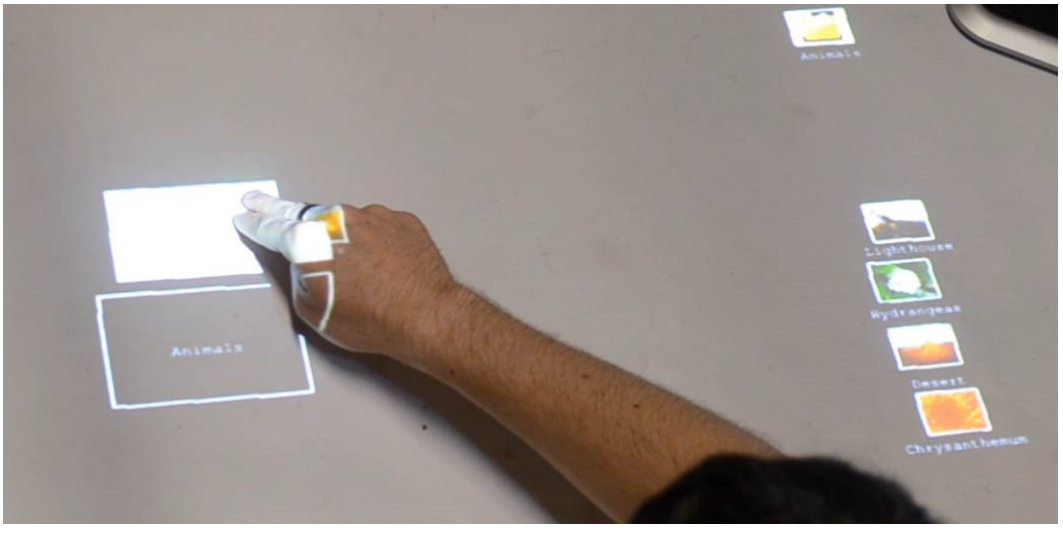

Figure 71 DataSinkActions (left) being used to select a data object (an image) to be associated with a given tag by dragging a data object onto it

The ability to select collections of data objects is supported though the DataListAction subclass. DataListAction follows a spatially aligned collection of data objects, passing the resulting list to the required function as a list of space-separated values stored in a single StringProperty. Whilst Data Sinks provided a distinct event for when the function should occur and what data object should be used, using the spatial alignment of data objects does not provide the same event. Given the user may put a number of objects 'off to the side' when no longer needed or just whilst working on a different task, how can the system ensure the function is not called when the first two objects are co-located, and then again when the third object is placed alongside, and again for the fourth and so on? Even when the final collection is spatially defined, how does the system know which data objects are first versus last in the list (e.g. reading right-to-left or left-to-right)? To resolve the event generation and ordering ambiguity, when the DataListAction is created, the Action creates its own special data object as a 'List' data object. Using this object, spatially collocated data objects will be ignored by DataListAction, until the special List object is placed alongside the desired 'first' data object in the collection. This provides a distinct event whilst also identifying which collection of data objects in the scene the user wants to use and the intended ordering of those objects. Figure 72 shows the joining of several video clips using physical proxies.

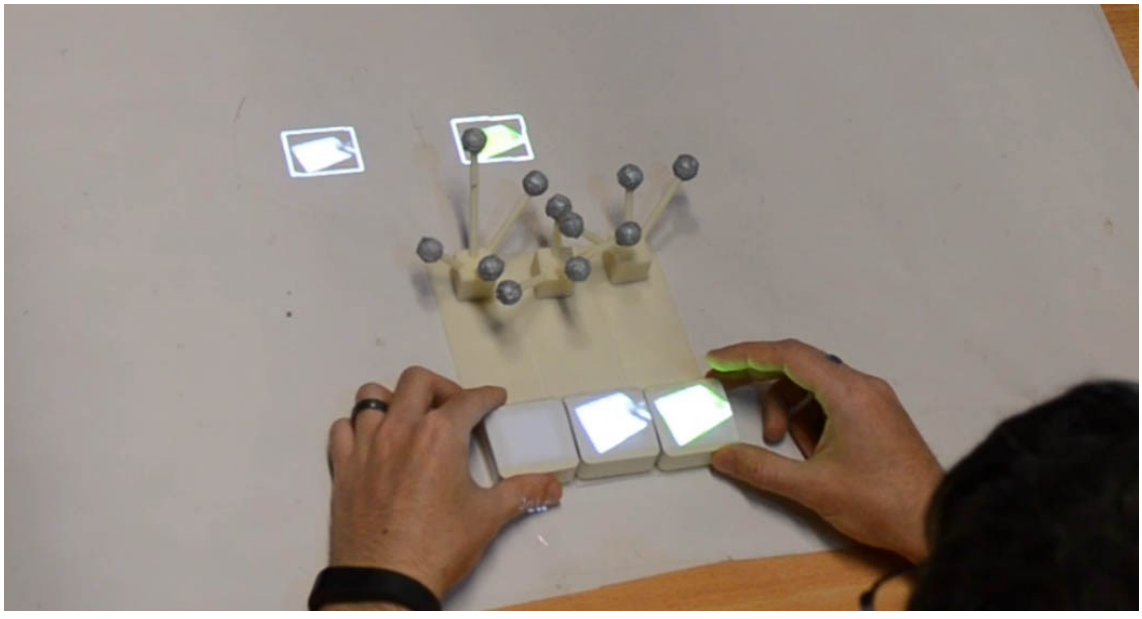

Figure 72 Placing the 'Join' data element alongside a desired collection of video clip data objects (the label for the Join can be seen on the user's thumb)

For example, a DataListAction might check if any two or more data objects are next to each other, and if so, trigger that Action and allow them to be passed to an external function, e.g. to associate them. However, in order to do this, the data objects identified

by Action need to be able to be passed through the patch panel to the receiving VirtualProperty that actually executes the function. As such, when the Action is triggered, DataListAction follows the list of spatially aligned data objects and creates a space-separated list of object identifiers stored in a single StringProperty. It is this Property that is mapped to the corresponding output function that requires the list of data objects. Using space/comma separation provides a simple and effective way to communicate a collection of unknown size, however more elaborate encodings of the identifiers (such as using XML or JSON in the StringProperty) could be used.

Data can be associated with physical objects as proxies by placing a tracked object on top of the projected object. Associating data with physical objects allows for faster interactions, allowing the user to manipulate multiple objects at once, picking them up, repositioning, and stacking, them as required. Data can be disassociated with physical objects by shaking the object, at which time the data object becomes touch-based again and re-appears in the middle of the interaction area.

Projected objects can be quickly grouped by the user dragging across multiple projected data objects, with each object following the user's finger as it collects other objects. The group can then be dragged and manipulated as a collection, or separated by pulling each object out individually.

6.3.4 Mapping Support

In order to define functions that require references to either a single or a list of data objects, two additional mapping parameters were defined. This first mapping allows an external application to request that an executable or function call in the patch panel be passed the identifier of a single data object of a given type. A simple mapping from the implementation of this design can be used to demonstrate this (Figure 73). This patch panel mapping is almost identical to previous ones that took a parameter from the user's input (e.g. a parameter from a slider or dial valuator). However, instead of using "%i" to indicate that an integer value is required, the use of the "%data-" followed by the name of data type indicating that that function can only occur when there is an associated data object of given type is provided.

```
<output name="Delete Object" execute="DeleteObject.exe %data-typeX" />
```

Figure 73 Patch panel declaration to receive a single data object

When the user selects the "Delete Object" function to control, the only possible Action that can be used is the DataSinkAction, and thus the user is automatically prompted to

create one. Whenever a data object is within the bounds of the Data Sink, the reference to that object is passed out through the VirtualProperty, replacing the "%data-typeX" parameter.

Similarly, there needs to be a mapping that can request collections of objects as a list. Instead of using "%data-typeX" to request a single object of type typeX, "%dataList-typeX" is used to request a collection (type) of objects of that type (Figure 74).

```
<output name="Merge Objects" execute="MergeObjects.exe %dataList-typeX" />
```

Figure 74 Patch panel declaration to receive a list of data objects

Whilst TAM currently supports lists as the only collection type, other types such as hierarchies could easily be supported by defining pairs of identifiers in the list to define the hierarchy. In addition to this, different collections can be created in different ways. For example, an external application could force the creation of a list one object at a time, by having a function accept only one data object. This means the user could only add one object at a time to the list, with the list actually being created and stored by the external host application and not TAM. This gives flexibility to the role that TAM should have if the external host application wants greater to influence how certain operations are performed.

6.3.5 Sifteo Incorporation

To easily support the definition of spatial relationships, Sifteos were integrated into the system. The core idea being that as data objects could be associated with physical objects, having the ability to detect when small, handheld objects were located next to each other in hardware would be advantageous and provide advantages over using the existing optical tracking.

Sifteo cubes (Figure 75) are the result of the commercialisation of the Siftables research product (Merrill et al., 2007) into small, low-cost computers that could interact with one another. The current commercialisation has led to their development as a children's toy, with two different generations of Sifteos with distinctly different advantages. Both generations provide 30x30mm LCD displays with a single push-button input on the screen, along with shake and flip detection as well as the ability to detect when other cubes are aligned to a cube's sides. However, the first generation required the cubes be in continuous communication with a PC which was responsible for hosting the different applications being run on them. Sifteo programs were executed on the PC, allowing for external libraries, TCP sockets, etc. to be incorporated as part of the program. The second

generation however comes with a dedicated embedded computer that allows the cubes to be used without a dedicated PC, and as a result, applications developed for the second generation do not allow the use of libraries outside of the limited Sifteo set, as everything is run on the dedicated device in a closed system. This presents a limitation, as the second generation provides no ways for actions performed with the cubes to be communicated with external systems, whereas the first generation supports such functionality. Both generations however do not allow images or content to be loaded onto the devices at run-time, instead all content is required to be loaded onto the cubes at compile-time. As such, any images TAM shows on the cubes (e.g. the visual representation of a data object) must be shown using projection, instead of each cube's internal LCD display. However this is just a limitation of this particular implementation.

Figure 75 First generation Sifteo cubes

The integration of Sifteo cubes offered a simple way to support the distinct detection of different actions (flip, press, shake and alignment with other cubes) performed with small tangible objects. To support the first generation cubes, a Sifteo program is run simultaneously to TAM, sending Sifteo state changes to TAM via TCP. The Action class in TAM was subclassed, receiving these state changes and generating events as required as triggers for the TAM patch panel. This allowed, for example, for the user to press the screen's button on a cube's face as a nullary trigger for input in the patch panel.

To associate data objects with a physical object in TAM, OptiTrack markers are attached to physical objects (Figure 76), including Sifteos, to provide 6DOF tracking. Given the small size of the Sifteo cubes, the markers were attacked to a flat plate, some distance from the marker. Whilst appearing cumbersome, they are merely an artefact of the using the available optical tracking system. A dedicated system would ideally track the objects from below the surface. When a physical object is placed on top of a projected data object,

an InteractionObject is created for the physical object (if it does not already exist) and the data object then associated with that InteractionObject. The InteractionObject then continues to render the visual representation of the associated data object on the physical object. When a Sifteo cube with data attached detects a shake, the data is detached and appears on the middle of the interaction area for touch-based interaction or to be attached to another physical object.

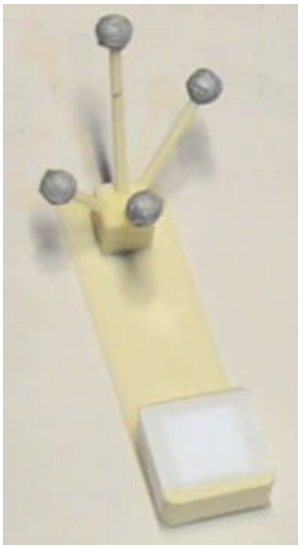

Figure 76 Sifteo cubes with attached OptiTrack markers

6.4 Example Applications

A number of example applications have been developed to demonstrate the available data-related functionality, with a photo tagging application and video editor used to demonstrate how data objects from an external application can be manipulated. This section provides an overview of each application, what TAM-specific code is required for it to function, and how the user goes about interacting with each application.

6.4.1 Video Editor

The video editing application allows users to play, clip/split, and re-order/join individual video clips using a user-defined UI, either in conjunction with the application's GUI or solely using tabletop interaction. A simple editor was created to demonstrate the fundamentals of application data control, however existing editors could have been used. The video editor was written in C# with .NET 4, and provides functions for loading, playing, and navigating a video. Additional controls are available for clipping the currently loaded video clip into smaller clips, before then joining those clips into larger ones. The application uses Windows Media Player as the actual media playing

component. All functionality for the video editor is available using the GUI. The whole video editing application (including GUI) took under a day to make.

Given the use of key-frames and other compression techniques along with the increasingly high resolutions used in modern video codecs, performing even basic tasks such as splitting and joining create significant processing overhead as the existing files must be processed to create new files for each operation, created a significant delay in the user's workflow. To enable the repeated clipping/joining of video clips on-the-fly, the application uses AviSynth[16] to emulate these actions on video clips. AviSynth is a scripted video codec that enables the repeated clipping/joining of video clips on-the-fly, essentially emulating such operations in real time, removing any immediate processing overhead in exchange for some processing during playback. Instead of containing encoded imagery, an AviSynth file contains commands about how other video clips should be clipped, audio dubbed, etc. When the file is played, the AviSynth codec parses the file and performs the scripted commands in real time, passing the result to the video playing application as if it were embedded in the actual AviSynth file being 'played', which is essentially just a video script being run in real time. Figure 77 shows a simple AviSynth script that crops and joins two different video files between given frames. Given AviSynth files themselves register as video files, commands can reference other AviSynth files instead of video clips, creating a hierarchy of nested AviSynth processors passing frames to other AviSynth processors.

```
AVISource("E:\FirstVideo.avi")
Trim(3075, 3825)
AVISource("E:\SecondVideo.avi")
Trim(1554, 2653)
```

Figure 77 Example AviSynth file that crops and joins two different video clips using frame offsets

The implemented video editor (Figure 78) provides the same basic controls as the previous video editor (play, pause, stop, skip to position, etc.). However, provides a number of additional functions:

- a data type for video clips,
- a data type for displaying a large thumbnail of the current video frame,
- an open function for loading video clips from the tangible context,
- a nullary clip function to define start and end points for cropping,

[16] http://avisynth.nl

- a save function to create a new file of the cropped sections, and
- a join function for joining multiple video clips in the tangible context.

The use of a generic 'clip' function allows the currently loaded video to be cropped at various timeline positions, creating a new AviSynth file in the source folder. These files can then be iteratively loaded and cropped. At any point, different video clips can be selected and joined with the join function, which creates a single file constituting all the individual clips as one. Each function is defined in the patch panel as single output (discussed later).

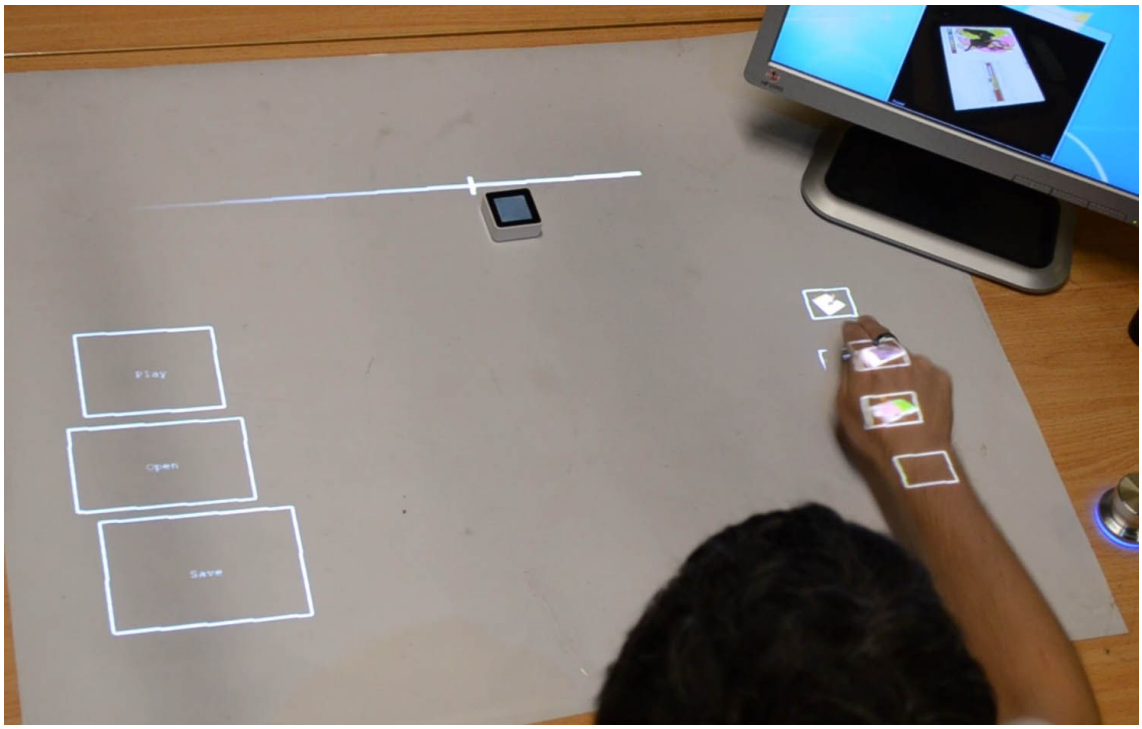

Figure 78 Example implementation of a tangible video editor showing video clip instances (using associated frames from the video clip as icons) that are not yet associated with physical proxies on the right-hand side

6.4.1.1 Creating the Tangible Video Editor

To create the tangible video editor, the associated timeline, play, and save controls can be created as previously described in section 5.4.1.1. To demonstrate the creation of data manipulation controls, the following three examples are described in detail in the following sections: creation of user controls with Sifteo cubes, creation of Data Sink controls, and the creation of interactions

For the generic clip control, the user presses the PowerMate button, selecting the video editor group and the clip function, before then selecting the Sifteo control Action. The user then picks up a Sifteo cube and presses its screen to select that cube, associating that cube's screen press with the clip function. The cube is then placed on the timeline slider. Now, the user can slide the Sifteo cube across the timeline, pressing the cube to define

start/end points for cropping. Once any number of points have been defined, the user can press the save button, at which point an AviSynth video containing the selected frames is generated, and the associated video clip data object displayed on the interaction area.

To open the various video clips, the user again presses the PowerMate button and selects the open function from the video editor group. The open function takes a single video clip data instance as a parameter, and thus is created as a Data Sink control. The user creates the sink using the same process as creating a button, by defining a labelled 2D region on the interaction area. Any video clip data instance (projected or associated with a physical object proxy) can be placed on the open function region to have the video editor open that video file for playback, navigation and cropping.

To join any number of different video clips, the user can create a control to join them. The user presses the button to create a new mapping, and selects the join function from the video editor group. The join function (discussed in more detail later) is a single mapping requiring a list of video clip data instances to join. As such, when the user selects the function, a single "Join" data instance is created in the middle of the interaction area (Figure 79). To join any number of clips together, the user places the desired clips (either as a projected control or by associating them with physical proxies) alongside each other. To trigger the join function and define which clip in the list is the 'first', the user places the "Join" data instance at the beginning of the list (Figure 80). When this occurs, the DataListAction generates the list of data object identifiers for the video clips in the list, and passes it to the external join function, which joins the clips and creates a new video clip instance of the joined clips. Once all the controls have been created, the process of opening, cropping/splitting and joining the various clips is an iterative tangible process.

