THE UNIVERSITY OF CALGARY

The Single Display Groupware Toolkit

by

Edward Hiatt Tse

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER 2004

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "The Single Display Groupware Toolkit" submitted by Edward Hiatt Tse in partial fulfillment of the requirements for the degree Master of Science.

Supervisor, Saul Greenberg
Department of Computer Science

Frank Maurer
Department of Computer Science

External Examiner, Mike Chiasson
Haskayne School of Business, University of Calgary

Date

Abstract

Researchers in Single Display Groupware (SDG) explore how multiple users share a single display such as a computer monitor, a large wall display, or an electronic table top display. Yet today's personal computers are designed with the assumption that one person interacts with the display at a time. Thus researchers and programmers face considerable hurdles if they wish to develop SDG. Our solution is the SDGToolkit, a toolkit for rapidly prototyping SDG. SDGToolkit automatically captures and manages multiple mice and keyboards, and presents them to the programmer as uniquely identified input events relative to either the whole screen or a particular window. It transparently provides multiple cursors, one for each mouse. To handle orientation issues for table top displays (i.e., people seated across from one another), programmers can specify a participant's seating angle, which automatically rotates the cursor and translates input coordinates so the mouse behaves correctly. Finally, the SDG Toolkit provides an SDG-aware widget class layer that significantly eases how programmers create novel graphical components that recognize and respond to multiple inputs. This thesis explores the design and evaluation of the SDG Toolkit.

Publications

Materials, ideas, and figures from this thesis have appeared previously in the following publications:

Tse, E. and Greenberg, S. (2002), **SDGToolkit: A Toolkit for Rapidly Prototyping Single Display Groupware**, *Extended Abstracts of the ACM Conference on Computer Supported Cooperative Work (CSCW '02)*, New Orleans, pp. 173-174.

Tse, E. and Greenberg, S. (2004), **Rapidly Prototyping Single Display Groupware through the SDGToolkit**, *Proceedings of the Fifth Australasian User Interface Conference*, *In Conference in Research and Practice in Information Technology (CRPIT)*, Dunedin, New Zealand, pp. 101-110.

Tse, E. and Greenberg, S. (2004), **SDG Toolkit**, *Video Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '04)*, Chicago, To Appear

Acknowledgments

This thesis would not have been possible without the support of many people:

To Saul my supervisor and mentor thank you for inviting me to work with your group as an undergraduate. You've given me the opportunity to explore my "creative" ideas despite their sometimes limited research potential. Thanks for your patience and guidance throughout the writing of my thesis work.

To my colleagues and friends at the Interactions Laboratory, this thesis would not have been possible without your thoughtful support with conference presentations, videos, brainstorming, and thesis revisions. I will always cherish the time we've spent together.

To Mike Boyle, thank you for providing me with the technical background needed to do my thesis work. Thank you for teaching me the art of programming and solving technical problems. I've learned so much from you.

To Tony Tang, thank you for actively using my thesis software even during its early stages of development. You're comments have significantly improved the quality of my thesis research.

To Sheelagh Carpendale, Rob Diaz-Marino, Stephan Habelski, Mark Hancock, Jonathan Histon, Russell Kruger, Stacey Scott, Nicole Stavness, Charlotte Tang, and Nelson Wong thank you for taking an interest in my thesis research. Your insightful and inspirational comments about my work has had a tremendous impact.

To Carman Neustaedter, thank you for reviewing my thesis in its early stages. I will always cherish the heart felt meetings we had in the quiet room.

To all the members of my family, you've had an immeasurable impact on my life. Thank you for sharing this wonderful experience with me.

To my parents, Jim and Jane, thank you for your patience and support throughout my academic career. Whenever I experienced difficulties you were there to support me. Thank you for everything.

To my brother David, thanks for sharing your triumphs and frustrations with me. You're always willing to lend a helping hand when I need it, thank you.

To my friends at Campus Crusade for Christ, ever since my first year in University you have welcomed me as part of your community, thank you for letting me be a part of such a worthy cause. The experiences we've had together are unforgettable.

To Shawn Cain, my mentor, thank you for answering my questions of faith during my first year of University. My life would not be the same without you. Thank you for your enthusiasm and dedication.

To my friends at the Emmanuel Fellowship, thank you for sharing your lives and experiences with me. We are a forever inseparable community.

To Peggy Chan, thank for your patience, kindness, warmth, and affection. Thank you for taking an interest in my day to day activities and showing me that every cloud has a silver lining. You give me the strength to pursue my dreams.

Dedication

I dedicate this thesis to my parents, Jim and Jane. You've watched me grow up, taught me about life along the way and then supported me as I pursued my goals. You have always guided me, loved me and wanted the best for me. Words cannot express my gratitude for everything that you have done.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

In this thesis I focus on technical aspects of Single Display Groupware (SDG). SDG is a groupware genre that investigates how multiple people share a single display such as a computer monitor, a large wall display, or an electronic whiteboard using multiple input devices such as multiple keyboards and multiple mice. This chapter sets the scene in three sections. I first enumerate the general problems associated with developing SDG applications. Next, I formalize these issues into goal statements and then describe how I plan to resolve these issues. Finally, I conclude this chapter with an organizational overview of this thesis.

1.1 Background

Throughout a typical day, co-workers located in the same office or work area will meet to engage in information sharing, brainstorming ideas, document writing, and collaborative design. Such meetings often involve shared surfaces such as a whiteboard or paper spread over a table, where interactions are done primarily with pens and felt markers. While collaborative work over conventional media has many proven benefits, researchers in Computer Supported Cooperative Work (CSCW) have argued that computer supported shared surfaces can further assist collaboration in ways that were not previously possible or practical. For example, just as people working alone on a computer benefit from extremely powerful productivity tools to create, modify and save documents, so should similar advantages accrue if computer support is provided for co-located collaborators.