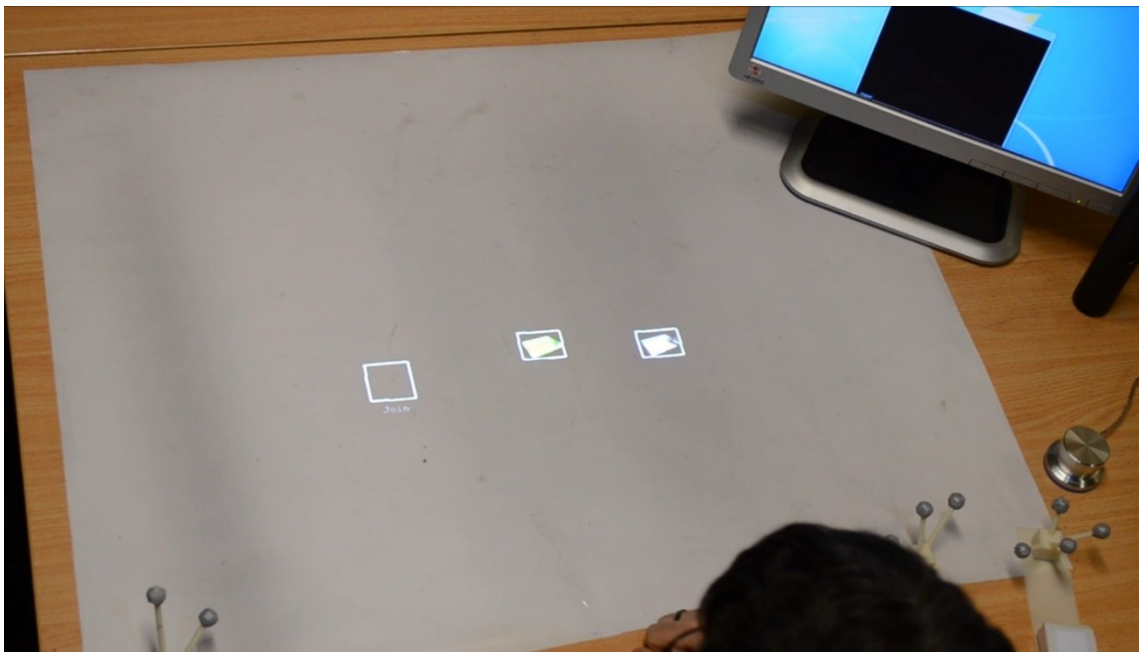

Figure 79 The "Join" data instance visible on the left side with other video clip data instances

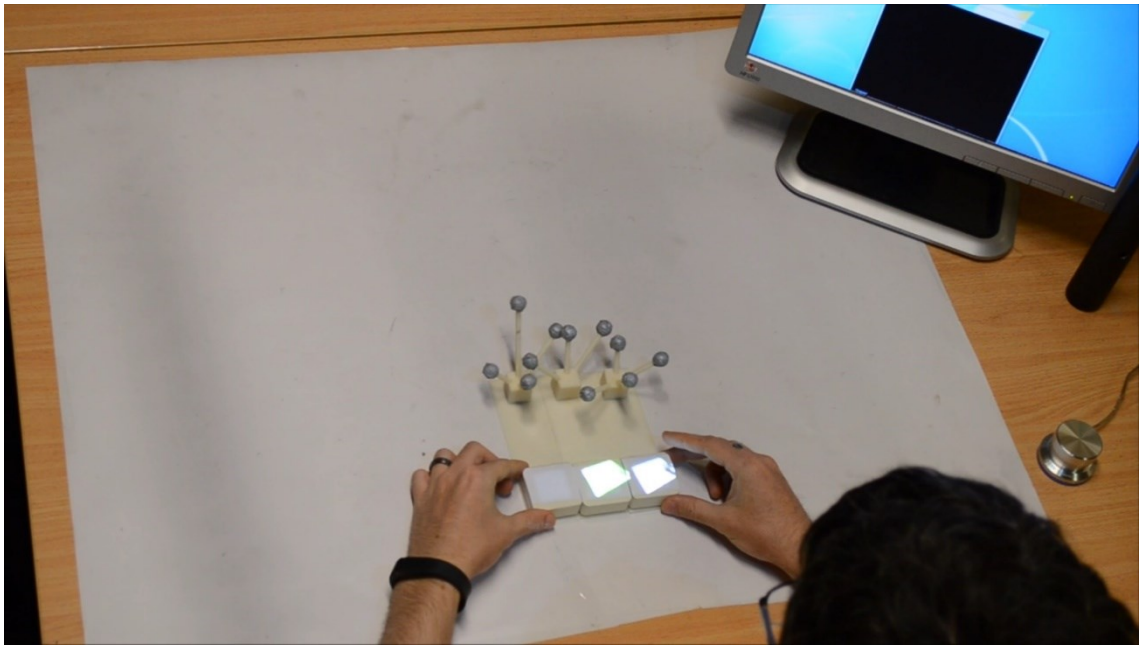

Figure 80 Joining two video clips together by placing the "Join" data instance at the beginning of the colleciton

6.4.1.2 Data Objects Used

To enable the TAM implementation to control the video editor, two different data types were used: a video clip type and a video preview type. All functions used in the patch panel used the video clip type, with the existence of a single video preview type instance used to enable the user to view the currently loaded video pervasively. Whilst the preview cannot be interacted with directly, it allows the user to see the currently playing video inside the physical realm, or optionally they can use the physical controls in conjunction

with viewing the current video on the existing GUI. As video clips are manipulated, they appear on the table-top as 40x40mm thumbnails (as were visible in Figure 78).

To enable this, two things had to be done. The first was generating thumbnails for the video clips. This was achieved by capturing the rendering of the video from the screen as the video clip was opened/saved. This thumbnail is then just saved to a network drive as a JPEG using the same filename as the current video clip. The second was to insert a shell execute call in the video editor function that creates new video clip files to notify the ad-hoc system of a new video clip data instance. Using this approach, whenever the video editor internally creates a new video clip, it automatically notifies TAM by passing the new object command as a parameter to the SendData.exe application. A network drive is used as the universally accessible location for the visual representation of the data as JPEGs. The data objects' identifier uses the application's own unique clip identifier, which is just the video clip's filename.

Two additional calls were inserted into the video editor's code to create a data object for the video preview, and a timer-event based call to update the preview. Using this approach, whatever is being displayed by the media player control in the GUI is continually saved to a file and displayed on the tabletop surface using the video clip preview data instance. Using this approach, the video preview can be positioned based on the user's desired configuration for their tangible UI. Whilst it appears as a fixed 100x100mm image, it would be trivial to provide patch panel mappings the change the size of the image being saved by the video editor, and thus change the size of the video preview image.

6.4.1.3 Patch Panel Functions

Aside from initially loading a specific file on the computer, all functions (play/pause, stop, navigate, split, join, save, and audio controls) in the video editor are exposed to TAM. Despite the open file path command not being exposed, a file-explorer could actually be implemented using folders and files as data objects of two different data types and a Data Sinks to open/load them. This is partially explored in the photo tagging example discussed in the next section. The individual functions exposed to the ad-hoc system are described in individually later in this section.

When a video editor function is to be called, the VirtualProperty starts a new instance of the video editor. This new instance just passes the data to the existing video editor instance via a .NET IPC component that provides an event call back, processCommand()

(Code Excerpt 3), with whatever parameters the sending application was executed with. This data is then parsed for commands of interest, where the internal functions of the application at then called with the corresponding parameters if required. As such, only a single function had to be added to the video editor to enable control from the ad-hoc system. None of the other code is TAM specific, aside from single lines of code for creating/updating data instances.

```
//Process received commands, first index is the command, subsequent ones
//are parameters (integers or data instance identifiers)
private bool processCommand(string cmds[])
{
    if (cmds[0] == "play")
    {
        Play();
    }
    else if (cmds[0] == "pause" || cmds[0] == "stop")
    {
        Pause();
    }
    else if (cmds[0] == "position")
    {
        //move timeline to a given position
        setPosition(cmds[1]);
    }
    else if (cmds[0] == "openfile")
    {
        //load a given video file
        Open (cmds[1]);
    }
    else if (cmds[0] == "clip")
    {
        //clipping a video involves defining a start and an end, so
        //different functions are called for each
        if (clipBegun == false)
        {
            clipStart();
        }
        else
        {
            clipEnd();
        }
    }
    else if (cmds[0] == "clipstart")
    {
        clipStart();
    }
    else if (cmds[0] == "clipend")
    {
        clipEnd();
    }
    else if (cmds[0] == "save")
    {
        //save the current video, with SaveVideo returning the filename
        string id = SaveVideo();
        //create the new saved file tangibly using its identifier
        newDataObject(id);
    }
    else if (cmds[0] == "join")
    {
        //ideally we should have more than one clip to join, so get
```

```
            //the array of them of based on space-separated parameter
            string[] clips = (cmds[1]).Trim().Split(' ');
            //join them, storing the result filename as the 'id'
            string id = Join(clips);

            //open and play the joined clip
            Open(id);
            Play();
        }

        return true;

    }
```

Code Excerpt 3 Function for receiving commands via IPC

Play/pause and stop are simple nullary functions in the patch panel that call the media player's own play management controls. For tasks such as splitting a video at a given position, the application actually requires two different things: the position at which to split, and the command to do so. Despite TAM only supporting atomic external output functions, the user can still create hybrid physical controls that provide both parameters implicitly. The position function is declared in the patch panel as a valuator with a corresponding cut function declared as a nullary function. To create the user controls, the user first creates a valuator (most likely a slider) to navigate to the cut position, and then could use a Sifteo collocated on that slider to trigger the actual split command. By using the movable Sifteo placed on the slider (as was shown on the top of Figure 78), the user can link the two functions as if they are in fact composite, which they are from the user's perspective. The user performs a single fluid action to cut a video in multiple places, moving the Sifteo across the slider, pressing the Sifteo as they move to define the cut sections.

More elaborate controls could of course be created. As previously highlighted (Section 5.4.1), a (blunt) film guillotine could be used as the 'split' action by monitoring the orientation of the guillotine blade, which would support an analogue 'function follows form' analogy. A Data Sink could be created under the guillotine, so any data object placed there would be loaded before being 'cut'. Physical pieces of film could even be tracked between two positions on opposite sides of the blade, allow the user to move the film under the guillotine to select where to cut.

To allow users to load/play different clips, the open file function was exposed to take a single video clip data object (Figure 81). Since the data object identifier that is passed back is the clip's filename, this can be directly passed to the open function of the media

player. This mapping allows the user to draw a Data Sink that opens and plays any clip placed on it. Using this, the user can grab and instantly load and play any clip on the table.

```
<output name="Open" execute="NotifyApp.exe open %data-clip" />
```

Figure 81 Open file command declared in the patch panel

To join clips, a join function was declared in the patch panel (Figure 82). The declaration requests a list of video clip data objects that is provided by TAM from the user. With the declaration, TAM generates a 'Join' data object that can be placed alongside any collection of video clip data objects located next to each other, at which time TAM will follow the spatially aligned series of data objects, passing the identifiers of each instance back to the video editor. Whilst the video editor then joins those clips and produces the result as a new video, the code in the host application may also chose to delete each of the individual clips constituting it. This way, once clips are joined, the resulting clip appears and the individual clips immediately disappear as they are deleted. The decision to include such functionality where clips are deleted after being joined would depend on the intended workflow of the user.

```
<output name="Join" ratelimit="5000" execute="NotifyApp.exe join %datalist-clip"
/>
```

Figure 82 Join video clips patch panel declaration

With this functionality, users can iteratively load, play, navigate, split and join different video clips entirely within a table-top context. Depending on the desired workflow, some or all of the functions be used as complimentary controls alongside traditional GUI use. For example, the ability to create a dedicated navigation/splitting control alongside the user's keyboard may be advantageous for short sessions for intense editing by allowing the user to quickly scan different video for splitting points, with the rest of the tasks and data management performed in the GUI.

6.4.2 Photo Tagger

The second application is a set of mappings for a photo tagging application that allows the user to tag photos either individually, by folder, or by custom grouping on the table. There is a single tagging application that accepts a tag string and the path to a file to tag. Each tag string appears as an entry in the patch panel, with tags defined as Data Sinks by the user (shown as the 'tagged' regions on the left of Figure 83). The user first presses the PowerMate button, selects the photo tagging application and any tagged mapping listed.

The user then defines a Data Sink region for that tag, with the name of the tag appearing in the middle of the region.

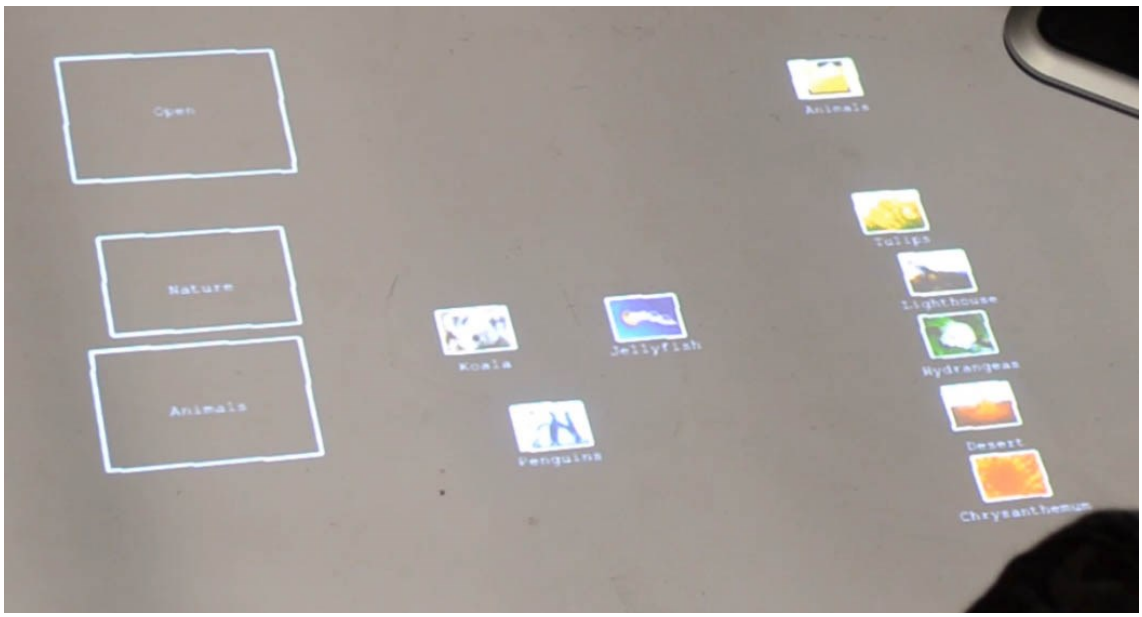

Figure 83 Example photo tagger layout with tag Data Sinks and a collection of images and a folder (top-right)

To use the application, the user uses the GUI to select a root folder, which the tagging application then recursively scans looking for folders containing JPEG files. When it finds individual files, they appear on the tabletop as distinct thumbnail previews with their file name beneath them, ready to be dragged onto a tag Data Sink to tag it. If a folder is found to have more than five images in it, the folder itself appears on the tabletop (top-right Figure 83), allowing the whole folder to be tagged.

Instead of just having to tag whole folders (if they contain more than five images), the use of an 'Open' function as a Data Sink allows an individual folder to be opened by dragging a folder onto it, with the individual images in that folder then appearing on the table. All other data objects currently on the table are removed as the user 'enters' that folder. The user can then tag each image within that folder. A separate 'Close' entry in the patch panel provides a function to close an individual folder, returning to the initial collection of images and folders. The 'Open' and 'Close' functions appear as two distinct patch panel entries requiring a single file/folder parameter (discussed next).

Since folders and individual files (ideally two different data types) can both be tagged by the same Data Sink tag, and since the Data Sink patch panel mapping can only accept a single data type, both image and folders must be exposed to TAM as being of the same data type. If different types are used for both folders and images, there must different mappings for each data type. This comes back to the limitations and balance between how

much logic should be put into the ad-hoc system versus relegated to the external application functions. To enable the patch panel to accept more than one type for a single mapping, only one data type is used for the XML declaration, but the data instances' identifiers are prepended with a string to indicate which type they are. This 'sub-typing' is ignored by the ad-hoc system, but means when the photo application receives a data instance, the external function's logic can check what type the data is and act accordingly. For example the Open/Close functions for folders do nothing if passed a data instance of an image type, executing only if the data instance identifier as the folder string prepended. Whilst this approach is a limitation of the implementation in this instance, such an approach could be used to ensure the external application retains more control over the individual mappings, as was previously discussed (Section 3.4.5.2) regarding how much responsibility should be abstracted/attributed the ad-hoc system versus the existing external system. However in this scenario, such a capability is a limitation to be addressed in future work, e.g. using hierarchical data types.

The photo tagging application demonstrates an example form of tangible data interaction where operations can easily be performed on groups of objects, either as they are (according to the existing file system hierarchy) or based on the users interactions to segregate different objects on the table, tagging groups at a time.

6.5 Discussion

Whilst previous systems enabled the tangible editing of data as a single, specific function of the system, their development as purpose-built systems limited their application outside very specific use cases. Given the interoperability problems that exist with interpreting binary data across applications, this implementation sought to support the manipulation of relationships within that data, utilising existing external application functions for manipulating the data. Essentially, data just becomes a different type of parameter mapped through the patch panel to the receiving VirtualProperty (and external function). As opposed to providing an integer value encapsulated in an IntegerProperty, a StringProperty is used to encapsulate one or more data objects identifiers. Complex relationships between the data objects can be defined using pairs of identifiers, allowing for more complex structures, such as hierarchies, to be tangibly created and communicated to external systems. This approach ensures the distinct separation of functionality and logic from the tangible and external systems (as defined by the MVC and MCRit models), whilst also providing some flexibility regarding how specific interactions are performed. For example, should the external application want more

control, it can request individual data objects and build up different data structures internally over time by the user repeatedly selecting individual data objects with Data Sinks.

Despite the functions in the patch panel appearing to be defined in an atomic manner, composites can be created, either as special Actions within the system (generating multiple output Properties), or as defined by the user as multiple individual controls. For example, the splitting a video in the video editing application required two things, the command to split the video, and also the position at which to cut. As without a position, the video cannot be split. Such constraints can be relegated to the UI, for example using a Sifteo cube to set the position slider control, with the press screen event on the Sifteo mapped to the split action. Whilst actually being two distinct functions, they can occur simultaneously in a complementary fashion.

Whilst not implemented, the architecture supports composite data functions, where multiple data objects from an Action stored in a hierarchical manner can be passed to a hierarchy of individual VirtualProperties. Using such an approach, one could have an Action generate a data object video to be split, as well as the position at which to split it at, which would be passed to two different VirtualProperties in a single PropertyMapping. This allows more than one piece of data to be associated with different functions, with the use of such an approach up to the individual implementation.