This thesis is concerned with the construction of computer support tools for co-located groups. In particular, I focus on *single display groupware* (SDG), defined as software viewed on a single display shared by co-located collaborators using multiple input devices [Stewart, 1999]. While early SDG implementations required turn taking, most now allow people to work simultaneously. SDG is important, for many researchers

believe that wall displays and electronic table tops will provide high collaborative opportunities. As display technology improves, SDG is inevitable because people will naturally expect to be able to work together over these large interactive surfaces.

Yet today's computers are not designed for co-located collaboration. Applications assume that only one person will use it at a time; operating systems recognize only one set of devices (usually a single mouse and keyboard) for input; they provide only one input focus (i.e., text insertion point and cursor position) [Hourcade, 2003].

The fundamental problem is that operating systems are designed with the assumption that there will only be one user per computer, and as a consequence SDG becomes hard to build. For example, when one plugs in multiple keyboards or mice into today's computers their inputs are merged into a single stream; no information identifying which keyboard or which mouse generated that input is supplied to the application. It is quite likely that single display groupware will redefine how we interact with computers today. However, before we can reach this point, researchers need to be able to build and test prototype SDG applications.

1.2 Our Own Painful First Steps in SDG Development

To illustrate how painful SDG development can be, I illustrate what researchers within our own group had to do to develop Single Display Groupware applications. Conceptually, SDG should have been easy to implement as there is a rich intellectual history in SDG design. For example, prototype SDG systems had already been developed over a decade ago at Xerox PARC. Their Multi Device, Multi-User, Multi Editor (MMM) System [Bier, 1991] allowed two collaborators to manipulate rectangles and write text using two mice and a shared keyboard. This novel interaction technique was followed by several studies on how children benefit from sharing a display in an educational setting [Druin et al, 1997, Inkpen et al. 1997], as well as explorations on how computer support could be provided for large interactive displays and tables [Steitz et al., 1999].

Seeing the value in this research several of our lab members set out to prototype SDG applications. The first step involved finding and making use of a tool that another

researcher had built to develop SDG applications, called the Multiple Input Devices (MID) Toolkit [Bederson et al., 1999]. MID was one of the only toolkits that provided programmers with a low-level means of obtaining input from multiple mice as separate streams. MID made SDG development possible, but because it was still fairly low level, SDG applications were difficult to build. Still, Zanella and Greenberg had some initial success developing one class of SDG interface widgets [2001]. Ultimately, they had to abandon MID because it proved inexorably bound to Windows '98 and Java. As the lab migrated to more modern versions of Windows, MID no longer worked. As well, MID forced people to work within Java even though our group had reasons to move onto a different language and development environment. Essentially, MID was a minimal hardwired point solution to SDG development, was low level, and did not have the power or flexibility we required.

Researchers then sought alternatives to capturing input from multiple mice. We and other researchers were exploring the use of PDAs as input devices [Myers, et al., 1998; Greenberg, et al., 1999]. While PDAs have their place in SDG, we believed that it was important to include input devices that are familiar to the majority of computer users i.e., mice and keyboards.

Thus our lab returned to the multiple mice input problem. From a technical perspective, operating systems such as Windows recognize a mouse as a special device where multiple mice input is merged into a single stream. The first solution was to rewrite the firmware of an existing USB mouse so that the system would no longer recognize it as a mouse (it actually saw it as a USB device that we named a 'marmot' seen in Figure 1.1). Greenberg and Fitchett then wrote customized drivers so that each marmot was recognized as a programmable physical widget or phidget (2001). Phidgets present the input obtained from multiple marmots to the programmer as separate input streams. While this worked, coordinate tracking and cursor drawing was still painful and inefficient since we did not use the most efficient implementation. It also meant that outside researchers could not use our applications unless we supplied these hand-crafted devices to them.

Figure 1.1. A Phidget marmot: The firmware for these mice was rewritten so that the computer would not recognize them as regular mice. They were marked with special stickers to distinguish them from regular mice.

What was especially disconcerting was that people's time and effort went into infrastructure development versus our main focus: the design and evaluation of SDG interaction techniques for upright and table top displays. We needed a simpler way to develop Single Display Groupware applications so that we could focus on the design of our applications rather than the "plumbing" involved.

This is the reason why I set a new goal of developing a toolkit to simplify the design of SDG applications. I wanted to go beyond mere input capture, where I would address programming concerns that would minimize the otherwise large overhead associated with basic SDG functionality. I wanted to target SDG researchers with limited programming experience, where the toolkit would make it easy for them to develop basic SDG applications, and possible for them to design complex ones as well.

This thesis describes the development of the Single Display Groupware (SDG Toolkit), a toolkit designed to simplify the development of Single Display Groupware applications. As we will see, the SDG Toolkit removes low-level SDG implementation burdens and gives programmers the necessary building blocks to develop interesting and novel SDG interaction techniques and applications. In the next section, I will enumerate some of the problems associated with SDG development.

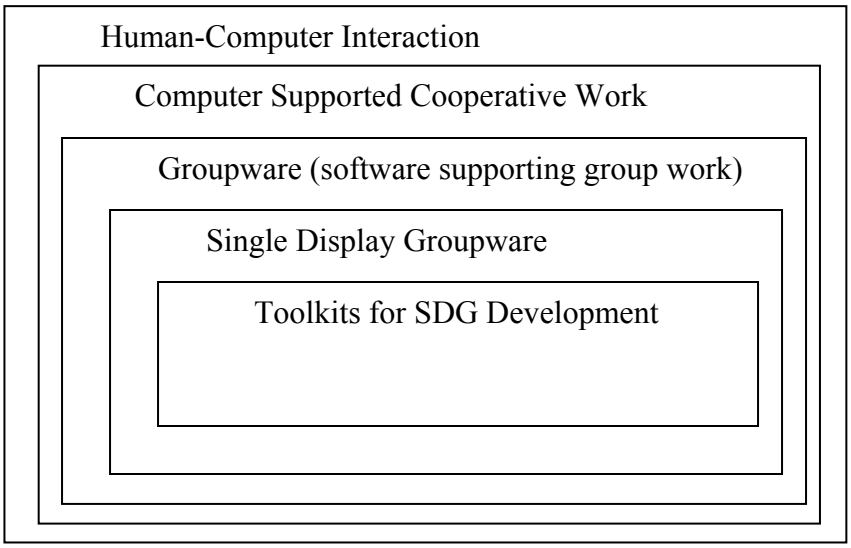


Figure 1.2. The context and scope of my research

1.3 Thesis Problems

Single Display Groupware itself is a broad topic situated in the fields of Human Computer Interaction and Computer Supported Cooperative Work. Figure 1.2 illustrates the context and scope of my research. In particular, I address the following problems in this thesis:

1. **We do not have a means to quickly and rapidly prototype SDG applications using multiple mice and keyboards**: The assumptions present in today's operating systems hinder the development of SDG applications by presenting the following hurdles to SDG development:

a) ***Multiple input and identification***: Since computers assume that there will only be one mouse or keyboard per system, applications that support multiple mice and keyboards often require programmers to build their own device drivers or use low level APIs to develop even the most basic SDG applications.

b) ***Cursor drawing***: Even after the basic SDG input problems have been resolved there is still a need to draw an individual pointer or cursor for each mouse. Often manually drawing a cursor for each mouse adds additional work for developers and results in sluggish application performance if not done in the most efficient manner.

c) ***Table top Support***: Current systems assume a single orientation for each mouse but SDG researchers need to support different orientations when designing table top applications.

2. **We do not have a means of easily prototyping SDG aware interaction widgets**: Just as widgets encapsulate common single-user interaction techniques, so will SDG interaction widgets capture multi-user interaction methods. Yet SDG widgets will be quite different, for they need to understand concurrent use. Starting with the familiar, for example, how would a graphical slider behave when used by multiple people simultaneously? I address the following sub problems in this thesis:

a) ***SDG widget infrastructure***: In order for SDG widgets to work there must be a way for SDG widgets to communicate with the SDG input API. A low level infrastructure must exist to identify SDG widgets and send notifications of multiple mouse movements and keyboard events. Currently, programmers need to write code to explicitly communicate SDG input events to each SDG widget. This results in extra programming effort required by the programmer and the risk of sluggish application performance if done crudely.

b) ***Basic SDG widget primitives***: To prevent all widgets from being developed from scratch we need to provide basic SDG widget building blocks as a starting point for developing more complicated widgets. Otherwise all SDG widgets must be developed from scratch. We do not have the basic widget building blocks needed to rapidly prototype simple and novel SDG widgets.

1.4 Thesis Goals

The primary goal and expected contribution of this thesis is the design and implementation and validation of a toolkit for rapidly prototyping single display groupware applications. I have several sub-goals:

1. **I will create an application programmers interface called the SDG Toolkit that will allow average programmers to easily develop SDG applications**: I will accomplish this goal in three steps:

a) ***Develop an SDG input API***: I will create an application programmer's interface (API) which allows programmers to access input from multiple mice and keyboards. This API will allow SDG applications to be developed easily by average programmers.

b) ***Automatically draw multiple cursors***: With the SDG Toolkit API, I will also provide a customizable graphical cursor for each mouse so that programmers will not need to add cursor drawing code into their own SDG applications.

c) ***Provide table top support***: I will provide a means for programmers to design applications that recognize different orientations of multiple mice over a table top display.

2. **I will develop a widget layer that enables average programmers to create novel SDG widgets**: I will accomplish this goal in two steps:

a) ***Develop an SDG widget infrastructure***: I will develop a basic SDG Widget Infrastructure that automatically detects SDG widgets and sends notifications of input events from multiple mice and keyboards.

b) ***Develop SDG widget primitives***: I will create SDG widget primitives which can be used by programmer to develop more complex widgets. These widget primitives could be extended to develop more complex novel widgets.

1.5 Organizational Overview

This thesis is divided into six chapters:

In Chapter 2, I present a technical overview of SDG. I first motivate why SDG is worth exploring. I then give a brief overview of how SDG systems manage input from multiple people. Finally, I extract commonalities of these systems to develop a list of requirements for an SDG toolkit.

In Chapter 3 I introduce the SDG Toolkit architecture through seven main components. I first discuss the issue of gaining input from a number of devices connected to the same computer as separate streams. Next, I explore the issue of

presenting these streams of input as uniquely identified input events for programmers. I then discuss the transformation of pointer data to window coordinates. Next, I discuss the drawing of multiple cursors for each user. I then discuss the implications of supporting horizontal and vertical displays. Next, I discuss solutions for handling the system mouse and managing multiple keyboard foci. I end this chapter with descriptions of several example applications that summarize and connect the seven aspects mentioned above.

In Chapter 4 I discuss the widget layer of the SDG Toolkit through two main components. First, an SDG widget infrastructure supports the identification of SDG widgets and forwards SDG events. Second, the provision of SDG widget primitives, which are basic building blocks that end programmers can extend when developing their own SDG widgets.

In Chapter 5, I provide validation that SDG applications and SDG widgets can be easily designed using the SDG Toolkit. I do this by describing several case studies of how the SDG Toolkit has been used in practice. I first describe case studies of my own development experiences using the SDG Toolkit, and use these as a self evaluation. I then describe the experiences of other SDG researchers using the SDG Toolkit and its widget layer. I further validate the design of the SDG Toolkit through case studies where I show that the principles of the SDG Toolkit do generalize to other novel input devices.

In Chapter 6, I conclude the thesis with discussions of my research contributions. First, I recap the thesis problems discussed in Chapter 1. This is followed by a detailed description as to how each thesis problem is solved with the SDG Toolkit. Second, I detail some of the future work I have done and that I am planning to do. I then conclude this thesis with a discussion of the future of SDG.

Chapter 2. Single Display Groupware and Toolkits

In this chapter, I present a technical overview of SDG. I first motivate why SDG is worth exploring. I then give a brief literature review of how SDG systems manage input from multiple people. Finally, I extract commonalities of these systems to develop a list of requirements for an SDG toolkit.