Aside from appearing to only support application data, the same constructs could be used to support things such as a tangible file explorer. Using type file types as files and folders, with ‘Open’ and ‘Close’ function mappings, the contents of a folder could be displayed as two different data types, with folders being able to be opened or closed, and files only opened. A primitive version of this was implemented in the photo tagging application, where the user can open folders tangibly, to tag individual photos, or use the folder object itself to tag the whole folder.

The use of an optical tracking system (the OptiTrack) is far from ideal, as the requirement for optical markers to always be visible to cameras to track objects creates issues for smaller objects, especially when they are in close proximity, i.e. when defining collections of data objects. When multiple markers are in close proximity, the different combinations of markers from different objects is matched by the tracking system as being different individual objects, resulting in data objects ‘jumping’ around the scene as false-positives are detected from object markers near each other. It was thought that by

moving the markers further away from the objects themselves (Figure 84), with each marker at different distance from its associated object, there would be less false positives, however that was not the case.

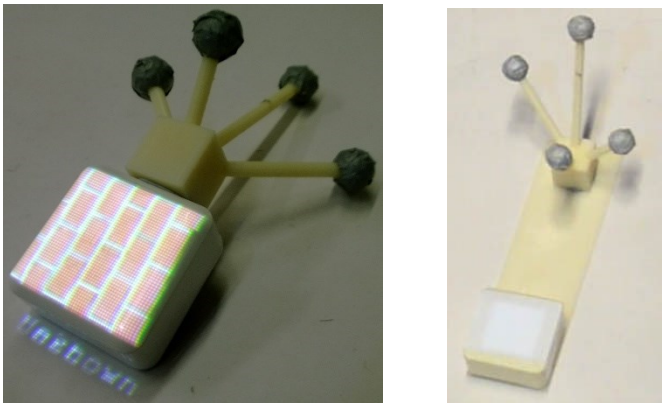

Figure 84 Optical markers as originally attached to Sifteos (left), and later located at varying distances from the corresponding objects (right)

Ideally, a sub-surface tracking system would be used to identify objects from below the table based on a fiducial or other marker, however the choice of the OptiTrack was purely based on availability and ease of use. Whilst the application functions under ideal conditions, anything outside of that causes issues, and as a result prevents any evaluation occurring with the current system. Future work will look to replace the system with a rear-tracking one, removing the false-positive matching when objects are in close proximity. However, it is important to note this as being a limitation of this specific implementation and not the underlying architecture proposed.

6.6 Summary

Building on the TAM architecture as previously used to define UI controls, this chapter has demonstrated how it can support the incorporation of individual data types for data-based interactions. The patch panel and associated Action and VirtualProperty classes were modified to support the requirement for data objects of a given type, allowing the definition of external functions to define the types of data objects required for that interaction to be valid. Two example applications were shown, demonstrating how external systems could easily be replicated or created by end users on-the-fly tangibly using the TAM architecture.

Chapter 7. Defining Application Logic Implementation

The previous two chapters have described two classes of ad-hoc interaction (AH-UI and AH-Data) that enable the ad-hoc creation the tangible UIs and data manipulation. This chapter examines an example implementation for a third kind of interaction, defining new application logic in a tangible ad-hoc system (AH-Logic). The implemented system allows users to introduce previously known objects into the system, defining a name and number of number of different roles/groups to which the object is a member. Using this, the user can create simple rule-based interactions involving physical objects, and more complicated interactions using logical groups of objects.

This chapter is structured as follows, the design and implementation of the system is described, addressing the features and requirements previously identified in Chapter 3. A formal evaluation is then presented, demonstrating the feasibility of the implementation, and more importantly the TAM-architecture in general, for end users. The chapter concludes with a discussion of the implementation.

7.1 Implemented System

As previous highlighted in the Supporting Logic section of Chapter 2, the end goal of enabling extensible logic is to allow users to quickly create simple, domain independent, interactive systems, without the system having any background knowledge about what they are wanting to create. This involves a number of different aspects for designing the system (as previously highlighted and discussed throughout this dissertation):

- Allow users to create simple, domain independent, logic/rule-based interactions within the system using arbitrary objects, without the system requiring any prior knowledge of the task or content being used.
- Allow the creation of those rules to be generalised to avoid repetitive entry.
- The programming of the system should, ideally, follow the natural progression for how the user would describe the same scenario to another person.
- The system should not have any distinct development/usage stages. Rather, the system should allow the user to create interactions as part of normal system interactions (akin to tangible PBD).

The implemented system allows the ad-hoc definition of interactive systems based on simple If-This-Then-That (IFTT) logic, created using a form of tangible PBD entirely within the tangible workspace, fulfilling the above requirements. PBD was used instead

of PBE, as PBE would require user to perform same interaction multiple times with unpredictable results occurring when they were attempting to create successive or conflicting interactions. In addition, systems utilising inferencing create procedures that are both complex and unstructured (Myers, 1986), which creates problems when the user wishes to later edit that functionality. Despite the focus on PBD, the TAM architecture supports both PBD and PBE as previously mentioned.

7.1.1 Using the System

Creating basic interactive systems with the implementation is simple, with the basic system flow following that highlighted in the initial preliminary studies that users:

1. introduce and define the objects,
2. define any common aspects between them,
3. define any interaction logic, and then
4. interact with the system.

The system was designed to not have distinct development and usage stages. Instead the workflow is flexible, allowing users to jump between steps based on the required workflow. To explore different modalities, gestures were used for system navigation instead of the PowerMate button, with the details of the gesture support discussed later.

In the following example, the user will create a simple system to demonstrate the previously highlighted example (Chapter 3) of a chemical reaction. In this case, the reaction will be using basic maths-style unit blocks to represent a reaction between hydrogen and chloride, to form hydrogen chloride. To explore different input modalities, gestures were used for system navigation, as discussed later in the chapter.

7.1.1.1 Introducing Objects

To begin, the user places the first block down on the table, and opens one of their arms out towards it, as if to gesture, “here it is” (Figure 85a). The system detects this gesture as ‘introducing a new object’, and projects an outline around the object, along with a projected keyboard in front of the user. The user is then prompted, via brief instructions using text on the top of the interaction area and a Text-to-Speech (TTS) voice prompt, to enter a name for the object using a virtual keyboard projected in front of the user (Figure 85b), before pressing the “Confirm” button (located in place of the Enter button).

The system then presents an RGB colour palette and prompts the user to select a default colour for the object (Figure 85c). The user can then touch any colour to have the object(s) augmented with that colour using the projector (using a slightly increasingly scaled

version of the object contour detected by the Kinect). To confirm a choice, the user is prompted to make the 'Confirm Gesture'. The Confirm Gesture requires the user to place their forearms vertically, about should-width apart, as if holding a camera to their face to 'capture' the current configuration (Figure 85d). When the gesture is detected, the system plays a camera shutter sound to confirm the current configuration has been 'captured'. The combination of both colour and the system displaying the object's name alongside the object address the primary methods of object identification used the preliminary study. After selecting the default colour, the system then resets back to the normal system state, where the user can just interact with the system.

In the chemistry example, the user places a block on the table, performs the introductory 'here it is' gesture by extending one arm towards the object, enters a name of "Hydrogen", and confirms it. They then select a default colour (e.g. green), before performing the confirm pose. The user then repeats the process to introduce another Hydrogen atom, placing the object, gesturing, entering a name and picking the same green colour. Whilst each object was introduced individually, if multiple objects are present on the table, they can all be introduced at once, with the same properties set for all. The user now has two individual objects. To create a reaction, there needs to be two different reactors, so the user places a third object, labelling it as chlorine and colouring it orange. The system now has three known objects that have been introduced.

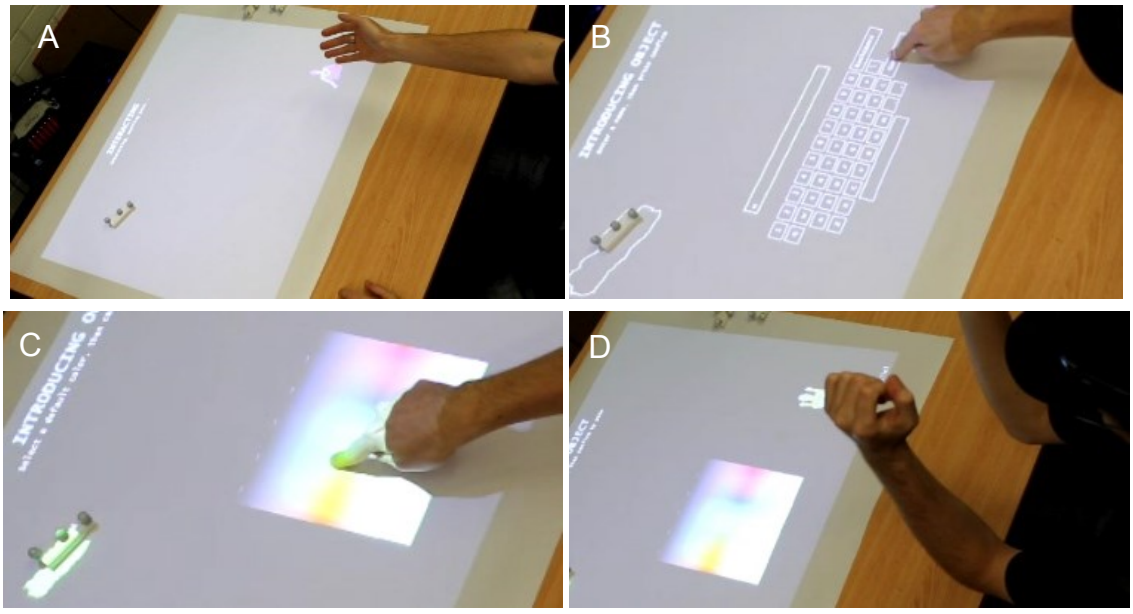

Figure 85 Introducing a new object (A) performing the introduction pose, (B) entering a name, (C) selecting a default colour and (D) confirming the selection.

Figure 86 Close-up of the virtual keyboard

7.1.1.2 Defining Groups/Classes/Types of Objects

To enable 'generic' interactions, where users can define a series of interactions without having to define every permutation of them, the user can create 'groups' of objects (sharing some common aspect, e.g. their class). With this, users can create groups of objects, where any one (or multiple) objects in that group can be used in interactions, allowing a form of substitution. For example, a single object (A), defined as being from a group, G (incorporating objects A and B), can be used to create an interaction with the system. As a result, once the interaction has been created, either object A or object B, can be used as part of that interaction since the interaction only requires a single object of type G, which either A or B can fulfil.

To define a group of objects, the user opens both arms out towards the objects, as if to gesture "here they are" (Figure 87, left). The system detects this gesture, and provides instructions using TTS and text for the user to select which objects belong in this 'group' of objects, whilst also presenting a keyboard for the user to label this group (Figure 87, right). Users point at each object involved in the group, with the system highlighting each object with its default colour when selected. The user must also enter a name for the group, before then pressing the Confirm button. If multiple objects are introduced at once (i.e. multiple objects on the table when the introduction gesture is performed), a group including all those objects is created by default.

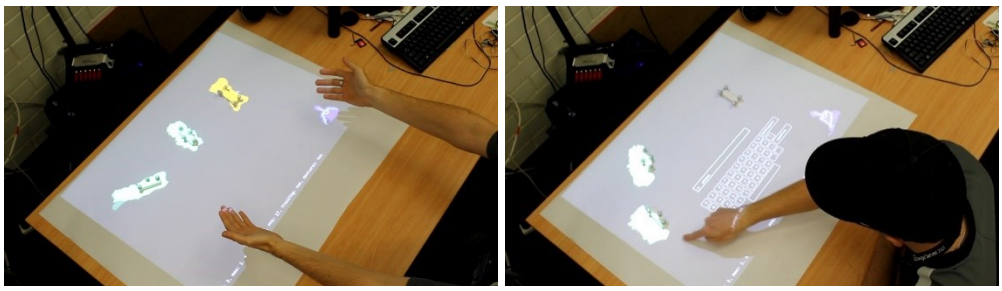

Figure 87 Defining a group of objects with gesture (left) and object selection and naming (right)

In the chemistry example, whilst as far as the user is concerned there are two hydrogen objects in the scene, the system must explicitly be informed that there is a common attribute between the objects. So the user extends both arms to gesture for a new group, points at each of the Hydrogen objects to select them, enters a name for the group, e.g. 'H Atoms', and presses confirm. The system is now aware not only of the three individual objects, but also that two of those, the ones we've labelled as Hydrogen, are part of a common 'group'. As previously mentioned, had the two hydrogen objects been introduced at the same time, this group would have been automatically created by the system by default.

7.1.1.3 Creating Interactions

Now that the system is aware of the three distinct objects, as well as the fact that there is some common factor between two of them, the user can now create a simple interaction (i.e. a chemical reaction) between them. For this interaction, when any hydrogen comes in contact with the chlorine, a reaction will occur. To create this interaction, the user gestures towards the table with both forearms placed horizontally at 90^{o} to each other (Figure 88a), as if pointing to objects with both hands saying "looks at this". The system detects this as the user wanting to create new interaction, and prompts the user to select which objects are involved. The user then points to each object involved (Figure 88b), for this example pointing to a single hydrogen and the chlorine. The user then performs the confirm gesture by placing the forearms vertically parallel.

Because the user selected an object that was part of one or more groups, i.e. the hydrogen object, the system prompts the user to resolve an ambiguity regarding if the user is referring to the specific object they chose, or just any member that was part of one of the groups that object is in (Figure 88c). This is presented as a semi-circle pie menu around the object, where the default selected option is the object's name, with the other options the name of each group that it is a member of. A semi-circle pie menu was used as to show the different options as they relate to the physical object, appearing to the user as if

the menu options were encompassing the physical object. The user selects which role the user was referring to for this interaction, before then performing the confirm gesture. The system now knows which objects and/or which groups/roles of those objects are involved, and the user is prompted to define the actual interaction to be monitored. For the reaction, the user moves the hydrogen and chlorine in close proximity, and makes the confirm action, with the system then playing a camera shutter noise as it 'captures' the configuration. Now that the system has the trigger condition that defines the interaction, the user must define what happens when the reaction takes place. To do this, the system presents the colour palette and prompts the user to define what should change when the interaction occurs. The user then points to the chlorine to select it, before then touching red to change the colour to indicate the reaction (Figure 88d). The user performs the confirm gesture, and the system resets back to the default interaction state.

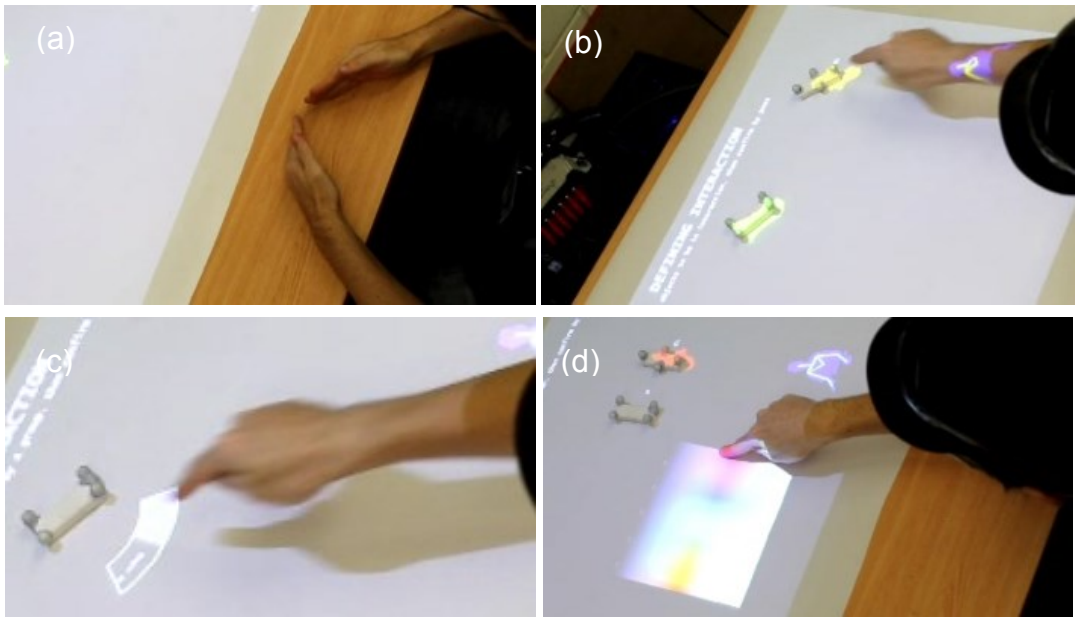

Figure 88 Creating rules for an interaction (a) performing the new interaction pose, (b) selecting objects involved, (c) resolving group selection for substitution and (d) performing the changes as a result of the interaction

7.1.1.4 Interacting

Now with the interaction created, the user can move the hydrogen object in close proximity to the chlorine, and the system will change from the default colour to the red colour set to indicate the interaction. When the objects are separated, the colours reset to default. Because the user created the interaction using the hydrogen group of objects as opposed to just that specific hydrogen object, the user can move the other hydrogen object (which was not used during the creation of the interaction) in close proximity to the chlorine object, and have the same reaction occur. This shows a form of substitution and allows a single interaction to be created but to be generalised to involve other objects,

preventing the user from having to manually create an interaction for every possible permutation. Internally, an Action instance just measures and stores the distance between the two (or more) objects and monitors that any object within the selected group fulfils the requirement of being within that distance, plus or minus ten percent. The group is just a shell InteractionObject containing the two hydrogen objects, which is used by the Action instance to evaluate if any of the InteractionObjects within the parent instance match the required criteria. The chemistry example represents just one example of how such a system could be used, with others including other educational aids, 'war table' planning, game design, etc.

7.1.2 Creating the System

The implemented system internally uses a state machine (Figure 89), however the ordering of different primary tasks is at the user's discretion and may be repeated in any order as required. Four main states exist; general interaction, introducing objects, defining groups, and defining interaction logic. Gestures are primarily used to switch between states.

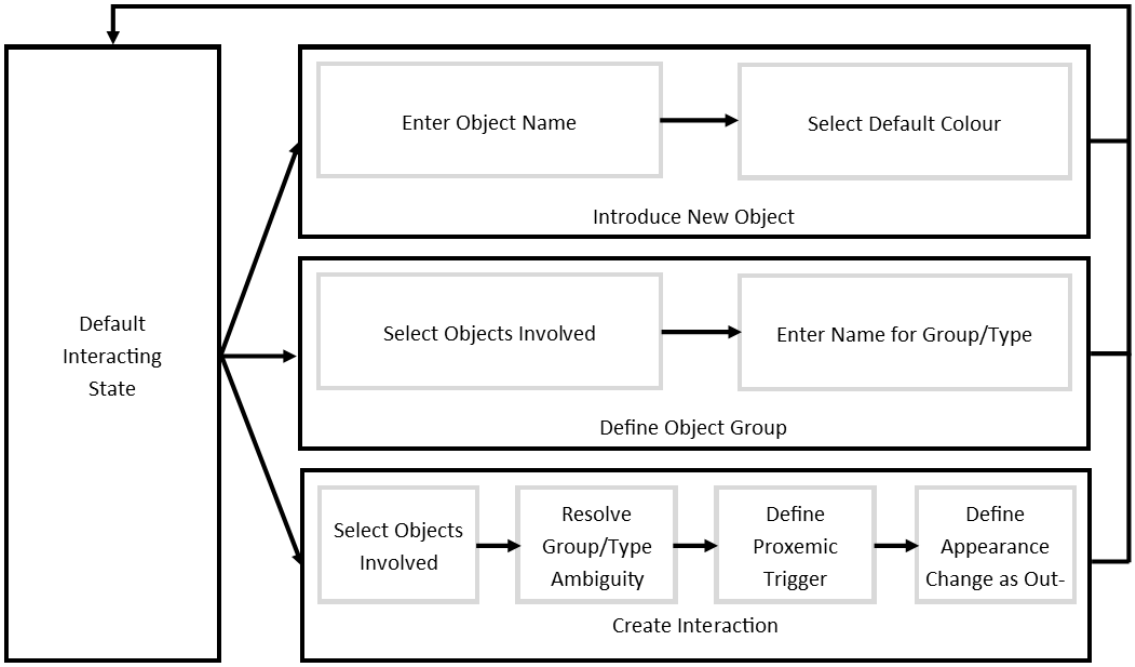

Figure 89 Implemented state machine

An InteractionObject is created for each physical object in the scene, containing the following individual Property types:

- PositionProperty: contains the object's current position,
- OrientationProperty: contains the object's current orientation,
- ColourProperty: the object current augmented colour, and

- ContourProperty: the object's contour.

The InteractionObject implementation has a render function that displays the object's name and augmented colour using the values stored in its PositionProperty and ContourProperty.

7.1.2.1 Detecting Objects in the Scene

A number of different existing software packages were used to simplify the development of the system, each providing distinct functionality. To detect objects and touches on the surface, a simple multi-step process (Figure 90) is applied:

1. When the system starts, it captures 30 consecutive depth frames of the blank surface using the OpenNI[17] Kinect software. These frames are then 'averaged' to generate a background mask.
2. This mask is then 'subtracted' from each subsequent frame, leaving only the outlines of objects that were not present during the initial background capture.
3. Whilst these outlines provide the full outline of each object, in order to detect touches the system requires the ability to detect fingers on the surface. To do this image is threshold to only show outlines for objects within 10mm of the surface.
4. OpenCV[18], an open-source computer vision toolkit, is then used to detect the contours and centre points of each outline (touch) present in the frame.
5. Whilst this process until now provides touch events, it cannot provide drag events, as there is no correspondence between touch points across frames, causing a drag action to appear as a series of individual touches over a path. As such, the TUIO toolkit (Kaltenbrunner et al., 2005) is then used to match touches/drags between frames based on the touches that were present near that location in the previous frame. TUIO simply matches points in the current frame to the closest points that existed in the previous frame.

[17] http://www.openni.org/

[18] http://opencv.org/

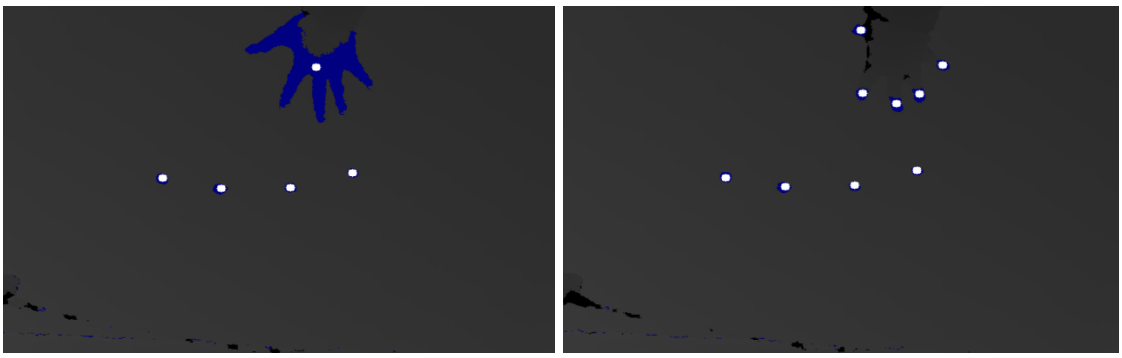

Figure 90 Detecting unit block and touch contours and centre of mass using a depth camera thresholded to different depths (left and right)

In summary, the system generates a background mask of the surface, subtracts it from subsequent frames, thresholds to a certain depth above the surface, then uses OpenCV to identify touches and TUIO to track them frame-to-frame. Note that in order to detect the objects in the frame, the OpenCV contour detection is performed before the thresholding, otherwise tall objects will appear as a series of small touches around their outside (similar to Figure 90, right). Whilst touches can be used to track objects between frames, the OptiTrack is still required to track objects as when an object is picked up, moved or released, the merging or separating of contours means the system cannot be certain what contour is what. Whilst tracking systems that user feature point detection can be used to enable marker-less tracking, such as those using SLAM/PTAM techniques (Davison, 2003), the dynamic nature of ad-hoc interaction (movements, handling by the user, obscured feature points, etc.) makes it unreliable in its current state. Given the focus of this investigation is on supporting the interactions, and not the underlying tracking, the assumption is made that future advances will replace the need for dedicated external tracking systems, as previously highlighted.

To highlight or augment the colour of arbitrary objects in the scene (Figure 91), the detected contour in rendered using OpenGL as a simple polygon (Figure 92). Whilst the approach does introduce a small amount of jitter in the contour between frames (given the accuracy of the Kinect between frames), it allows for the immediate and dynamic augmentation of objects without requiring any modelling by the user. Recent advances following this investigation in the modelling of ad-hoc environments using technologies such as Kinect Fusion (Izadi et al., 2011) would allow the system to passively generate accurate 3D models of objects that would allow for more precise augmentation.

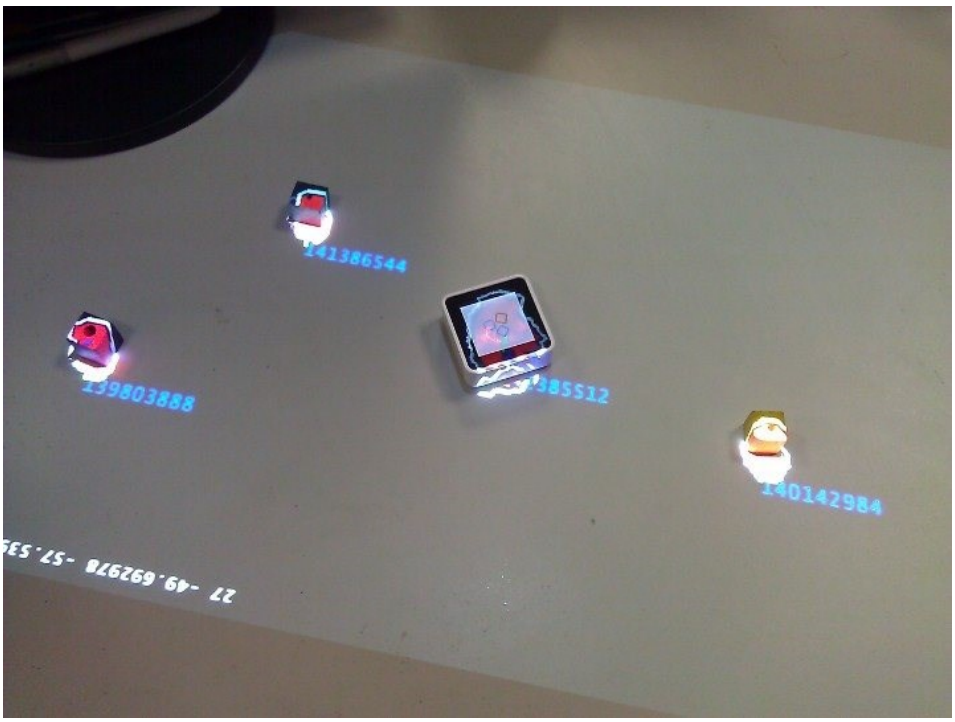

Figure 91 Early development showing the TUIO identifiers for random objects

Figure 92 Showing dynamic contour detection and colour projection for an arbitrary object

7.1.2.2 Interacting

To enable the user to interact based on the proximity of objects, a ProximityAction was used. ProximityAction captures the distance between any number of objects, and matches against those distances in subsequent frames with a 10% margin of error. To enable generic interactions, any sub-object in an InteractionObject can fulfil the requirement of its parent. As previously mentioned briefly, the two individual hydrogen

InteractionObjects are part of a third, encompassing InteractionObject that serves as the hydrogen group. If a ProximityAction instance uses the encompassing instance, then the Action will check if any of the individual InteractionObjects contained within the encompassing instance fulfil the ProximityAction's condition, allowing for a single rule to be defined that can be triggered by any number of different objects. If the ProximityAction's evaluation function finds the interaction has occurred, an associated PropertyMap modifying the associated object's ColorProperty is executed. Whilst this implementation is limited in the types of physical interactions it can sense and the output result, there is no reason why further implementations could not be extended with external functionality and other Action types. Given the focus on extending application logic and the user generation of rules, the limited capabilities of this implementation allows any evaluation to focus on if users can understand how to represent the problem scenarios in the required structure for the underlying architecture, as opposed to how they can navigate the implementation-specific menus to define what triggers/outputs they want to control.

Interactions with PropertyMaps that result in conflicting outcomes (i.e. two simultaneous interactions wanting to change the colour of a single object to different values) are both executed. However to ensure a consistent output, the PropertyMap of the Interaction instance involving the most objects is executed last, ensuring a consistent result (resolving the two conflicting segments previously identified in Table 1). For example two separate interactions involving objects A, B, and C and objects A and B objects would result in the PropertyMap of interaction A, B being executed first, but later overwritten by the PropertyMap of A, B, and C.

7.1.2.3 Gesture Navigation

The use of gestures over other forms of navigation was based on:

- Wanting to use a different input modality to that being used within the system (as outlined from the preliminary study).
- Supporting a more natural method of interaction where (ideally) system navigation commands were part of the interactions themselves, i.e. no additional explicit commands should be required.
- The use of gestures is pervasive, and does not require the user to possess anything specific object or have certain things available (e.g. a flat tabletop). Gestures can be performed regardless of the environment in which the user is located.

- In addition to the previous point, the use of gestures ensures the interaction area and surfaces remain entirely available for user-defined content, free from system artefacts.

As opposed to using gestures involving some movement over time, the system instead looked for distinct poses held by the user. To detect them, a second Kinect faces the user to view their movements, with the OpenNI framework used for tracking. OpenNI provided abstraction from the RGBD sensor hardware, providing low level depth information, as well as skeleton tracking (Figure 93) and pre-defined gesture detection, with 24 unique skeleton joints tracked by the system.

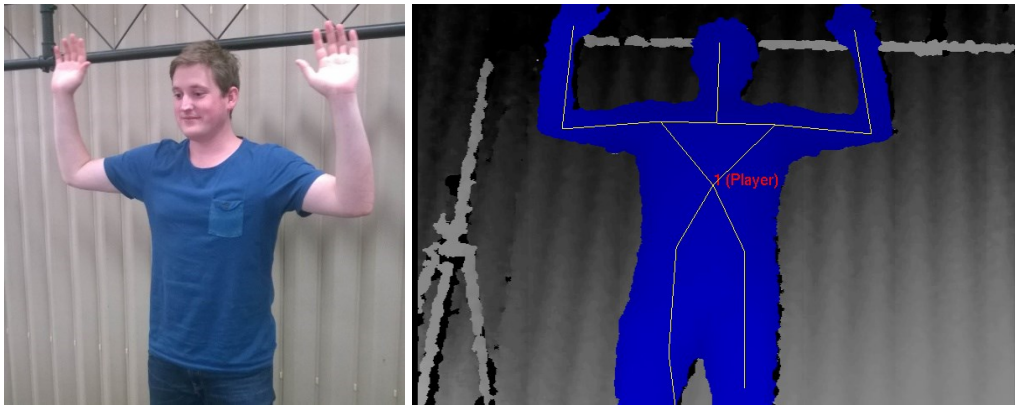

Figure 93 OpenNI skeleton tracking, image courtesy of James Baumeister

Given none of the default gestures provided by OpenNI were relevant to the tasks, a dedicated pose-detection system was written from scratch using data from the OpenNI skeleton. The angles between the user's elbow, shoulder and torso were matched against custom values to detect when a gesture was occurring. Based on an example of each gesture, angles in the user's skeleton were matched with individual tolerance values based on repeated trial runs. The different poses supported by the system were evaluated sequentially, and ordered so that multiple gestures involving similar physical actions evaluate the more complex gesture first. For example, the user opens one arm to introduce an object, and opens both to create a group. To avoid the introduction gesture being detected when the user is performing the group gesture, the group gesture is evaluated first, and only if that is not matched, is the 'simpler' introduction evaluated, i.e. if part of gesture A involves the same gesture used for gesture B, then A must precede B in evaluation.

To avoid false positives occurring when the user is interaction normally with the system, the system requires users to hold a pose for one second before the associated action is performed. To provide feedback regarding when the system was monitoring for a held

pose, a small feedback window is shown in the bottom-right of the interaction area (Figure 94), with the user's body (as detected by OpenNI) projected in blue and skeleton in yellow (the default colours used by OpenNI). When a gesture is matched, the user's body turned white (as to provide contrast to the normal state) and the name of the gesture being matched, e.g. "Introduction Gesture", displayed underneath. Skelton segments matched with less than 70% confidence as determined by OpenNI are rendered in red as to indicate to the user that the system was having difficulty tracking them. To cancel out of any state, the user simply raises either hand vertically, as if to gesture 'stop', with the system resetting to the default interaction state. The full list of gestures can be seen in Appendix A – User Study Material.

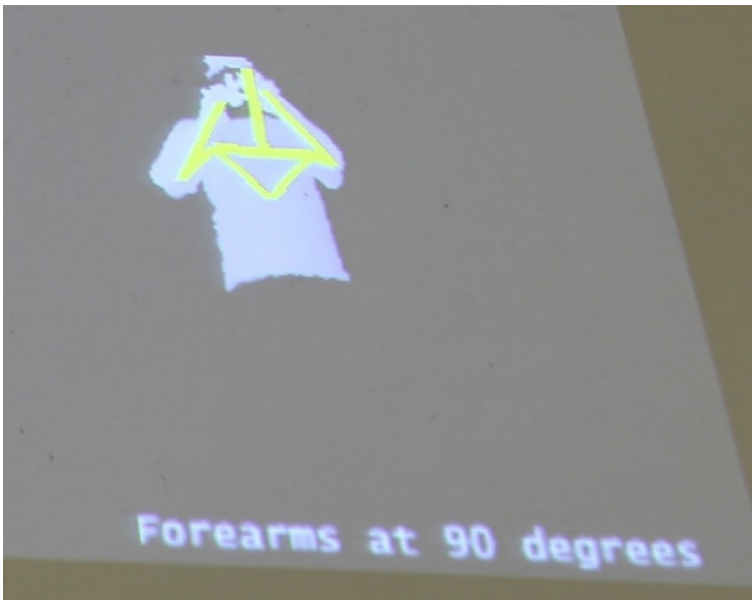

Figure 94 Gesture feedback on the tabletop

7.2 Evaluation

Following the development of the system that allows the creation of novel logic-based interactions by end users, a user evaluation was conducted to evaluate how well users would be able to construct interactive systems using the system without prior training. This evaluation of an adaptable system follows such previous studies as Klemmer et al. (2004) and McDaniel (1999) in evaluating the effectiveness and efficiency of adaptable and adaptive systems. This section provides an overview of the design and development of the user study and its implementation. The results of the study are then discussed as they relate to the core architecture and implemented system, along with the outcomes of the study regarding associated changes.

7.2.1 Study Goals

As highlighted in Chapter 2, one of the issues in evaluating adaptable systems comes from their very nature – how do you evaluate something that can, ideally, be adapted to anything you want it to be? As previous discussed, in looking at automated adaptive UI systems, Paramythis et al. (2001) suggested breaking the adaptive system down into individual components (e.g. the AI detecting the user actions, what they are trying to achieve, what modifications should be made, etc.), evaluating each on their own. However this does not work for non-automated systems where those distinct components do not exist.

Like McDaniel (1999) in developing the PBE game generator Gamut, the justification was made not to directly compare the system against other systems (i.e. a study between-systems). As the end goal is to enable users to create ad-hoc systems without any background knowledge, the study should instead evaluate whether users can utilise the system in such a way instead of comparing it against others (McDaniel, 1999). Given the system used is only one implementation of the architecture, it was thought that by evaluating users within the context of ART, the study could evaluate how many attempts were required by users to obtain a given goal. If the underlying architecture to which problems must be adapted is intuitive and thus requires little exploration by the user to understand and use, there should be little epistemic interaction, with the majority being pragmatic interactions to directly achieve the known goal (Kirsh and Maglio, 1994). Using this approach allowed the study to evaluate how well the user understands what they are trying to achieve, instead of just measuring the time this specific implementation requires to achieve it.

As previously described (Section 3.1), ideally a system such as this would allow the user to describe the scenario however they saw fit, as if describing it to another person. Whilst there will always be a level of adaptation imposed on the user due to the limitations of the system (e.g. input modalities, language support, etc.), this investigation seeks to minimise the translation required to match the limitations. As such, for the user to utilise as system such as this, two things must occur for the user to fully externalise their internal thoughts into an ad-hoc system:

1. Users must adapt their view/design of their mental system to be created to match that supported by the ad-hoc system.
2. Users must then translate that knowledge into the ad-hoc system.

As a result, the evaluation was designed to evaluate:

- How easily can users grasp the concepts involved in TAM to convert a scenario into the required structure?
- How easily can users then communicate that structure to the ad-hoc system?

If users can easily understand the concepts involved in the system, e.g. introducing objects and the use of groups to define common types for substitution, and translate an example scenario requiring those concepts into the ad-hoc system, then the system is effective in its core goal of enabling such interactions to be defined by the user. Whilst the implementation-specific UI and methods could be evaluated on their own, this approach allows the evaluation to explore if the requirements of the TAM architecture are accessible for end users.

7.2.2 Study Design

In designing peripheral tangible interaction, Edge (2008) described the use of mock tasks to evaluate TUIs with potential future users, with the general format being:

1. system introduction,
2. observation/recording of task performance, and then
3. survey-based feedback.

The system introduction step employed a similar design to Scott et al. (2005) in trying to under how users are engaging with and understand the system, with participants sitting alongside a display and collocated interaction area.

Participants were seated at a desk and asked to watch a video showing a single, group-based interaction involving three objects being created using the system. The interaction used the chemical reaction example, where any hydrogen entity coming into contact with chlorine would trigger a reaction, changing the colour of the chlorine. Two hydrogen objects were introduced, and placed in a group together. The chlorine was then introduced, and an interaction created so when any object in the hydrogen group was in close proximity of the chlorine, the reaction occurred. To ensure equal training across participants, all participants could only watch the video once, but could pause it and ask questions at any time. This use of video over written instructions has been leveraged in similar novel PBD evaluation tasks[19].