2.1 Motivation

SDG is worth exploring for three reasons. Computers today are being used by multiple people despite the limitations imposed by the operating system. Studies have shown that there are benefits to being able to work together on the same display. Finally, large displays naturally afford interaction by multiple users. Each reason is briefly elaborated below.

First, people today are using computers for collaboration despite the fact that the operating systems are unaware of this collaboration. We have all done this when providing over the shoulder help to colleagues working on their computer e.g., as when we tutor someone using computer software. For example, Nardi and Miller [1990] studied people using spreadsheets. They found that one reason spreadsheet users are so productive is because they successfully enlist the help of other, more knowledgeable users in constructing their spreadsheets. Through over the shoulder assistance, experienced co-workers share domain knowledge with less experienced colleagues, using the spreadsheet as a medium of communication. People already use computers for collaborative activity so it is natural to expect that the computer should explicitly support this collaboration.

Figure 2.1. Two boys playing in the one-mouse/one-cursor condition (left) and two girls playing in the two-mouse/two cursor condition (right). Photo from Inkpen et al., 1999

Second, studies of children using a single computer have shown that collaboration over a shared display has real benefits, especially when multiple mice are available. For instance, Inkpen et al., [1995] showed that children working side by side on the same display are at an advantage compared to children working side by side on different computers. Studies have also shown that multiple mice (one for each person) is better than sharing a single mouse. For example, Stewart et al., [1998] showed that children forced to share a single mouse often respond by either fighting for control of the mouse or by being frustrated as they do not feel like an equal participant of the collaborative activity. While control of the mouse does not clearly indicate control over group collaboration [Cole, 1995] subsequent studies have shown that students without control of the mouse periodically loose interest and look away from the task on the computer screen [Inkpen et al., 1999], as seen in Figure 2.1. In contrast, Scott et al. [2002] and Inkpen et al. [1995] showed that children prefer working together on the same display and that they are able to solve more puzzles when working together with multiple mice than working alone in separate rooms.

Finally, the recent proliferation of large display technologies introduces a new genre of surfaces that afford multiple users. Because large displays can be easily viewed by many people, this makes them well suited for collaborative activity. While sharing displays may seem somewhat odd to those used to conventional computers, we must realize that their solitary nature is an artefact of technology design. In contrast, almost all real world artefacts are shareable. People now collaborate over large display surfaces even though they have to take turns using a single mouse and keyboard connected to the

computer [Bishop and Welch, 2000, Greenberg, 1999]. In summary, a natural next step in interface design is to provide a means for multiple people to collaborate over a shared display. That is, single display groupware should be a natural capability of all computers.

2.2 Styles of Input for SDG systems

True SDG systems differ from sharing a traditional computer because they recognize and support multiple input devices. There are a variety of ways to provide multiple inputs to an SDG system. I begin with a discussion of workarounds that have been used to share a single computer. Next, I discuss specialized input devices created to support multiple users, followed by how multiple mice are supported on a single computer. Then, I discuss multi-user interaction when sharing a large vertical display or a table top display. Finally I look at several distributed toolkits; while these are not SDG systems they share related design concerns.

2.2.1 Sharing Existing Input Devices

When coworkers collaborate using a shared computer today they need to work around the assumption that there will be only one user per computer. It is common for multiple people to share the computer by taking turns using the mouse or the keyboard. It is also possible to create software that recognizes multiple users by dedicating a section of the keys on the keyboard to each user.

A simple way for collaborators to provide text and pointer input to a shared desktop monitor is to share a single keyboard or mouse using turn taking. This method is commonly seen in over the shoulder interactions when one user assumes control of the computer by grabbing the keyboard or mouse and interacting with the system.

Alternatively, some systems allow multiple mice and keyboards to be plugged into the system (ideally one per user). Unfortunately, operating systems manage multiple devices by merging the input streams. For example, since both mice would control the same cursor the participants would need to negotiate control of the mouse through social turn taking. The problem with this approach is that the computer software is oblivious to

Figure 2.2. Two people playing a computer game collaboratively with only one keyboard.

the collaborative activity happening on the computer, which limits how people can work together.

One workaround used by many computer game designers was to map keys on a keyboard to the arrow keys for multiple users. Figure 2.2 shows two people playing a popular computer game called The Simpsons. Each person would share a side of the keyboard to manipulate characters in a game. Using this method, the game knew that multiple people were interacting simultaneously and could appropriately move the two characters in the game in spite of the "one person per computer" assumption made by the operating system. This approach has limitations as both users have to share a very small amount of keyboard space and it limits each person to a subset of the keyboard.

2.2.2 Specialized Input Devices

A different approach eschews the mouse and keyboard by providing each person with specialized input devices, such as joysticks or game controllers. In particular, console systems allow multiple people to collaborate over a shared display using multiple game pads and joysticks, similarly PDAs have been used as input devices to shared displays.

Most console game systems, e.g., X-Box, Playstation, and Nintendo GameCube are specialized hardware boxes that support at least two and often four different controllers. These provide the capability for multiple people to play simultaneously over a single shared display (a television set). However using console game systems poses several hurdles to prototyping SDG applications. The first is a pragmatic concern: console