[19] http://www.willowgarage.com/blog/2012/11/13/enabling-end-users-program-new-robot-skills

Following the video, participants were asked to create three scenarios, each conceptually building the last, to demonstrate a simple tabletop game. The complete each tasks, participants were required to:

1. Introduce an object for each entity in the scenario.
2. Define (if any) logical groups between those objects.
3. Define any interactions using either the individual objects or their groups.

The exact ordering of these was flexible, meaning the user did not have to explicitly introduce all the entities at once, rather they could just introduce what was required for the current step.

The tasks conducted using the above structure consisted of:

1. A neutral, sovereign country being invaded by an attacking army. Upon the attacking army reaching the country's borders, they would take the country over, changing the nation's colour to red. This required a single, simple interaction (Figure 95).

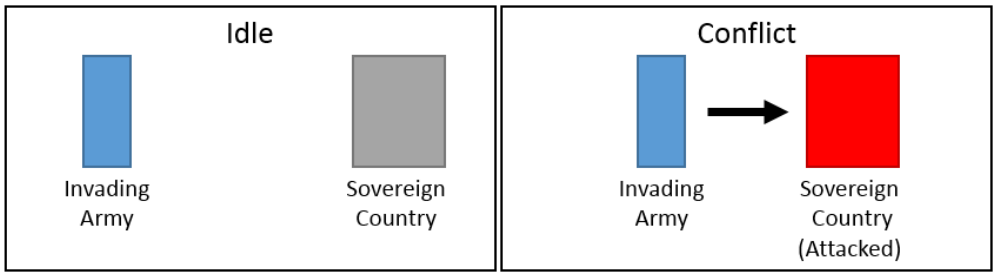

Figure 95 Showing the idle scenario state (left) and trigger state (right)

2. The same as task one, except the sovereign country's allies arrive to its borders, defeating the attacking army, and changing the national colour to green in mourning of those lost in battle. This required two interactions (Figure 96), one for the attacking army, and one for the attacking army and defending army (with the second interaction having higher precedence over the first).

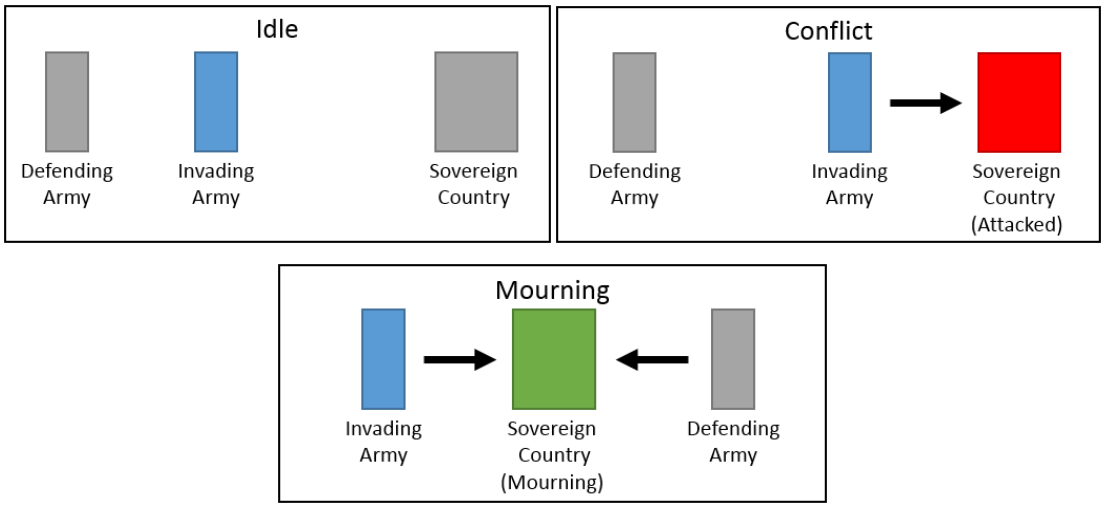

Figure 96 Showing the idle scenario (top-left), first interaction (top-right) and second interaction (bottom)

3. The same as task two, but with two attacking, and two allied/defending armies. No matter which combination reach the country, as long as there is at least one defending army, sovereignty prevails, with the colour only changing to green in mourning if a battle occurs. This required the creation of two groups for the different armies, and two interactions using those groups created (Figure 97).

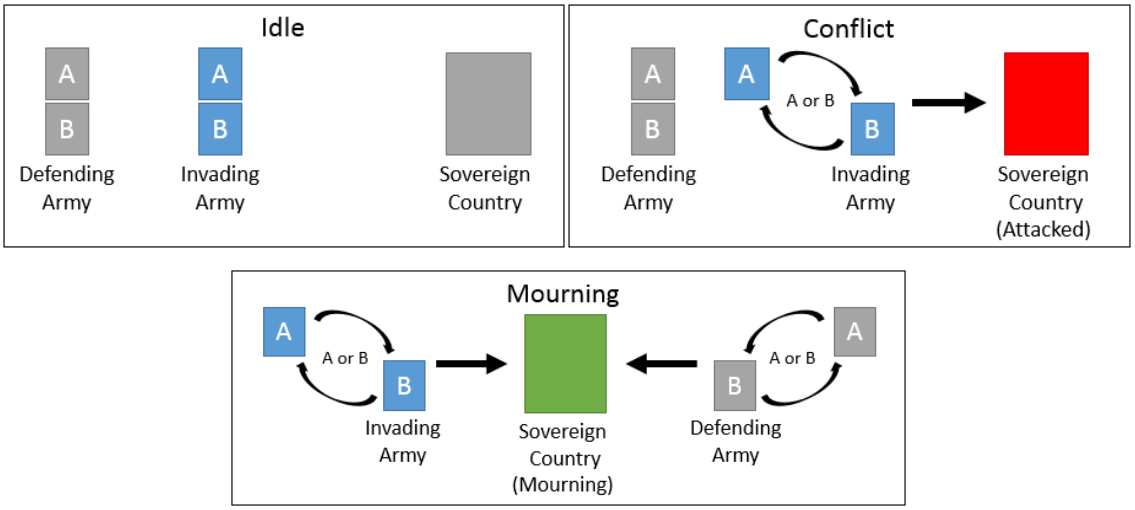

Figure 97 Showing the idle scenario with groups (top-left), first interaction (top-right) and second interaction (bottom)

Participants could not progress to the next task until the previous one was completed. Whilst these scenarios were chosen purely for their pedagogical value, there is a clear application between the evaluation scenarios and possible applications to creating board games, teaching or other interactive systems utilising similar logic.

Given the system utilises gestures to navigate the system, each user was given a reference sheet (see Appendix A – User Study Material) showing each pose and its corresponding function. There was one error in an incorrect image shown for the cancel gesture, however

the error was highlighted when the sheet was given to the participant with no effect on the study. The reference sheet was provided as the study was not interested in evaluating the efficiency of the gestures used in this specific implementation, rather the functionality they linked to. Participants were read aloud a description of each task, before being given a printed copy. Participants were filmed from above when using the system, whilst the supervisor recorded notes regarding actions of interest.

After the study, participants filled out a questionnaire (see Appendix A – User Study Material) focusing on the system's intuitiveness and perceived ease of use. The questionnaire utilised a mix of visual analogue scales and qualitative questions. The study concluded with an informal discussion about their experiences and thoughts using the system. Participants were recruited via email within the university and available social networks. Each participant received a chocolate bar as a token of appreciation for their effort.

7.2.3 Results

The study conducted consisted of 23 participants (2 female, 21 male, mean age of 24, standard deviation (SD) of 3.19), nine of which had prior experience with tangible UIs and nine of which did not have any computer science background. The results of two participants were excluded due to one having severe language difficulties given their use of English as a second language, and the other as the system repeatedly malfunctioned mid-way through their run (both of which had a background in computer science), making 21 participants in total (2 female, 19 male). Perhaps the most important result from the study from was that all 21 participants completed all three test tasks/scenarios, indicating the structure of problems as required by the system represents an achievable goal for end users. The success of participants for the three tasks is as follows.

7.2.3.1 Task 1

All participants completed the first scenario on their first attempt at creating an interaction. Four participants created groups for the two objects used in the task (mean of 0.19 groups/participant, SD = 0.40), however this was unnecessary as the scenario did not require them.

7.2.3.2 Task 2

Sixteen participants completed the second task on their first interaction attempt, with four completing it on their second, and the last two taking three attempts (mean of 1.28 attempts/participant, SD = 0.56). Six participants (29%) unnecessarily utilised groups in

the scenario (mean = 0.33, SD = 0.58), under the belief that groups were needed for interactions involving more than two objects.

7.2.3.3 Task 3

For the final task, fifteen participants succeeded with their first attempt at creating the interaction attempt, with five participants taking two attempts, and one taking three (mean = 1.33, SD = 0.57). A mean of 2.38 groups (SD = 1.32) were created per participant, with the goal being two (one for each the invading and allied armies).

7.2.3.4 Overall

Table 5 shows the mean number of interactions and groups created for each task. The fact that the vast majority of participants succeeded in creating the desired interaction on their first attempt is important. It indicates that the participants' personal expectations about how the system should function matched its actual functionality, both in terms of the participants having to adapt their internal mental representation of the problem to match the system's structure, and then translating that knowledge into the system.

	Mean/SD Interactions	Ideal Number of Interactions	Mean Groups/SD	Ideal Number of Groups
Task 1	1.00/0	1	0.19/0.40	0
Task 2	1.28/0.56	1	0.33/0.58	0
Task 3	1.33/0.57	1	2.38/1.32	2

Table 5 Average attempts per task

To evaluate the intuitiveness, difficulty, responsiveness, guidance and logical progression of the system, participants were asked for feedback by placing marks anywhere on a horizontal line with opposing values placed at each end of the line (e.g. Intuitive versus Unintuitive). By the participant placing a mark along the line, fine-grained values could be obtained (versus using a set scale). This was converted to a 0-100 range, with ratings classified as favourable if below 50. All participants reported that the system progressed in a similar order to how they would have described the scenario to another person, giving a mean rating of 11.99 (SD 0.12) on a scale between logical (0) and illogical (100). They also reported that it matched how they thought through the problem mentally, with a slightly lower mean of 13.49 (SD 0.15) on the same scale. One participant in particular noted that it was easier to work through the scenario using the system physically instead of mentally, with another noting "I can't think of a more suitable way". Another participant highlighted the flexibility offered by the system workflow as a benefit.

However, one participant did say that they felt the user required more steps than if they were describing to another person.

In the questionnaire participants reported that the process of introducing objects was both easy and intuitive to perform, with all participants responding favourably and 19 responding favourably regarding ease of use (mean 23.02, SD 0.19, with 0 easy to use and 100 difficult). For grouping objects, 19 participants gave a favourable rating (mean 19.29, SD 0.18, with 0 being intuitive and 100 unintuitive) for intuitiveness and 16 giving favourable results for ease of group creation (mean 28.49, SD 0.23, with 0 easy to use and 100 difficult). All participants acknowledged understanding the goal of the grouping functionality at the end of the study. In defining logic for the interactions, participants reported positive feedback for both the intuitiveness (mean 23.25, SD 0.20) and ease of creation (mean 25.64, SD 0.20, with 0 being intuitive and 100 unintuitive).

In feedback for the implementation of the system, the intuitiveness of the gestures used had a mean of 26.35 (SD 0.21, with 0 being intuitive and 100 unintuitive), with the ease of creating those gestures receiving a mean of 19.05 (SD 0.17, with 0 being easy and 100 difficult). The use of gestures is further highlighted in the Observations section below.

Participants felt the system was less responsive, with a mean of 38.89 (SD 0.24) on a scale of 0 (responsive) to 100 (unresponsive), however participants reported that the guidance provided by the system using text and TTS instructions was appropriate for such a system.

7.2.3.5 Observations

The most problematic component of the system were the tracking systems, specifically the use of the Kinect for skeleton tracking, which varied greatly between different participants. Almost all participants addressed the skeleton tracking directly in their feedback. One participant in particular was wearing a flight suit, which caused severe problems for the gesture tracking, requiring the user to repeatedly perform a calibration pose so the system could reset and start tracking again. This was compounded by the fact that the participant had recently had shoulder surgery, making gestures far from ideal. As a result, it is thought that the tracking issues affected the rating participants gave regarding system responsiveness. As mentioned prior, given the focus of this investigation on enabling ad-hoc interaction at a low level, and not primarily on the tracking systems, these problems are expected and can be addressed in future revisions of the implementation. Despite these issues, the vast majority of users liked the system.

One interesting observation into the ad-hoc nature of the interactions was how participants customised each example with different naming of the objects (e.g. NATO as the defending force). This implied users were making wider cognitive connections between the objects and the context within which they were thinking the scenario mentally. Users often started the tasks using short names, progressing to entering longer ones as their confidence using the projected keyboard grew.

7.2.4 Impact on System Design

Whilst the feedback from the study showed the system and rule structure as being effective for supporting the creation of these systems, participants noted that the system should take into account more information about the scene, supporting the results of the original preliminary study. For example, if the surface is blank, and the user places something in the middle of it, and then pauses, the system can safely assume they want to perform some operation with that object. Conversely, if there are multiple objects on the table and the user pushes everything to the outer-rim, clearly they are preparing for a coming operation. As such, the system was modified to support a single 'interaction area' that bound the main area (accessible within seated arms-reach). This was then used to display relevant navigation options to the user, e.g. defining an interaction or group if multiple objects are in the interaction area. This approach was used by Khalilbeigi et al. (2012) to show more information for objects based on their proximity.

7.3 Discussion

Aside from the user's interactions being limited to the proxemic interactions using the ProximityAction, the implementation is limited in support for temporal interactions, both as input and output. However, there is no reason why the internal logic in Action could not take into account a temporal element. The only limiting factor for regarding time for the current design is that outputs cannot be staged over time using a PropertyMap, as either the PropertyMap is applied, or not. However, to achieve temporal-based outputs, the values of given Properties involved in the PropertyMap could be changed over time. This would mean that despite the application of a PropertyMap of one Property to another being nullary, the values being used within those Properties could be modified over time by the Action. Such functionality is explored using the continuous valuator-like controls (such as the slider and dial) presented in previous chapters.

In describing future tangible systems, Ishii (2008) identified this problem in tangible systems where to enable more generic level of support, the tangible themselves are

abstracted to high and higher levels, resulting in the tangibles losing their individual affordances, essentially becoming generic 'pucks'. However, whilst the implementation of this architecture was limited in its support for different tangible inputs and outcomes, the core architecture developed is not, as shown in the previous chapters enabling deeper affordance-based functionality. Any number of different Properties can be created and associated with objects, representing any number of unique affordances of those objects. As a result, now that there is a way to represent and store such attributes in the architecture, there needs to be a way to support the sensing and UI side of the problem, where these affordances need to be able to be identified and sensed, whilst not overwhelming the user with infinite options about how the object could be used. As Foley et al. (1984) noted that in the physical world, any number of interactions can be performed with an object (e.g. squeeze, stroke, push, etc.), and thus there needs to be an intuitive UI and sensing system for these inputs to be identified, sensed and mapped by corresponding functionality. Given the architecture to represent these systems now exists, future focus can begin to explore the other half of the problem, with the primary constraint being the passive monitoring of the arbitrary states of different objects used within the system.

The results of the evaluation definitively showed users without any prior experience can create interactive, tangible ad-hoc systems with very little training, converting abstract interactive systems into physical ones. Users are able to grasp the concepts required for the TAM architecture, being able to externalise and translate a number of scenarios into a form understandable by the system. Participants using the system reported a strong level of cohesion between their mental representation of the system and how it needs to be translated to match the system architecture, with the results proving true even for participants without a technical background.

Chapter 8. Discussion

Previous chapters have described a generic ad-hoc tangible architecture (TAM) and three example implementations to support the user in creating ad-hoc iterations, involving the creations of system UI controls (AH-UI), data (AH-Data) and logic (AD-Logic) manipulation. This chapter frames that work in a larger context when compared to existing work and the larger goals of this research.

8.1 Existing Tangible Patch Panels

As highlighted in Chapter 2, a previous tangible patch panel was developed as part of the iStuff framework (Borchers et al., 2002, Ballagas et al., 2003) that enabled the mapping of a number of dedicated, purpose-built active input devices to software functions. The system used events generated by active electronic objects to create a heap of tuple space events to be processed by the system. To map the input of one device to a function, the user had to first create a custom intermediary event type, manually performing the mapping between the data provided from the input event and the destination function using code. Given each input device generates its own event type, custom mappings had to be written/developed for every interaction the user wanted to perform. Essentially, device-specific events were generated and had to be manually transformed to pass data to an external function, with the generation of the mappings done using a GUI on a PC.

Despite the focus of iStuff being on flexible interactions for ubiquitous computing in the "post-desktop era" (as the author states), the resulting system still required a desktop PC to be usable, removing the user from the ubiquitous context whenever they want to perform a new task. My investigation has focused on the automatic mapping of novel input controls to outputs, whilst still allowing manually mappings to be performed for more advanced interactions and user by passing data an external application (where additional logic resides) and back in again in a feedback loop. The TAM approach not only lowers the threshold for development, but increases the speed at which interactions can be created at runtime. In addition, the iStuff framework (Ballagas et al., 2003) only utilised dedicated physical input controls with embedded electronics. To create new controls from scratch, the physical and associated electronics must first be developed, before an associated device-specific event created and compiled into the system, not only taking the user out of context, but out of the system entirely. Conversely, this approach allows user to create novel input controls on-the-fly from passive materials (or active

given appropriate mappings) and map them to existing functionality without leaving the current context, or interacting in any modality outside the one they are already using.

8.2 Level of Tangible Support

Previous tangible systems have sought to support the user in specific use cases, allowing them to leverage a specific set of functionality to achieve desired outcomes. However, as computing systems have increasingly become commodities, so too has the associated functionality. Generic software suites offer an endless set of functions, each adaptable to different use cases as required. Given previous systems have demonstrated the clear benefits of tangible interactions, there has yet been no generalised, extensible approach to tangible interaction – tangible systems have not matured at the same rate as their software-based counterparts. This is especially surprising given the hardware required for such systems has also become commodity. Depth camera systems that used to cost tens of thousands of dollars and now present in people's living rooms. Increasingly powerful mobile systems with various arrays of sensors are being carried around by us with almost religious dedication. As such, the lack of support for generalised tangible interactions is surprising given hardware support is available.

In order to support generic interactions, the software must be able to model and represent extensibility, even before the human interaction methods used for creating such content/functionality are taken into account. Without the underlying ability to represent the world, the UI has nothing to control. Given previous work has sought to classify the types of interactions possible using TUIs, it is best to evaluate the proposed architecture and implemented systems within that work. Ullmer and Ishii (2000) identified three categories of physical objects; spatial, relational, and constructive (later defined as TACs). The TAM architecture supports all three categories, allowing for Actions to be defined using the spatial or relational properties within or between objects and touch, allowing for these properties to be both used as the primary means of interacting with the system (e.g. defining the logic for simple systems), or in passing the resulting values to external systems (using various UI controls or data manipulation). In addition to this, constructive assemblies are possible where the user uses multiple objects in unison to achieve a shared goal. Despite controls such as the improvised joystick involving multiple objects, the TAM architecture makes no distinction between single objects or constructive assemblies. Rather, TAM only cares about the desired properties of the resulting control. Whilst not implemented in any of the example systems, interactive surfaces could easily

be support by passing scene data to the Action class, allowing for the generation of topological systems.

The four TUI categories identified by Klemmer et al. (2004) (spatial, topological, associative and forms) are also supported:

- Interactions with objects in the Cartesian plane are supported (spatial).
- Relationships between objects can be used as the means of interaction (topological).
- Physical objects and interactions can be associated with digital content (associative).
- With the addition of passing scene data into the Action class, interactions on paper can be sensed and associated with application functionality (forms). Whilst not currently implemented, all that would be required would be to pass RGBD data into the Action class as part of the evaluation each frame.

When these four categories are matched back against the iterations and focus of development presented in this dissertation, there is a clear mapping:

- manipulating the UI was *spatial* (AH-UI),
- manipulating the data was *associative* (AH-Data), and
- manipulating logic was *topological* (AH-Logic).

The coupling between input and output is dependent on the user and the associated system controls. Richer input controls utilising embodied controls are no different to any other control. Given the 'output' VirtualProperties can be treated as normal Properties and associated with physical InteractionObjects (which may use these properties to render state information), tighter feedback loops can be created. In addition to the ability for the system to store/restore previous Property states (as used for toggled interactions such as in the third implementation), the system can also support epistemic interactions by the user. Once the user tries something with an undesired outcome, the ad-hoc system can simply restore the previous state, before the user resumes pragmatic interactions. This rapid cycling between exploring application functionality and task progression support the key premise of supporting Action Regulation Theory as defined by Hacker (1994).

Tangible controls used in the system can be assigned multiple roles, either within logic-based interactions or interpreting different physical interactions with different functions, supporting the full range of TUI equivalencies identified by Ullmer and Ishii (1997). Instead of assigning multiple functions to individual objects, multiple objects may also

be used, allowing TAM to support both time and space-multiplexed (Ullmer, 2002) interactions and allow the user to define an appropriate DOF (Sharlin et al., 2004) and degree of coherence (Koleva et al., 2003) between the object and the application. The ability for external systems to provide triggers and associated Property values to the system allows the third party integration of new types of input controls without having to author any controls. Indeed active organic controls such as digital foam (Smith et al., 2008) allows for the integration of OUIs (Holman and Vertegaal, 2008).

Hornecker and Buur (2006) highlighted three main concepts for tangibles, identifying their representational significance, externalisation, and perceived coupling. The level to which TAM-based systems support these concepts is primarily reliant on how the end uses decides to create and leverage their tangible artefacts. However, the ability for systems based on TAM to allow the user to define roles for objects used, walkthrough problem using them and then have some direct outcome (within or external to the system) is supported, and has already been demonstrated in the sample implementations.

All seven categories of 3D widgets (3D object selection, 3D object manipulation, 3D scene control, 2D document visualization, discrete valuators, continuous valuators, and menu selection), as identified in the taxonomy by Dachselt and Hinz (2005) are supported. Menu selection is supported using the tangible menu equivalent to GUI menus as identified by Ullmer and Ishii (1997). The weakest support however comes from document visualisation, however document display and navigation can easily still be supported, similar to how the video preview in the video editor sample application was done.

The development and abstraction of sensing different types of physical interactions and inputs by the user using the Action class is similar to the recently published WorldKit system's "interactors", which are also inheritable and extensible for sensing different types of feedback, however their development did not involve or support the integration of external systems. Instead, they just represented a standardised way for developers to add new sensing capabilities within a fixed-capability system, without ensuring that those capabilities can be automatically supported by existing functionality. However, the development of WorldKit helps to demonstrate the need for ubiquitous tangible ad-hoc interaction.

Previously theoretical interactions described in Opportunistic Controls as using the physical attributes of the environment as ad-hoc user controls can now be realised. Unlike

the prior-authoring required by the OCs system, TAM allows the ad-hoc incorporation of novel both touch and tangible based controls based on the affordances of the user's current environment.

8.3 Applications

Outside of lowering the barrier for users to create tangible UIs from scratch whilst allowing them to extend previously fixed systems to suit their individual needs, the TAM architecture also has a number of other applications as described in this section. Given TAM allows for the simple representation of different objects, properties, and external functions, the implemented versions can be used by developers as a framework to create tangible applications by itself. Even if the framework was not used as to enable ad-hoc extensibility, it can be used to provide structural separation for dedicated tangible applications, as defined by the MVC/MCRit.

TAM can also be used to help design systems where a known set of functions exist, but the methods of interactions used for them are unknown (e.g. prototyping). Both physical controls built from passive components as well as touch and active-integrated controls can all be integrated at runtime, with roles and functionality modified as needed without having to author any code outside of the initial mappings. 3D printed controls and improvised input devices can be functionally equal to purpose built controls, giving flexibility to non-technical designers.

Aside from developing functional prototypes, objects can be incorporated to fulfil ad-hoc roles. The Illuminating Light system could be created using TAM, providing an extensible framework where new types of optical controls could be defined. Using a purpose-built Action class that renders the light path, the groups and Properties associated with different objects (that are defined at run time) are used by the Action to adjust the path of the light. For example the severity of a concave/convex lens might be an associated IntegerProperty that can be adjusted by the user at runtime. Users can create new types of input controls in the system, without having to author any code.

Similarly, systems such as WorldKit can be produced. Whilst WorldKit had a similar goal for ad-hoc interactions, the focus was on the creation of controls, and not the integration with external systems. Every control supported by WorldKit can be created within TAM, allowing for the dynamic support of external systems. In addition, various example applications from previous systems could easily be created using TAM and the example implementations shown, such as the use of VoodooIO to create a sound mixing board

(Villar and Gellersen, 2007) or the creating of an AR chemistry set (Fjeld and Voegtli, 2002).

Whilst not the core function, a suitable application would be as an extendable function to a modern version of Wellner's digital desk. Despite the core logic already present, it can be expected that at some point the user will seek to utilise unfamiliar objects with the system. TAM would allow this kind of additional interaction, allowing new objects and functions to become part of the existing system, without the user having to involve the original developer.

Systems such as URP (Underkoffler and Ishii, 1999) could also benefit from an ad-hoc constraint based system. Whilst presenting my work I was approached by someone that had worked with architects and mentioned that a system such as this would allow them to quickly introduce models into the system and apply constraints based on the items. For example, they might need to ensure that the sewerage plant is more than two units (kilometres) from any domestic building. The user can create a group for domestic buildings, then create a single rule to ensure they can move the items on the table and be reminded when they violate planning rules. The user can then introduce power sub-stations and create a rule saying that the all industrial plants must be within 1km of a power station. By adding more rules, the user can intuitively explore different configuration options tangibly. When merging TAM functionality with the previous URP system, users can extend the pre-built functionality, adding a tangible sun for controlling time of day shadows, etc. Arguably, instead of the system just seeing generic buildings/objects, the system can begin to understand their individual types, and the relationships between those types.

Previous tangible systems could be reproduced using TAM. The chemistry example used throughout has been the focus or an application of multiple tangible research projects (such as (Patten et al., 2001) and (Fjeld and Voegtli, 2002)), as well as commercial projects[20]. Data-focused systems such as MediaBlocks and Tangible Tiles could be created from scratch by users, allowing the physical manipulation of intangible data and attributes. A more flexible and extensible version of Peripheral AR (Edge, 2008) can be created, with the user capable of creating new peripheral controls for required functions using appropriate input modalities as needed. In addition, such controls could be

[20] http://daqri.com/press_posts/fastcompany-daqris-4d-cubes-use-augmented-reality-to-teach-kids-the-periodic-table-of-elements/

incorporated into existing systems that utilise a variable number of controls, such as with Tangible Query Interfaces (Ullmer et al., 2003).

8.4 Summary

TAM supports the full, comprehensive range of tangible controls when evaluated against the different categories and types of TUIs as identified in previous work. In addition to supporting the full range of methods of tangible interaction, the ability for feedback loops incorporating external logic enables the incorporation of and creation of organic and embodied UIs. The ability for states of objects to be preserved by keeping track of the history of Property objects affords the user a low cost of epistemic exploration, as required by Action Regulation Theory. The use of the TAM architecture in the example implementations enables a number of previous tangible systems to be created ad-hoc by the user (as highlighted in previous chapters), whilst adding flexibility and extensibility to them. In addition to the dedicated systems and examples that have been developed to demonstrate functionality, there is clear application for the work presented in this dissertation to enable extensibility within existing research systems, representing a step towards enabling the generic tangible interactions required to support the vision of the Digital Desk and Luminuous Room projects.

Chapter 9. Conclusion

This dissertation has focused on enabling end users to extend tangible systems without requiring them to leave their current tangible context. The TAM architecture that has been developed successfully demonstrates the ability to create extendible, ad-hoc tangible systems involving application UI control, data manipulation and logic creation by end users. The implementations have demonstrated that interfaces to such systems do not have to be complex, allowing users with little or no previous knowledge to author new system content. Working towards the same pervasive, digitally assisted existence that the Digital Desk began exploring over 20 years ago, the TAM architecture represents a step towards the realisation of that capability. I look forward to continuing to develop digital support for pervasive, ad-hoc interactions, continuing to merge the real and digital realms.

9.1 Contribution

This dissertation presented a number of distinct contributions to the field of tangible user interfaces and adaptable interaction. The TAM architecture developed supports both the original system developers and end users in developing and using extensible, tangible ad-hoc systems. The research explored three distinct facets of such interactions, in supporting the creation of tangible ad-hoc user interfaces (AH-UI), manipulation of external application data (AH-Data), and the manipulation of the application logic (AH-Logic), providing an extensible architecture to support each, with a fully functional implementation of that architecture.

The contributions presented in this dissertation are summarised as the following:

- An extensible architecture to support generalised ad-hoc interactions, identifying the isolation and encapsulation of key components and their roles as they relate to the larger requirements.
- A working system supporting the run-time creation-of and interaction-with physical ad-hoc controls from passive materials.
- Individual implementations of the architecture demonstrating different aspects of the functionality across the manipulation of functions, data, and logic.
- Evaluations of ad-hoc systems by end users, showing that end users are capable of effectively using such systems.

This final chapter provides a summary of the contributions, describing limitations of the work, concluding with future directions for the work.

9.1.1 Requirements for Tangible Ad-hoc Systems

With TAM, the threshold for developing TUIs has been reduced. TUI developers can benefit from the architecture and example implementations that reduce the overhead required in creating TUIs, whilst the ad-hoc implementations of the architecture enable novice end-users to develop comprehensive, functional and non-trivial TUIs, without prior training. This indicates that tangible ad-hoc PBD holds much promise for future efforts in allowing end users to create and extend the capabilities of existing systems

9.1.2 Architecture to Support Ad-Hoc Interaction

The TAM architecture provides a flexible system designed to not only support the development of systems across the full spectrum of TUI categories from the developer's perspective, but also support the run-time extension and reconfiguration of that system.

Using a small number of theoretical system components, previously unknown interactive systems can be created by end users. The system is described using any number of hierarchically-defined InteractionObjects, each described by any number of hierarchically-defined Properties, each storing associated information. Interactivity is supported through an Action class, evaluating all known information about the current environment and InteractionObjects. InteractionObjects part of multiple hierarchies can be resolved by the Action class to allow for generic interactions based on 'groups/types' of objects. Interactivity is supported through a patch panel mapping, where values from individual Property instances are assigned to other Properties, using a number of individual PropertyMappings. Communication with external systems is supported by means of VirtualProperties and VirtualActions, where data from external systems is disseminated as if natively generated by those components. This allows external systems to be integrated without authoring any code in the tangible system or recompiling, whilst also not requiring the use of any system-specific libraries in the external system – just launching an application, using a network socket, or IPC. New input modality triggers and output modalities can be added post-development, with little effort, with such functionality appearing as if it were native to the ad-hoc system, and thus transparent to the end user. Aside from using the virtual instances, additional native application functionality can be incorporated by simply extending the core Property and Action classes.

9.1.3 Implementation of an Ad-hoc Architecture

The implementation of the TAM architecture has been as three distinct systems, each focusing on a different aspect of ad-hoc control.

9.1.3.1 Controls (AH-UI)

The AH-UI implementation allowed the creation of user controls for existing external application functionality, i.e. the user knows what the system can do, but wants to define how to do it. System navigation was supported by means of a context-sensitive control button, as suggested from the preliminary study. After pressing the button, the user can select a function to control (after first selecting the group with which the function was associated), with functions defined in an external XML file. Depending on the parameters required for the selected function, different Action types are applicable, with the different input types presented to the user. Upon selecting one, they are guided through creating the control, using touch or tangible interaction. Using such an approach allows passive input controls (arbitrary objects, 3D printed controls or improvised constructive controls) to provide the same functionality as dedicated controls, but without requiring any electronics or coding. Whilst no formal evaluation was performed, the demonstrated functionality offered and the ability to create controls in seconds, providing a number of direct advantages when compared to previous systems in its own right. A public demonstration of the system was very well received, with participants showing appreciation for the system's goal and seeing wide applications for the technology in their own work.

9.1.3.2 Data (AH-Data)

The incorporation of data from an external system allows users to leverage spatial intelligence and skills when performing interactions on collections of data. Aside from being able to create more involved tangible systems outside of just simple valuators, users can group and stack multiple data objects simultaneously, creating a spatially multiplexed UI compared to the previous temporal one where they could only interact with one piece of data at a time using the mouse. The extension of the patch panel to support functions accepting data objects as parameters creates a balance between what occurs in the tangible ad-hoc system versus what occurs in the external host application, providing a customisable level of abstraction. However, to ensure simplicity, some compromises may result. For example the photo tagging application required a single function (open) to accept both image and folder data types, which is not currently supported in the implementation. Much like the definition of multiple InteractionObjects to define groups/types, a similar approach could be used in defining data type hierarchies, supporting inheritance and polymorphism. However this would need to be balanced regarding the complexity that would be imposed on the ad-hoc system, versus how complex it would just be to manage the different data types entirely within the host

application. This was the case with the photo tagger, where a single data type was used, and objects then identified by type by the external host application.

9.1.3.3 Logic (AH-Logic)

The AH-Logic implementation allowed novel interactive systems to be created based on rule-based logic involving the proxemic relationships of objects. Cognitive systems can quickly be externalised with little effort required by the end user, without having to author any content or interact outside their current mode of interaction. Whilst it was self-evident that it was possible to model interactive systems using such an approach, a user study was conducted to verify the applicability of the TAM implementation and approach for end users. Results showed that with very little training, users are able to convert interactive systems they have stored as mental models, and transform them into interactive tangible systems without having to do any programming. Feedback from the participants showed they found the system to be intuitive to use, supporting them in describing the scenarios similar to how they would ideally describe it to another person. Feedback was positive, and whilst participants appreciated the goal of using gestures to support natural system navigation, current tracking systems leave room for improvement.

9.1.3.4 Discussion

Across all three iterations, the implementation of the TAM architecture reduced the overhead required for developing tangible systems, both for developing fixed functions and for the ad-hoc functionality, when compared to the existing method of generating systems from scratch (at best using sensor abstraction such as the Proximity Toolkit). In using TAM for tangible system design, designers can still develop dedicated systems as they do now, however with the added benefit that external functionality and new interaction modalities can be incorporated as the specific uses require. Whilst large parts of dedicated systems will remain fixed, it does not mean that the related/encompassing function logic, UI, and data accessibility cannot be extensible if required.

9.2 Future Work

The investigations in this dissertation have focused primarily on the creation of systems where users solely interact on a tabletop. This is a step up compared with previous systems, such as Light Widgets, given the user can remain in the tangible context, controls can be authored on the table before being moved anywhere else within the tracked volume. However, it would be advantageous to allow the user to create system components and interactions regardless of their location and the surrounding surfaces.

The system supports the Digital Desk-style use case, but it now needs to support more pervasive use, such in the Luminous Room.

Collaborative scenarios where multiple users are collocated around a single interaction area also creates issues. The implementations presented in this dissertation have supported multi-touch interaction, however the supporting UI would need to be adapted to remove any assumptions regarding the user's location or orientation, as well as exploring how to support the collaborative authoring of ad-hoc content. Whilst prior work exists that could aid this transition, e.g. using occlusion sensitive menu layouts (Brandl et al., 2009, Leithinger and Haller, 2007), further work is required to fully adapt each step of the process to a mobile, collaborative context.

These issues however focus more on the implementation of the system, than the underlying TAM architecture. As identified in the discussion chapter, whilst theoretically supported, interactive surfaces remains a type of TUI not yet explored using TAM. Future work includes developing an interactive surface system to identify and resolve any unexpected issues in the architecture and the associated implementation.

References

ABRAMS, M., PHANOURIOU, C., BATONGBACAL, A. L., WILLIAMS, S. M. & SHUSTER, J. E. 1999. UIML: an appliance-independent XML user interface language. *Computer Networks,* 31**,** 1695-1708.

AKAOKA, E., GINN, T. & VERTEGAAL, R. 2010. DisplayObjects: prototyping functional physical interfaces on 3D styrofoam, paper or cardboard models. *Proceedings of the fourth conference Tangible, Embedded, and Embodied Interaction (TEI 2010).* Cambridge, Massachusetts: ACM.

AVRAHAMI, D. & HUDSON, S. E. 2002. Forming interactivity: a tool for rapid prototyping of physical interactive products. *Proceedings 4th conference on Designing Interactive Systems: Processes, Practices, mMethods, and Techniques.* London, England: ACM

BALLAGAS, R., RINGEL, M., STONE, M. & BORCHERS, J. 2003. iStuff: a physical user interface toolkit for ubiquitous computing environments. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI 2003).* Fort Lauderdale, Florida, USA: ACM.

BALLAGAS, R., SZYBALSKI, A. & FOX, A. 2004. Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments. *In:* ANDY, S. & ARMANDO, F. (eds.) *Proceedings of the second IEEE international conference on Pervasive Computing and Communications.* Orlando, Florida.

BIMBER, O. & RASKAR, R. 2005. *Spatial Augmented Reality: Merging Real and Virtual Worlds* A K Peters/CRC Press.

BOLT, R. A. 1980. "Put-that-there": Voice and gesture at the graphics interface. *Proceedings of the seventh annual conference on Computer Graphics and Interactive Techniques.* Seattle, Washington, United States: ACM.

BORCHERS, J., RINGEL, M., TYLER, J. & FOX, A. 2002. Interactive Workspaces: A Framework for Physical and Graphical User Interface Prototyping. *IEEE Wireless Communications,* 9**,** 64-69.

BOWMAN, D. A., KRUIJFF, E., LAVIOLA, J. J. & POUPYREV, I. 2001. An Introduction to 3D User Interface Design. *Presence: Teleoperators and Virtual Environments,* 10**,** 96-108.

BRANDL, P., LEITNER, J., SEIFRIED, T., HALLER, M., DORAY, B. & TO, P. 2009. Occlusion-aware menu design for digital tabletops. *Extended Abstracts on Human Factors in Computing Systems (CHI 2009).* Boston, Massachusetts: ACM.