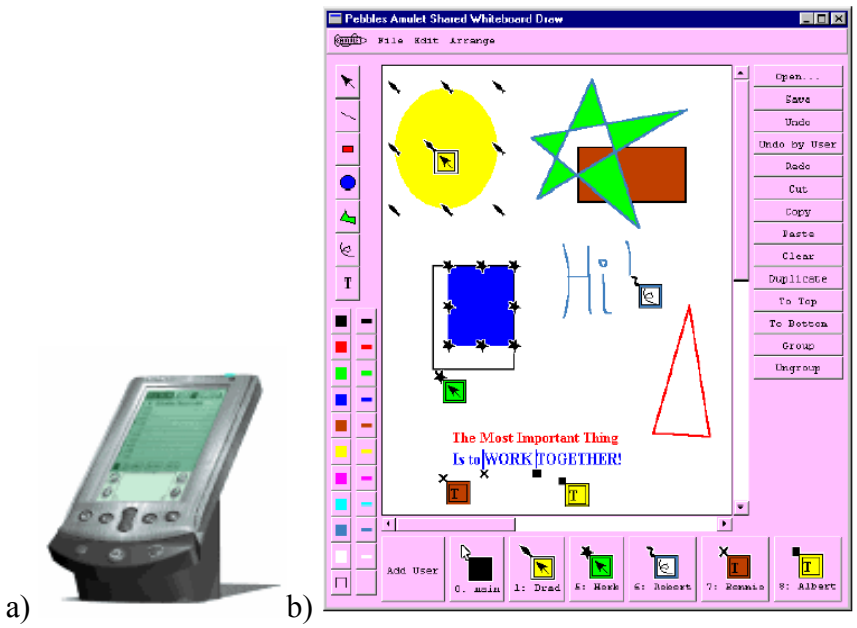


Figure 2.3. Pebbles Draw a) A Palm Pilot, b) The application seen on the shared display. Photo from Myers, et al., 1998

systems use specialized hardware and programming environments that are unfamiliar to most researchers. Next, although console game input devices are useful for simple movement in a game, they are not suited for pointing and typing tasks needed in most productivity applications

Next, multiple PDAs have been explored as an input device to SDG [Myers et al., 1998, Greenberg et al., 1999]. A good example is Myer's Pebbles Draw. Built atop his Amulet toolkit, Pebbles Draw allowed multiple people to draw simultaneously using multiple Palm Pilots (Figure 2.3a). People could move their cursor and draw on the shared display (seen in Figure 2.3b) by moving the stylus on the top portion of their respective Palm Pilot. Each person's cursor was represented on the shared display with a particular shape while large squares on the bottom palette represented the currently selected colour and drawing mode of each participant. Pebbles Draw is one of the few programs that allowed text entry from multiple participants. Text entered through the PDA would be recognized by Pebbles Draw and drawn with different colours on the screen.

2.2.3 Multiple Mice

The mouse is the most commonly used input device for pointer input. If multiple mice connected to a single computer were available for true simultaneous and independent input, collaborating participants could take advantage of the skills that they already have. In this section I will describe several development efforts made by researchers to support multiple mice on a shared computer. I begin with a discussion of how one can redevelop the windowing system to support multiple mice. Next, I discuss how low level input APIs (that reveal the workings of the windowing system) can access multiple mice as separate streams. Finally, I explore the development of high level APIs (that hide the workings of the windowing system) to support multiple mice.

One way of circumventing merged mouse input is to modify the actual windowing system. One example (and possibly the first SDG system ever) is Bier and Freeman's Multi-Device Multi-User Multi-Editor (MMM) System [1991]. To support multiple mice, they reimplemented the windowing system from scratch. MMM obtained input from up to two serial mice by directly reading their input from the serial port. In Figure 2.4, we see an example layout of the MMM system. Each user has a small window (like the one seen on the bottom left of Figure 3) that represents information specific to each participant e.g., the current drawing mode, user name and an icon representing who has control of the single shared keyboard. While MMM set the standard for future SDG

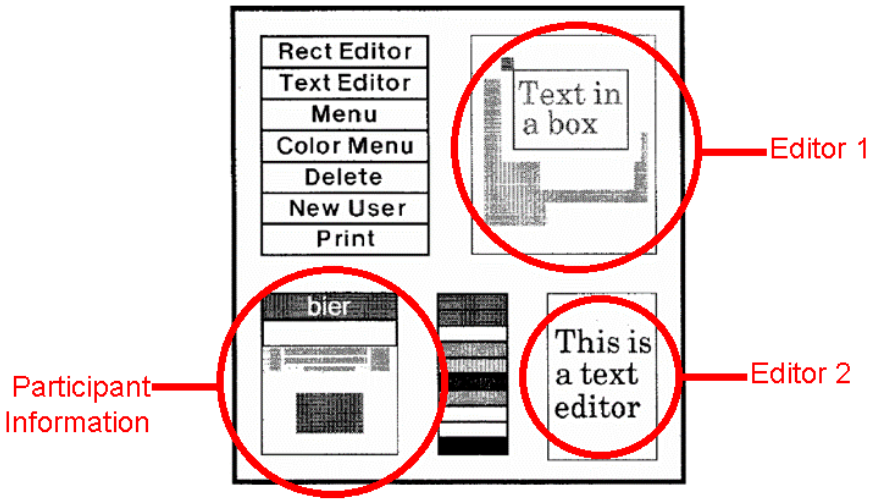


Figure 2.4. The MMM system. Photo from Bier and Freeman, 1991

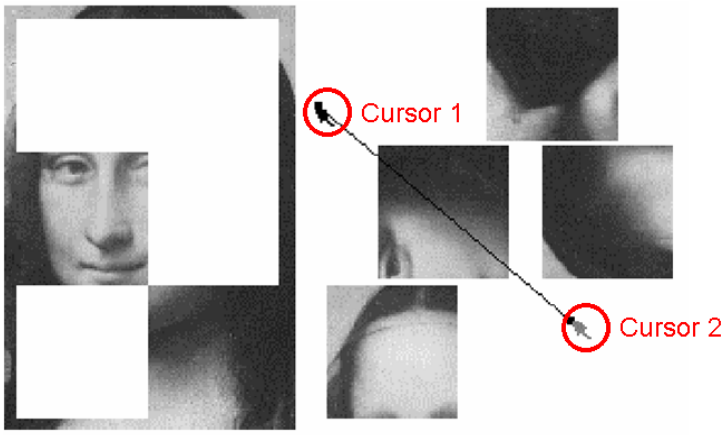


Figure 2.5. A collaborative jigsaw puzzle created using the Colt toolkit. Each piece is controlled by a line segment, where two users adjust the location and rotation of each piece by manipulating endpoints of the segment. Photo from Bricker, et al., 1999

explorations, it was not generally available to the research community. Even if it were, its low level manipulation of the windowing system meant that it would be difficult to modify and extend.