BROECKER, M. 2013. *Rendering Techniques for Spatial Augmented Reality.* Doctor of Philosophy, University of South Australia.

BROOKS, R. A. 1997. The Intelligent Room project. *Proceedings of the second international conference on Cognitive Technology, Humanizing the Information Age.* Aizu-Wakamatsu City: IEEE.

CARD, S. K., MORAN, T. P. & NEWELL, A. 1983. The Human Information Processor. *The Psychology of Human-Computer Interaction.* New Jersey: L. Erlbaum Associates Inc. Hillsdale.

CHENG, K.-Y., LIANG, R.-H., CHEN, B.-Y., LAING, R.-H. & KUO, S.-Y. 2010. iCon: utilizing everyday objects as additional, auxiliary and instant tabletop controllers. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI 210).* Atlanta, Georgia: ACM.

CONNER, B. D., SNIBBE, S. S., HERNDON, K. P., ROBBINS, D. C., ZELEZNIK, R. C. & DAM, A. V. 1992. Three-dimensional widgets. *Proceedings of the 1992 symposium on Interactive 3D Graphics.* Cambridge, Massachusetts: ACM.

COUTAZ, J. 2007. Meta-user interfaces for ambient spaces. *Task Models and Diagrams for Users Interface Design.* Springer Berlin Heidelberg.

COUTAZ, J. & CALVARY, G. 2012. HCI and software engineering for user interface plasticity. *In:* JACKO, J. A. (ed.) *Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, Third Edition.* CRC Press Taylor and Francis Group.

CRAMPTON SMITH, G. 1995. The Hand that Rocks the Cradle. *ID magazine.*

DACHSELT, R. & HINZ, M. 2005. Three-dimensional widgets revisited-towards future standardization. *New directions in 3D user interfaces, Shaker Verlag*, 89-92.

DAVISON, A. J. 2003. Real-time simultaneous localisation and mapping with a single camera. *Proceedings of the ninth IEEE international conference on Computer Vision.* IEEE.

DEEPAK, B. 2001. Dynamic Shader Lamps: Painting on Movable Objects. *In:* RAMESH, R. & HENRY, F. (eds.) *Proceedings IEEE and ACM International Symposium on Augmented Reality (ISMAR 2001).* New York, New York: ACM.

DEY, A. K., HAMID, R., BECKMANN, C., LI, I. & HSU, D. 2004. a CAPpella: programming by demonstration of context-aware applications. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI 2004).* Vienna, Austria: ACM.

DORING, T., SYLVESTER, A. & SCHMIDT, A. 2013. A design space for ephemeral user interfaces. *Proceedings of the seventh conference on Tangible, Embedded and Embodied Interaction (TEI 2013).* Barcelona, Spain: ACM.

DOUGLAS, S., DOERRY, E. & NOVICK, D. 1990. Quick: a user-interface design kit for non-programmers. *Proceedings of the 3rd symposium on User Interface Software and Technology (UIST 1990).* Snowbird, Utah: ACM.

EDGE, D. 2008. *Tangible user interfaces for peripheral interaction.* Doctor of Philosophy, University of Cambridge.

FAILS, J. A. & OLSEN, D. 2002. Light widgets: interacting in every-day spaces. *Proceedings of the seventh conference on Intelligent User Interfaces.* San Francisco, California: ACM.

FISHKIN, K. P. 2004. A taxonomy for and analysis of tangible interfaces. *Personal and Ubiquitous Computing,* 8**,** 347-358.

FISHKIN, K. P., MORAN, T. P. & HARRISON, B. L. 1999. Embodied User Interfaces: Towards Invisible User Interfaces. *Proceedings of the 7th working conference on Engineering for Human-Computer Interaction.* Deventer, Netherlands: Springer.

FITZMAURICE, G. W., ISHII, H. & BUXTON, W. A. S. 1995. Bricks: laying the foundations for graspable user interfaces. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI 1995).* Denver, Colorado: ACM/Addison-Wesley Publishing.

FJELD, M., BICHSEL, M. & RAUTERBERG, G. 1999. BUILD-IT: a brick-based tool for direct interaction. *In:* HARRIS, D. (ed.) *Job design, product design and human-computer interaction.* Aldershot, UK: Ashgate.

FJELD, M., BICHSEL, M. & RAUTERBERG, M. 1998. BUILD-IT: An intuitive design tool based on direct object manipulation. *In:* WACHSMUTH, I. & FRÖHLICH, M. (eds.) *Gesture and Sign Language in Human-Computer Interaction.* Bielefeld, Germany: Springer Berlin/Heidelberg.

FJELD, M., LAUCHE, K., BICHSEL, M., VOORHORST, F., KRUEGER, H. & RAUTERBERG, M. 2002. Physical and Virtual Tools: Activity Theory Applied to the Design of Groupware. *Computer Supported Cooperative Work (CSCW),* 11**,** 153-180.

FJELD, M. & VOEGTLI, B. M. 2002. Augmented Chemistry: An Interactive Educational Workbench. *Proceedings of the first International Symposium on Mixed and Augmented Reality.* IEEE Computer Society.

FOLEY, J. D., WALLACE, V. L. & CHAN, P. 1984. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications,* 4**,** 13-48.

FRANK, M. R. 1995. Grizzly Bear: a demonstrational learning tool for a user interface specification language. *Proceedings of the 8th symposium on User Interface and Software Technology (UIST 1995).* Pittsburgh, Pennsylvania: ACM.

FRANK, M. R., SUKAVIRIYA, P. N. & FOLEY, J. D. 1995. Inference bear: designing interactive interfaces through before and after snapshots. *Proceedings of the 1st conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques.* Ann Arbor, Michigan: ACM.

FUKUCHI, K., IGARASHI, T., SUGIMOTO, M., FERNANDO, C. & INAMI, M. 2009. Push-pins: Design-by-user approach to home automation programming. *Proceedings of International Joint Conference on Pervasive and Ubiquitous Computing (UBICOMP 2009).* Orlando, Florida: ACM.

GREENBERG, S. & BOYLE, M. 2002. Customizable physical interfaces for interacting with conventional applications. *Proceedings of the 15th symposium on User Interface Software and Technology (UIST 2002).* Paris, France: ACM.

GREENBERG, S. & FITCHETT, C. 2001. Phidgets: easy development of physical interfaces through physical widgets. *Proceedings of the 14th symposium on User Interface Software and Technology.* Orlando, Florida: ACM.

GROSSMAN, T. & WIGDOR, D. 2007. Going Deeper: a Taxonomy of 3D on the Tabletop. *Proceedings of the 2nd international workshop on Horizontal Interactive Human-Computer Systems (TABLETOP 2007).* Newport, Rhode Island: IEEE.

GRUDIN, J. 1994. Computer-supported cooperative work: history and focus. *Computer,* 27**,** 19-26.

GUPTA, A., FOX, D., CURLESS, B. & COHEN, M. 2012. DuploTrack: a real-time system for authoring and guiding duplo block assembly. *Proceedings of the 25th symposium on User Interface Software and Technology (UIST 2012).* Cambridge, Massachusetts ACM.

HACKER, W. 1994. Action regulation theory and occupational psychology: Review of German empirical research since 1987. *German Journal of Psychology,* 18**,** 91-120.

HALBERT, D. C. 1984. *Programming by example.* Doctor of Philosophy, University of California.

HARDY, J. & ALEXANDER, J. 2012. Toolkit support for interactive projected displays. *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia.* Ulm, Germany: ACM.

HENDERSON, S. & FEINER, S. 2010. Opportunistic Tangible User Interfaces for Augmented Reality. *Visualization and Computer Graphics, IEEE Transactions on,* 16**,** 4-16.

HENDERSON, S. J. & FEINER, S. 2008. Opportunistic controls: leveraging natural affordances as tangible user interfaces for augmented reality. *Proceedings of the 2008 ACM symposium on Virtual reality software and technology.* Bordeaux, France: ACM.

HINCKLEY, K. 1996. *Haptic issues for virtual manipulation.* Doctor of Philosophy, University of Virginia.

HINCKLEY, K., PAUSCH, R., GOBLE, J. C. & KASSELL, N. F. 1994. Passive real-world interface props for neurosurgical visualization. *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence.* Boston, Massachusetts: ACM.

HOLMAN, D. & VERTEGAAL, R. 2008. Organic user interfaces: designing computers in any way, shape, or form. *Communications of the ACM,* 51**,** 48-55.

HOLMQUIST, L. E., REDSTRÖM, J. & LJUNGSTRAND, P. 1999. *Token-based access to digital information*, Springer.

HORNECKER, E. & BUUR, J. 2006. Getting a grip on tangible interaction: a framework on physical space and social interaction. *Proceedings of the SIGCHI conference on Human Factors in computing systems.* Montreal, Quebec: ACM.

HUDSON, S. E. & MANKOFF, J. Rapid construction of functioning physical interfaces from cardboard, thumbtacks, tin foil and masking tape. Proceedings of the 19th annual ACM symposium on User interface software and technology, 2006. ACM, 289-298.

HUTCHINS, E. L., HOLLAN, J. D. & NORMAN, D. A. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction,* 1**,** 311-338.

ISHII, H. 2008. Tangible bits: beyond pixels. *Proceedings of the second international conference on Tangible and embedded interaction.* Bonn, Germany: ACM.

ISHII, H., LAKATOS, D., BONANNI, L. & LABRUNE, J.-B. 2012. Radical atoms: beyond tangible bits, toward transformable materials. *Interactions,* 19**,** 38-51.

ISHII, H., RATTI, C., PIPER, B., WANG, Y., BIDERMAN, A. & BEN-JOSEPH, E. 2004. Bringing Clay and Sand into Digital Design — Continuous Tangible user Interfaces. *BT Technology Journal,* 22**,** 287-299.

ISHII, H. & ULLMER, B. 1997. Tangible bits: towards seamless interfaces between people, bits and atoms. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Atlanta, Georgia: ACM.

IZADI, S., NEWCOMBE, R. A., KIM, D., HILLIGES, O., MOLYNEAUX, D., HODGES, S., KOHLI, P., SHOTTON, J., DAVISON, A. J. & FITZGIBBON, A. 2011. KinectFusion: real-time dynamic 3D surface reconstruction and interaction. *ACM SIGGRAPH 2011 Talks.* Vancouver, British Columbia, Canada: ACM.

KALTENBRUNNER, M., BOVERMANN, T., BENCINA, R. & COSTANZA, E. TUIO: A protocol for table-top tangible user interfaces. Proceedings of the the sixth International Workshop on Gesture in Human-Computer Interaction and Simulation, 2005.

KATO, H., BILLINGHURST, M., POUPYREV, I., IMAMOTO, K. & TACHIBANA, K. 2000. Virtual object manipulation on a table-top AR environment. *Proceedings of the IEEE and ACM International Symposium on Augmented Reality.* Munich, Germany: IEEE.

KHALILBEIGI, M., SCHMITTAT, P., MÜHLHÄUSER, M. & STEIMLE, J. 2012. Occlusion-aware interaction techniques for tabletop systems. *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts.* ACM.

KIRSH, D. & MAGLIO, P. 1994. On distinguishing epistemic from pragmatic action. *Cognitive Science,* 18**,** 513-549.

KJELDSEN, R., LEVAS, A. & PINHANEZ, C. 2003. Dynamically Reconfigurable Vision-Based User Interfaces. *In:* CROWLEY, J., PIATER, J., VINCZE, M. & PALETTA, L. (eds.) *Computer Vision Systems.* Springer Berlin/Heidelberg.

KLEMMER, S. R., LI, J., LIN, J. & LANDAY, J. A. 2004. Papier-Mache: toolkit support for tangible input. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Vienna, Austria: ACM.

KOLEVA, B., BENFORD, S., NG, K. H. & RODDEN, T. 2003. A Framework for Tangible User Interfaces. *Proceedings on Real World User Interfaces, Mobile HCI Conference.* Udine, Italy.

LEE, G. A., NELLES, C., BILLINGHURST, M. & KIM, G. J. 2004a. Immersive Authoring of Tangible Augmented Reality Applications. *Proceedings of the third*

IEEE/ACM International Symposium on Mixed and Augmented Reality. Arlington, Virginia: IEEE.

LEE, J. C., AVRAHAMI, D., HUDSON, S. E., FORLIZZI, J., DIETZ, P. H. & LEIGH, D. 2004b. The calder toolkit: wired and wireless components for rapidly prototyping interactive devices. *Proceedings of the fifth conference on Designing interactive systems: processes, practices, methods, and techniques.* Cambridge, Massachusetts: ACM.

LEITHINGER, D. & HALLER, M. 2007. Improving menu interaction for cluttered tabletop setups with user-drawn path menus. *Proceedings of the second annual IEEE International Workshop on Horizontal Interactive Human-Computer Systems,* . Newport, Rhode Island: IEEE.

LI, Y. & LANDAY, J. A. 2005. Informal prototyping of continuous graphical interactions by demonstration. *Proceedings of the 18th annual ACM symposium on User interface software and technology.* Seattle, Washington: ACM.

LIMBOURG, Q. & VANDERDONCKT, J. 2004. Addressing the mapping problem in user interface design with UsiXML. *Proceedings of the 3rd annual conference on Task models and diagrams.* Prague, Czech Republic: ACM.

LUYTEN, K. & CONINX, K. 2001. An XML-Based Runtime User Interface Description Language for Mobile Computing Devices. *In:* JOHNSON, C. (ed.) *Interactive Systems: Design, Specification, and Verification.* Springer Berlin / Heidelberg.

MARNER, M. R. 2013. *Physical-Virtual Tools for Interactive Spatial Augmented Reality.* Doctor of Philosophy, University of South Australia.

MARNER, M. R., SMITH, R. T., WALSH, J. A. & THOMAS, B. H. 2014. Spatial User Interfaces for Large Scale Projector-Based Augmented Reality *Computer Graphics and Applications, IEEE*.

MARNER, M. R., THOMAS, B. H. & SANDOR, C. 2009. Physical-virtual tools for spatial augmented reality user interfaces. *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality.* Orlando, Florida: IEEE.

MARQUARDT, N., DIAZ-MARINO, R., BORING, S. & GREENBERG, S. 2011. The proximity toolkit: prototyping proxemic interactions in ubiquitous computing ecologies. *Proceedings of the 24th annual ACM symposium on User interface software and technology.* Santa Barbara, California: ACM.

MATTHIAS, R. & MORTEN, F. 1998. Task Analysis in Human-Computer Interaction - supporting action regulation theory by simulation. *Zeitschrift Fur Arbeitswissenschaft,* 3**,** 152-161.

MAULSBY, D., GREENBERG, S. & MANDER, R. 1993. Prototyping an intelligent agent through Wizard of Oz. *Proceedings of the conference on Human factors in computing systems (CHI 1993).* Amsterdam, Netherland: ACM.

MAYBURY, M. 1999. Intelligent user interfaces: an introduction. *Proceedings of the 4th international conference on Intelligent user interfaces.* Los Angeles, California: ACM.

MCDANIEL, R. G. 1999. *Building Whole Applications Using Only Programming-by-Demonstration.* Doctor of Philosophy, Carnegie Mellon University.

MCGRENERE, J., BAECKER, R. M. & BOOTH, K. S. 2002. An evaluation of a multiple interface design solution for bloated software. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* Minneapolis, Minnesota: ACM.

MERRILL, D., KALANITHI, J. & MAES, P. 2007. Siftables: towards sensor network user interfaces. *Proceedings of the first international conference on Tangible and embedded interaction.* Baton Rouge, Louisiana: ACM.

MILGRAM, P., TAKEMURA, H., UTSUMI, A. & KISHINO, F. 1995. Augmented reality: a class of displays on the reality-virtuality continuum. *In:* DAS, H. (ed.) *Proceedings of Society of Photographic Instrumentation Engineers.* 1 ed. Boston, Massachusetts: SPIE.

MYERS, B. A. 1986. Visual programming, programming by example, and program visualization: a taxonomy. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Boston, Massachusetts: ACM.

MYERS, B. A. 1992. Demonstrational interfaces: A step beyond direct manipulation. *Computer,* 25**,** 61-73.

MYERS, B. A. 1995. User interface software tools. *ACM Transactions in Computer Human Interacteraction,* 2**,** 64-103.

NEWCOMBE, R. A., IZADI, S., HILLIGES, O., MOLYNEAUX, D., KIM, D., DAVISON, A. J., KOHLI, P., SHOTTON, J., HODGES, S. & FITZGIBBON, A. 2011. KinectFusion: Real-time dense surface mapping and tracking. *Proceedings of the 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2011).* Basel: IEEE.

NORMAN, D. A. 1988. *The psychology of everyday things*, Basic books.

ODA, O., LISTER, L. J., WHITE, S. & FEINER, S. 2007. Developing an augmented reality racing game. *Proceedings of the second international conference on Intelligent Technologies for Interactive Entertainment.* Cancun, Mexico: ICST

(Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

OLSEN, D. R. 1998. Basic Interaction. *In:* PABST, J. (ed.) *Developing user interfaces.* San Francisco: Morgan Kaufmann Publishers.

PALEY, W. B. 1998. Designing special-purpose input devices. *SIGGRAPH Computer Graphics,* 32**,** 55-59.

PARAMYTHIS, A., TOTTER, A. & STEPHANIDIS, C. 2001. A modular approach to the evaluation of Adaptive User Interfaces. *In:* WEIBELZAHL, S., CHIN, D. & WEBER, G. (eds.) *Proceedings of the Workshop on Empirical Evaluations of Adaptive Systems.* Sonthofen, Germany.

PATTEN, J., ISHII, H., HINES, J. & PANGARO, G. 2001. Sensetable: a wireless object tracking platform for tangible user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Seattle, Washington: ACM.

PETERSEN, N. & STRICKER, D. 2009. Continuous natural user interface: Reducing the gap between real and digital world. *Proceedings of the 2009 eighth IEEE International Symposium on Mixed and Augmented Reality.* Orlando, Florida: IEEE.

PIPER, B., RATTI, C. & ISHII, H. 2002. Illuminating clay: a 3-D tangible interface for landscape analysis. *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves.* Minneapolis, Minnesota: ACM.

PORTER, S. R., MARNER, M. R., SMITH, R. T., ZUCCO, J. E. & THOMAS, B. H. 2010. Validating spatial augmented reality for interactive rapid prototyping. *Proceedings of ninth IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2010).* Seoul, Korea: IEEE.

PRADEEP, V., RHEMANN, C., IZADI, S., ZACH, C., BLEYER, M. & BATHICHE, S. 2013. MonoFusion: Real-time 3D reconstruction of small scenes with a single web camera. *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2013).* Adelaide, Australia: IEEE.

RASKAR, R., WELCH, G., CUTTS, M., LAKE, A., STESIN, L. & FUCHS, H. 1998. The office of the future: a unified approach to image-based modeling and spatially immersive displays. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques.* Orange County, California: ACM.

RASKAR, R., WELCH, G., LOW, K.-L. & BANDYOPADHYAY, D. 2001. Shader lamps: Animating real objects with image-based illumination. *Rendering Techniques 2001*. Springer.

REKIMOTO, J. & SAITOH, M. 1999. Augmented surfaces: a spatially continuous work space for hybrid computing environments. *Procceedings of the SIGCHI conference on Human factors in computing systems.* Pittsburgh, Pennsylvania: ACM.

RYOKAI, K., MARTI, S. & ISHII, H. 2004. I/O brush: drawing with everyday objects as ink. *Proceedings of the SIGCHI conference on Human factors in computing systems*. Vienna, Austria: ACM.

SALBER, D. & COUTAZ, J. 1993. Applying the wizard of oz technique to the study of multimodal systems. *Human-Computer Interaction.* Springer.

SCHAFER, K., BRAUER, V. & BRUNS, W. 1997. A new approach to human-computer interaction-synchronous modelling in real and virtual spaces. *Proceedings of the second conference on Designing interactive systems: processes, practices, methods, and techniques.* Amsterdam, Netherlands: ACM.

SCOTT, S. D., CARPENDALE, M. S. T. & HABELSKI, S. 2005. Storage bins: mobile storage for collaborative tabletop displays. *Computer Graphics and Applications, IEEE,* 25**,** 58-65.

SHAER, O. & HORNECKER, E. 2010. Tangible User Interfaces: Past, Present, and Future Directions. *Foundations and Trends Human-Computer Interaction,* 3**,** 1-137.

SHARLIN, E., WATSON, B., KITAMURA, Y., KISHINO, F. & ITOH, Y. 2004. On tangible user interfaces, humans and spatiality. *Personal and Ubiquitous Computing,* 8**,** 338-346.

SHNEIDERMAN, B. 1983. Direct manipulation. *Computer,* 16**,** 57-69.

SIMON, T. M., THOMAS, B. H., SMITH, R. T. & SMITH, M. 2014. Adding input controls and sensors to RFID tags to support dynamic tangible user interfaces. *Proceedings of the 8th International Conference on Tangible, Embedded and Embodied Interaction.* Munich, Germany: ACM.

SMITH, R. T., THOMAS, B. H. & PIEKARSKI, W. 2008. Digital foam interaction techniques for 3D modeling. *Proceedings of the 2008 ACM symposium on Virtual reality software and technology.* Bordeaux, France: ACM.

SPIESSL, W., VILLAR, N., GELLERSEN, H. & SCHMIDT, A. 2007. VoodooFlash: authoring across physical and digital form. *Proceedings of the first international*

conference on Tangible and embedded interaction. Baton Rouge, Louisiana: ACM.

STARY, C. & TOTTER, A. 1997. How to Integrate Concepts for the Design and the Evaluation of Adaptable and Adaptive User Interfaces. *In:* STEPHANIDIS, C. & CARBONELL, N. (eds.) *Proccedings of the 3rd ERCIM Workshop on User Interfaces for All.* Strasbourg, France.

STUERZLINGER, W., CHAPUIS, O., PHILLIPS, D. & ROUSSEL, N. 2006. User interface facades: towards fully adaptable user interfaces. *Proceedings of the 19th annual ACM symposium on User interface software and technology.* Montreux, Switzerland: ACM.

SUTHERLAND, I. E. 1968. A head-mounted three dimensional display. *Proceedings of the 1968 Fall Joint Computer Conference.* San Francisco, California: ACM.

TANG, A., OWEN, C., BIOCCA, F. & MOU, W. 2003. Comparative effectiveness of augmented reality in object assembly. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Ft. Lauderdale, Florida: ACM.

THEVENIN, D. & COUTAZ, J. 1999. Plasticity of user interfaces: Framework and research agenda. *Proceedings of the conference on Human-computer interaction (INTERACT 1999).* Edinburgh, Scotland: IFIP.

TRAVERS, M. 1994. Recursive interfaces for reactive objects. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Boston, Massachusetts: ACM.

ULLMER, B. & ISHII, H. 1997. The metaDESK: models and prototypes for tangible user interfaces. *Proceedings of the tenth annual ACM symposium on User interface software and technology (UIST 1997).* Banff, Alberta: ACM.

ULLMER, B. & ISHII, H. 2000. Emerging frameworks for tangible user interfaces. *IBM Systems Journal,* 39**,** 915-931.

ULLMER, B., ISHII, H. & GLAS, D. 1998. mediaBlocks: physical containers, transports, and controls for online media. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques.* Orlando, Florida: ACM.

ULLMER, B., ISHII, H. & JACOB, R. J. 2003. Tangible query interfaces: Physically constrained tokens for manipulating database queries. *Proceedsing of the comference on Human-computer Interaction of INTERACT 2003).* Zurich, Switzerland: IFIP.

ULLMER, B. A. 2002. *Tangible interfaces for manipulating aggregates of digital information.* Doctor of Philosophy, Massachusetts Institute of Technology.

UNDERKOFFLER, J. 1997. A view from the Luminous Room. *Personal and Ubiquitous Computing,* 1**,** 49-59.

UNDERKOFFLER, J. & ISHII, H. 1998. Illuminating light: an optical design tool with a luminous-tangible interface. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Los Angeles, California: ACM Press/Addison-Wesley Publishing Co.

UNDERKOFFLER, J. & ISHII, H. 1999. Urp: a luminous-tangible workbench for urban planning and design. *Proceedings of the SIGCHI conference on Human factors in computing systems.* Pittsburgh, Pennsylvania: ACM.

UNDERKOFFLER, J., ULLMER, B. & ISHII, H. 1999. Emancipated pixels: real-world graphics in the luminous room. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* Los Angeles, California: ACM Press/Addison-Wesley Publishing Co.

VILLAR, N. 2007. *Flexible Physical Interfaces.* Doctor of Philosophy in Computer Science, Lancaster University.

VILLAR, N., BLOCK, F., MOLYNEAUX, D. & GELLERSEN, H. 2006. VoodooIO. *Proceedings of the ACM SIGGRAPH 2006 Emerging technologies* Boston, Massachusetts: ACM.

VILLAR, N. & GELLERSEN, H. 2007. A malleable control structure for softwired user interfaces. *Proceedings of the first international conference on Tangible and embedded interaction.* Baton Rouge, Louisiana: ACM.

WACHS, J. P., KÖLSCH, M., STERN, H. & EDAN, Y. 2011. Vision-based hand-gesture applications. *Communications,* 54**,** 60-71.

WALDNER, M., HAUBER, J., ZAUNER, J., HALLER, M. & BILLINGHURST, M. 2006. Tangible tiles: design and evaluation of a tangible user interface in a collaborative tabletop setup. *Proceedings of the 18th Australia conference on Computer-Human Interaction: Design: Activities, Artefacts and Environments.* Sydney, Australia: ACM.

WEISER, M. 1991. The computer for the 21st century. *SIGMOBILE Mobile Computing and Communications Review,* 3**,** 3-11.

WEISER, M. & BROWN, J. S. 1996. Designing calm technology. *PowerGrid Journal,* 1**,** 75-85.

WEISS, M., JENNINGS, R., KHOSHABEH, R., BORCHERS, J., WAGNER, J., JANSEN, Y. & HOLLAN, J. D. 2009. SLAP widgets: bridging the gap between virtual and physical controls on tabletops. *Proceedings of the 27th international*

conference extended abstracts on Human factors in computing systems. Boston, MA, USA: ACM.

WELLNER, P. 1991. The DigitalDesk calculator: tangible manipulation on a desk top display. *Proceedings of the fourth annual ACM symposium on User interface software and technology*. Hilton Head, South Carolina: ACM.

WELLNER, P. 1993. Interacting with paper on the DigitalDesk. *Communications of the ACM,* 36**,** 87-96.

XIAO, R., HARRISON, C. & HUDSON, S. E. 2013. WorldKit: rapid and easy creation of ad-hoc interactive applications on everyday surfaces. *Proceedings of the 2013 ACM annual conference on Human factors in computing systems*. Paris, France: ACM.

ZIGELBAUM, J., KUMPF, A., VAZQUEZ, A. & ISHII, H. 2008. Slurp: tangibility spatiality and an eyedropper. *Extended abstracts on Human factors in computing systems*. Florence, Italy: ACM.

ZIOLA, R., GRAMPUROHIT, S., LANDES, N., FOGARTY, J. & HARRISON, B. 2010. OASIS: Examining a Framework for Interacting with General-Purpose Object Recognition. University of Washington, Intel Labs Seattle.

Appendix A – User Study Material

This appendix provides the resources used as part of the user study discussed in Chapter 7.

University of South Australia

School of Computer and Information Science

Division of Information Technology, Engineering and the Environment

Participant Information Sheet

Evaluating physical programming by demonstration

Researcher: James Walsh **Supervisor: Prof. Bruce Thomas**

(08) 830 23976 **(08) 830 23464**

Thank you for your interest in the study. Your participation in the study is strictly voluntary.

The aim of this research is evaluate the effectiveness of a developed system in allowing users to utilize everyday objects to develop interactive systems. Using a projector and camera system, users can introduce objects into the system and then define interactions between the objects. For example, when block A moves to block B, change the colour of block B to red. With this study, we aim to evaluate the effectiveness of this system for people without prior experience using such systems. We hope to prove that the developed system is an effective means for developing interactive systems using everyday objects.

The study will take place indoors on the Mawson Lakes campus of the University of South Australia. You will be asked sit in front of a desk with a projector and camera mounted above. You will be asked to place a number of everyday objects on the table, and then, given prompts from the system using the projector, move and interact with the objects in such a way that you can instruct the system about certain interactions that you want to occur (given the instructions). Your interaction with the system may be recorded using audio and/or video.

You may withdraw from the research project at any stage without affecting your status, treatment, and care. There is no known risk in participating in the study.

All information collected as part of this study will be retained a period of five years in a safe environment, after which time it will be destroyed. Electronic data will be encrypted, and with paper based copies being shredded following their conversion to electronic format. Data will be stored electronically on hard drive in a restricted access lab in MLK D1-07. Only the researchers involved in this project will have access to this data. All records containing personal information will remain confidential, and no information which could lead to the identification of any individual will be released. The researcher will take every care to remove responses from any identifying material as early as possible. Individual's responses will be kept confidential by the researcher and not be identified in the reporting of the research. However the researcher cannot guarantee the confidentiality or anonymity of material transferred by email or internet. As a participant in this study, you will be given the opportunity to receive a copy of any publication made that has reference to the results obtained.

This project has been approved by the University of South Australia's Human Research Ethics Committee.

The Executive Officer HREC is available to discuss any ethical concerns you may have about this user study. Her contact details are included below.

Vicki Allen

Executive Officer
Human Research Ethics Committee, University of South Australia
Research Services, Mawson Lakes Campus
Email: vicki.allen@unisa.edu.au
Phone: 8302 3118
Fax: 8302 3921

University of South Australia

Consent Form

Project title: Evaluating Ad-hoc Tangible User Interfaces.

Researcher's name & contact details:

James Walsh – james.walsh@unisa.edu.au

Supervisor's name & contact details

Prof. Bruce Thomas – bruce.thomas@unisa.edu.au

- I have read the Participant Information Sheet and the nature and purpose of the research project has been explained to me. I understand and agree to take part.
- I understand the purpose of the research project and my involvement in it.
- I understand that may be filmed and audio recorded during my participation in this study. I understand that I may opt out if I desire.
- I understand that I may withdraw from the research project at any stage and that this will not affect my status now or in the future.
- I understand that while information gained during the study may be published, I will not be identified and my personal results will remain confidential.

Name of participant…….....

Signed…….**Date**…….

I have provided information about the research to the research participant and believe that he/she understands what is involved.

Researcher's signature and date……...

This project has been approved by the University of South Australia's Human Research Ethics Committee. If you have any ethical concerns about the project or questions about your rights as a participant please contact the Executive Officer of this Committee, Tel: +61 8 8302 3118; Email: Vicki.allen@unisa.edu.au

TAM System Poses

(One Arm Extended)

New Object

(One Arm Raised)

Cancel

(Both Arms Open— palms up)

New Group

(Both Arms on Table at 90°)

New Interaction

(Both Arms Parallel)

Confirm

Questionnaire

Evaluating Ad-hoc Tangible User Interfaces

Age:

Gender: M F

Have you used an interactive table-top system before, such as Microsoft Surface? (*Please circle*)

'What is a table-top system?' No Yes Extensively

Have you used tangible user interfaces before (anything other than normal mice and keyboards)? (*Please circle*)

'What are tangible user-interfaces?' No Yes Extensively

Please answer the following questions based on y*our* interactions with the system:

1. How intuitive did you find the process of formally 'introducing' and 'defining' objects (i.e. placing them down, performing the gesture, then entering a name and default colour)? *(please draw a line to indicate your response)*

   Intuitive ._____._____.Unintuitive

2. How easy did you find the process of actually 'introducing' and 'defining' objects? *(please draw a line to indicate your response)*

   Easy ._____._____.Hard

3. How intuitive did you find the concept of 'grouping' objects? *(please draw a line to indicate your response)*

   Intuitive ._____._____. Unintuitive

4. Did you understand what the grouping of objects was trying to achieve? *(please circle and comment)*

   Yes No

   Please Comment:

_____

_____

_____

_____

5. How easy did you find the task of 'grouping' objects? *(please draw a line to indicate your response)*

   Easy ._____._____.Hard

6. How intuitive did you find the concept of defining rules for an interaction between objects? *(please draw a line to indicate your response)*

   Intuitive ._____._____. Unintuitive

7. How easy did you find the process of creating a set of rules between objects to define an interaction? *(please draw a line to indicate your response)*

   Easy ._____._____. Hard

8. How intuitive did you find the pre-defined gestures used in the system? *(please draw a line to indicate your response)*

   Intuitive ._____._____. Unintuitive

9. How appropriate did you find the pre-defined gestures used in the system (i.e. how well did they match their task)? *(please draw a line to indicate your response)*

   Appropriate ._____._____. Not appropriate

10. How responsive did you find the system to be? *(please draw a line to indicate your response)*

    Responsive ._____._____. Unresponsive

11. List at least three parts of the system that you found particularly good:

_____

_____

_____

_____

12. List at least three parts of the system that you found particularly bad:

_____

_____

_____

_____

13. Did you feel the system provided enough guidance for you to perform the required tasks? *(please circle)*

    Not Enough ._____._____.Too Much

14. Compared to how you would describe the task's objective to **another person**, how logical was the order in which the system made you perform tasks? *(please draw a line to indicate your response)*

Logical ._____._____. Illogical

- Please Comment:

_____

_____

_____

_____

15. Compared to hot you thought the problem through in **your head**, how logical was the order in which the system made you perform tasks? *(please draw a line to indicate your response)*

Logical ._____._____. Illogical

- Please Comment:

_____

_____

_____

_____

16. Do you have any other comments or suggestions?

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

End of Questionnaire

Appendix B – Simple Application/Interaction in Code

The TAM architecture can be utilised in a number of ways. Aside from being used to enable end-user extensibility, the architecture could also be used as a tangible framework in itself, abstracting much of the complexity to simplify developing traditional, fixed tangible systems. Abstracting away any system-specific code or functionality, the core functionality of the architecture can easily be leveraged within code. Code Excerpt 4 (creating the objects) and Code Excerpt **5** (creating the interaction) show how a simple 'infection' game could be created using the implementation of the architecture. When two objects come in close contact, the infected entity infects the non-infected entity, changing its colour as when infected. Overall, defining two objects, their default colours and creating the interaction is only 19 lines of code in total, eight of which are creating the objects to use (which would only be done once separately when the objects are introduced) and 11 to create the actual interactive component.

```
// Create and define two objects, one red on that's 'infected', one normal

InteractionObject infected = new InteractionObject("Infected");

InteractionObject clean = new InteractionObject("Clean");

ColourProperty aColour = new ColourProperty();
ColourProperty->GetSubproperty("Red")->Set(255); // get/set Property named "Red"
ColourProperty->GetSubproperty("Blue")->Set(0); // get/set Property named "Blue"
ColourProperty->GetSubproperty("Green")->Set(0); // get/set Property named "Green"

infected->Properties->Add(aColour); //add red Colour Property to infected

clean->Properties->Add(new ColourProperty()); //add Colour Property to clean
```

Code Excerpt 4 Defining two objects for an 'infection' game

```
// Basic interaction showing how one object might 'infect' another in a game
// using previously created/introduced objects created from
Interaction* infectionInteraction = new Interaction();
// Set up the specific Action to monitor to be based on Proximity
ProximityAction proxyAction = new ProxyAction();

// Set the action based on current state of A & B
proxyAction->Set( { A, B } );

// Assign the ProximityAction instance to the interaction
infectionInteraction->AssignAction(proxyAction);

// Set up the result of the interaction so the infection 'spreads' by
// creating a PropertyMap and a single mapping between the two objects
PropertyMap* pm = new PropertyMap();
infectionInteraction->Add(pm);
PropertyMapping* singleMapping = new PropertyMapping();
pm->Add(singleMapping);

// Tell the mapping to copy from infected object
singleMapping->From = infected->GetProperty("Color");
```

```
// to the clean object
singleMapping->To = clean->GetProperty(“Color”);

// Store our interaction so the system can evaluate in subsequent frames
storeInteraction(infectionInteraction);
```

Code Excerpt 5 Simple Interaction demonstrating an 'infection' game

Appendix C – Public Demonstration

A public demonstration of the system was shown at the 2013 International Symposium on Mixed and Augmented Reality (ISMAR 2013), held in Adelaide, Australia (Figure 98). Attendees had the opportunity to play with the system in use, creating a number of different touch or tangible controls using the available resources.

Figure 98 ISMAR 2013, reprinted with permission from Stewart von Itzstein

Over 70 attendees directly used the system, with many others of the 230 total attendees observing and providing feedback. Attendees found the system interesting, with many intrigued with the application of such a system in their own works in augmented reality, which by its definition is already requiring the user to perform interactions in the real world, utilising the affordances of the available objects for the augmented interactions. The primary interests received involved the use of the system for the early stages of system's development that are currently undergoing prototyping. Feedback was positive, with attendees receptive to the array of unique controls that could be developed, especially those made from passive materials, e.g. the 3D printed lever and the improvised mug joystick.

One attendee representing a major automobile manufacturer was intrigued in the application of the ad-hoc controls to car dashboard design. They commented that currently it is a very involved process to create functional prototypes to even just evaluate the positioning of different controls, and approach such as this would greatly reduce the amount of time required, whilst allowing the designers to more quickly evaluate different design decisions.

Another attendee from a major defence contractor saw promise and application of the technology in a number of areas. One was the customisation of controls for radar operators, noting that different operators have their own unique configuration and

workflow for operating the machinery, and that it would be beneficial if at the start of each shift the operator could layout or load their own configuration of controls in addition to the standard ones. The other area was the application of the original rule-based logic interactions for field-planning. The attendee noted different entities in the environment could rapidly be modelled, with different constraints created to guide planners regarding the appropriate deployment and travels paths for resources.

Appendix D – Attachments

Additional materials are provided to support this dissertation.

CD-ROM

The attached CD-ROM provides electronic copies of the following materials:

- demonstration videos of content presented in this dissertation,
- an electronic version of this dissertation in PDF format, and
- an electronic version of the current author publications to date in PDF format with accompanying videos.

Internet

A copy of this dissertation along with updates to this project can be found on the following website:

http://www.setoreaustralia.com/storage/james/thesis/