Another approach uses low level input APIs provided by the operating system to capture multiple input streams. This method was used by Colt, which allowed multiple USB mice to be treated as separate streams over a single computer [Bricker et al., 1999]. Colt provided a separate coloured cursor for each mouse, and allowed two users to simultaneously interact with and manipulate graphical objects. For example, Figure 2.5 shows two people manipulating the position and orientation of a puzzle piece. As mentioned, Colt application programmers accessed the low level input APIs directly through a window event mechanism. Yet low level APIs make it difficult to rapidly prototype applications because they require an understanding of how the windowing system handles messages between multiple threads. In contrast, most programmers (who are used to single user applications) expect to handle input from multiple mice as separate events.

Finally, a more "programmer friendly" approach is to provide an easy to use API to access input from multiple mice in high level languages. Bederson and Hourcade's [1999] MID toolkit is a good example. It internally managed the low level API complexities by packaging up the captured input stream as an API similar to Java's Mouse events. Although this simplified the task of handling input events from multiple

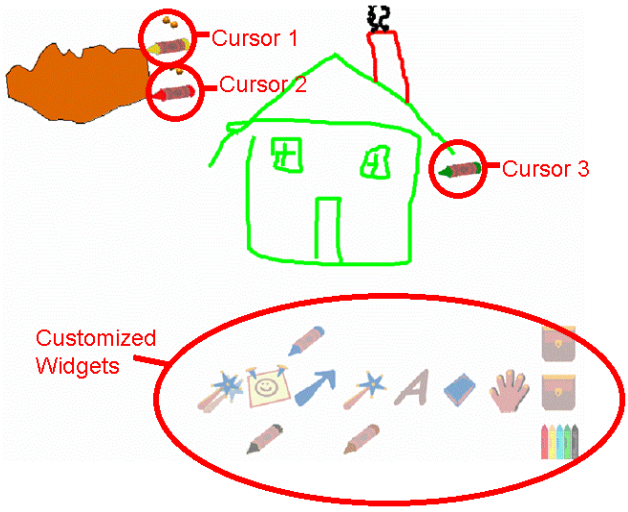


Figure 2.6. KidPad: An application developed with the MID Toolkit. Note the crayons are customized widgets built on top of MID. Photo from Bederson, et al.,1996

mice, other SDG tasks such as drawing cursors or developing custom SDG widgets had to be done manually by the programmer. Nonetheless, quite complex applications were written atop MID. These include a sophisticated drawing environment known as KidPad [Bederson, et al.,1996]. Each participant was provided with a cursor that could grab widgets such as a crayon in the drawing environment (Figure 2.6).

2.2.4 Input for Large Upright Displays

Large upright displays afford interaction by multiple users. Yet, in order for multiple people to collaborate over a single large display, suitable input devices are needed. While these could be multiple mice, the affordances of large displays suggest alternative input approaches. In this section, I will discuss a number of technologies that researchers have developed to support multiple users over a large display surface. These include the use of multiple personal computers as input devices, direct single touch display surfaces, and multiple direct touch surfaces.

Large displays may be situated in a meeting room, where people are seated behind personal workstations. Thus a "natural" way to support multiple inputs to the large display is to connect it to a computer in a distributed network technology. Mantei's CaptureLab [1988] allowed people to take turns controlling a shared projected display using personal computers. Participants would need to press a button on the keyboard to

take control of the shared projected monitor. In 1987, researchers at the Xerox Palo Alto Research Centre (PARC) developed Colab, a distributed system where multiple users could edit artefacts on a public display through their personal computers [Stefik, et al., 1987] (Figure 2.7). One application developed using Colab was Cognoter, a system for building mind maps on a shared display. From their individual PC, each person could add new text nodes and create links between nodes. While all could work simultaneously, only one person could edit a particular node at a time to prevent simultaneous editing errors. People composed the text on their personal display and then submitted the changed item after completion. This meant that users had a tendency to focus on their own display rather than looking at the shared large display and that they did not know what other people were working on until they submitted their edits to the large shared display [Tatar et al., 1991].

For more intimate gatherings directly touching the large display surface is likely more suitable. Several early displays did this by recognizing single touches. For example, LiveBoard was a pen-sensitive projection display that presented itself as a computerized white board [Elrod, et al., 1992]. Commercial single touch displays are now available e.g., Smart Technologies touch sensitive surfaces for both projected and plasma displays (http://www.smarttech.com). Since each of these systems only support a single touch they do not support multiple users, except through turn taking.

Figure 2.7. Multiple people using personal computers to control a large display in Colab. Photo Credits http://www2.parc.com/istl/members/stefik/colab.htm

One way to support multiple users on single touch display surfaces is by tiling several large displays. If we assume that only one user will use each display, people can then work simultaneously on their respective portion of the tiled workspace. This approach was used by researchers at IPSI. They tiled several touch sensitive Smart Boards to create a large interactive wall known as DynaWall (Figure 2.8b), where multiple people could work simultaneously, albeit on different Smart Boards (Streitz, et al., 1999). Since each person must interact on a different Smart board, collaborators have to stay physically distant from one another. They also could not work simultaneously on the same artifact.

Ideally all large displays should recognize multiple touches. Since identification is difficult only a few systems exist that recognize multiple touches. MERL researchers developed the Diamond Touch (Figure 2.8a), a surface capable of detecting multiple touches from different people in contact with specialized capacitive pads (usually by sitting on them) [Dietz and Leigh, 2001]. Because each person sits on a signal emitting pad, the Diamond Touch board can distinguish which person is interacting with it. For application developers, the Diamond Touch SDK offers a low level software API. Unfortunately, the Diamond Touch is limited. Its technology can support only a modest size display, which limits collaborative interaction to a very confined space (up to 42.1 inches along the diagonal axis).

An alternative to the finger would be to use a laser pointer as input to a large display. Laser pointers remove the requirement of being physically close to the large display but they often cannot capture multiple modes of interaction i.e. a left mouse button click. Olsen and Nielsen [2001] developed Xweb, a system that tracked the position of a single laser pointer using multiple web cameras. Participants could simulate a ‘mouse click’ by leaving their laser pointer in the same located for an extended period of time. Vogt et al. [2003] developed a system to support multiple laser pointers on a single large display. To distinguish different people, each laser pointer was modified so that it would emit a different blinking pattern. Using multiple web cameras these blinking patterns could be detected and used to identify each participant.

Figure 2.8. The Diamond Touch and iLand interactive surfaces: a) The Diamond Touch InputBoards b) Several vertical Smart Boards aligned adjacently in DynaWall.
Photo credits a) http://www.merl.com/projects/DiamondTouch b) Streitz et al., 1999

Another approach to capturing multiple touches was developed by Smart Technologies. Their Digital Vision Technology (DViT) uses four cameras on each corner of the display to detect up to two points of contact (http://www.smarttech.com). Using infrared technology it is possible to isolate the outline of a finger so that its position can be calculated from the four camera images. Smart's DViT system has the advantage of being able to support large display sizes (up to 77 inches along the diagonal axis); it is limited in that it cannot distinguish between different users. To access multiple points on the DViT board, programmers have to use the low level Smart Board SDK.

2.2.5 Table Top Displays

A natural extension to using a large upright display surface is to place the display surface horizontally so that multiple people can access this surface as a table. This is usually done by placing either single or multiple touch display technologies in a horizontal position. IPSI created the InteracTable by configuring a plasma Smart Board display horizontally on a table [Strietz, et al., 1999] seen in Figure 2.9. Similarly, MERL created their own multi touch table by using a downward pointing projector atop a horizontal Diamond Touch surface [Dietz and Leigh, 2001].

There are several input issues particular to horizontal displays. First, operating systems are designed with the assumption that they will be used in a single orientation,

Figure 2.9. A plasma display Smart Board placed horizontally as the InteracTable Photo credits Steitz et al., 2002

with all text, buttons and images oriented towards a single user. To develop programs that support people seated at different orientations, developers have to build applications "from the ground up." The exception is MERL's Diamond Spin, a toolkit that allows programmers to quickly develop objects in Java that could be easily oriented for any direction on a table [Shen, Vernier, and Ringel, 2004]. It currently works only with the Diamond Touch display.

Another issue is that people have a tendency to touch horizontal displays even when they are not using it, e.g., by leaning over it, by placing coffee cups on it or by resting their elbows on it. To get around this table top implementers often add a large bevel on the edge of the display, but this distances collaborators from the interactive surface.

2.2.6 Simplifying Widget Development

While collaboration-aware widget development is possible using existing SDG toolkits, they are usually quite difficult and require significant programming effort. On the other hand, distributed systems, designed to support geographically dispersed participants, have made significant advances in the simplification of widget development. While distributed systems are different from SDG, we could learn from their capabilities. In particular, groupware toolkits usually provide separate functionality for widget

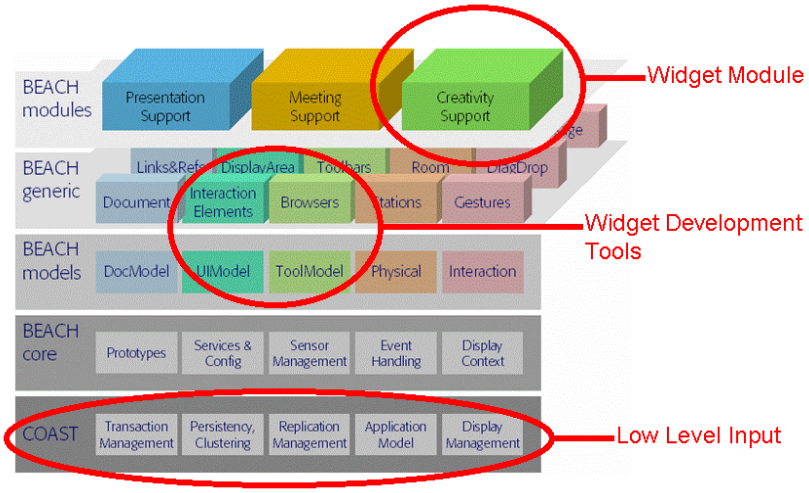


Figure 2.10 The layered BEACH architecture. Photo credits: Tandler,, 2003

development, provide example widget source code and make it possible to extend existing widgets.

Developers can rapidly prototype collaboration-aware widgets if they have tools to help them accomplish their tasks. For example, Beach included a set of tools that dealt with the interaction and visualization of collaboration-aware widgets inside a module known as "Creativity Support" (Figure 2.10). These tools allowed programmers to quickly access input functionality from distributed participants so they would not need to write their own communication routines through the low level COAST input API. Thus, programmers could focus on the design of their collaboration-aware widgets rather than worrying about the underlying ""plumbing"" involved.

Often the quickest method for learning how to use a particular widget API is to examine the source code of an existing example application. Groupkit [Roseman and Greenberg, 1995] did this by providing the source code to several stock collaboration-aware widgets (Figure 2.11a). Programmers could learn from existing examples and use them as the first step in designing their own novel widgets.

Example widgets often contain superfluous code to the basic widget application designer. A toolkit could provide a basic widget example that handled input and graphics appropriately. In addition to providing several novel widget examples (Figure 2.11b), MAUI [Hill and Gutwin, 2003] provided a simple widget control that handled distributed

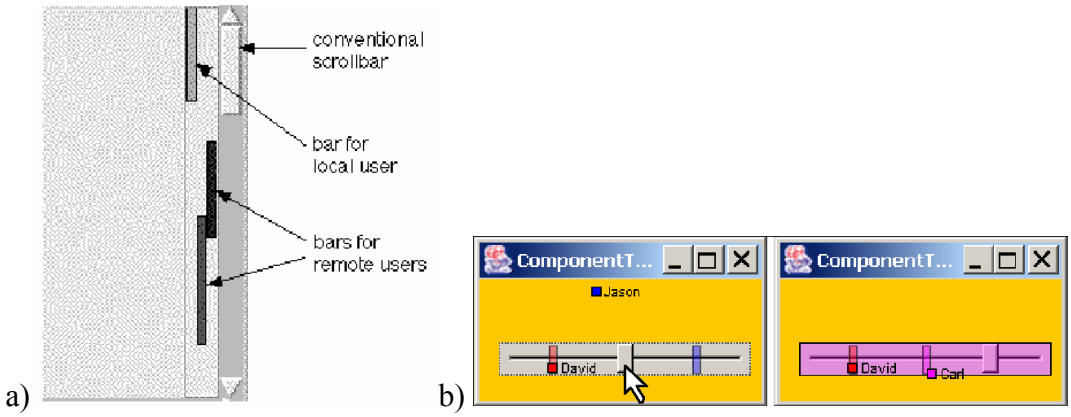


Figure 2.11. Multiple user widgets. a) Scroll bar in GroupKit b) Slider in MAUI. Photo credits a) Greenberg, et al., 1996 b) Hill and Gutwin, 2003

input events. Widget developers could extend this functionality by making their novel widgets inherit from this simple control. By using inheritance there is a distinct separation between code used to communicate with the input layer and code specific to the novel widget interaction behaviour.

In summary, we have seen many methods of collecting input to SDG displays. Some are workarounds, some are speculative technologies. My own interest lies in how we can construct SDG displays using multiple mice and keyboards. This will be the focus of the remainder of this chapter.

2.3 Toolkit Requirements

To lay the foundation for the development of a comprehensive toolkit for researchers, I examined ten toolkits that support collaborative work: Four are early multiple input device systems and toolkits, three support large displays such as the Diamond Touch board and the Smart Board, and three are designed for distance-separated collaborators that support the notion of collaboration-aware widgets. In each section I will explore the general problem, followed by related work and a recommendation for SDG toolkit designers. Table 2.2 summarizes the features across these toolkits.

2.3.1 Obtaining Low Level Input from Multiple Mice and Keyboards

At the very least, designers of SDG toolkits need to obtain low level device input as separate streams. While the designer needs to learn, access and manipulate quite low-level system APIs, callbacks and contained event information, and / or device drivers, the compelling motivation is that they can present these input events to end-programmers in a much more convenient manner (as discussed in later sub-sections). The problem is that these APIs are often deprecated and even removed by the operating system vendors, with the consequence that a toolkit that worked on one version of the operating system will no longer work on the next version.

MMM and Marmots gathered input events by directly accessing the device, while MID and Colt were based on system level APIs. While they all gathered individual mouse input as delta coordinates (e.g., 1 pixel right, 2 pixels down), they did this in quite different ways. MMM directly read the input from the serial port that the mice were attached to. Marmots were phidgets that returned deltas and button clicks [Greenberg and Fitchett, 2001]. Colt took advantage of Microsoft Windows 95's Access.Bus API. MID used a DirectX API available only in Microsoft Windows '98. The Diamond Touch SDK [Dietz and Leigh, 2001] requires continuous polling of the hardware surface to obtain touch information. The task of the toolkit designer in these systems is to distinguish the multiple input streams (e.g., by device port or by handles, perhaps managed as separate threads), and to parse these as separate event streams that are eventually passed on to the end programmer. Yet, these system-level APIs proved notoriously short-lived. MMM's architecture was based on a proprietary platform, and this meant that no others could use it aside from the inventors. The DirectX and the Access.Bus API only lived within a single version of the Windows operating system, which meant that they could not be used by end-programmers who had migrated to future versions. Because Colt, MID, MMM and Marmots had architectures fundamentally entwined to these system-level APIs, none were upgraded to newer APIs that came with the next operating system. This meant that all these SDG toolkits and systems had short-lived utility.

Recommendation: The toolkit should layer system level API dependencies, where it guarantees that input from multiple devices can be passed onto higher layers as separate

streams. When new system-level API dependencies are used, the layer can be rewritten and substituted without affecting higher layers.

2.3.2 Presenting Uniquely Identified Events with Pointer Coordinates Relative to the Screen or Working Window

Input from multiple pointing devices needs to be presented in a way that is easy for end programmers to use. First, the different pointing devices and keyboards need to be somehow identified if programmers are to distinguish between their input streams. Second, end programmers are familiar with absolute pointer coordinates relative to the working window since this is how they are represented in almost all graphical user interface programming environments. The problem is that most low-level system APIs, as mentioned in Section 2.2.1, provide delta coordinates for each input pointing device. If these were passed onto to the end-programmer as is, it would significantly add to their burden of matching pointer movements with window coordinates.

Earlier systems had mixed solutions to this problem. MMM [Bier and Freeman, 1991] required a complete redevelopment of the windowing system to properly provide coordinates relative to individual windows or editors. Mice were identified simply by looking at what serial port the stream came from. It was not a toolkit and did not provide any packaging of raw mouse events through an API. Thus programmers (who were also the MMM developers) had to constantly deal with raw input streams, and it was their responsibility to transform coordinates into something that was usable. Colt [Bricker, 1998] finessed the difficulty of handling multiple mice in their Windows'95 version by using specialized input devices intended for games, e.g., joysticks. Colt made the handling of device APIs the responsibility of the end programmer, where they had to access them through an awkward data structure that returned the device ID as well as the pointer deltas through message interception. Colt did provide some help to learning this through comments and source code examples. The MID Toolkit [Bederson and Hourcade, 1999] was the first to ease the task of the end programmer. MID provided uniquely identified input events that appeared as regular java mouse events augmented with an added ID parameter to identify the mouse. While a huge advantage, these events

provided absolute screen pointer coordinates. It was left to the programmer to transform these coordinates relative to a particular window. Very recently (and after the initial deployment of our own SDG Toolkit), Smart Technologies let people access coordinates of one or two touch points on their DViT Smart Boards by automatically providing coordinates relative to the working window in their Smart Board SDK. There is no real ID, but they do identify which touch was first vs. second when multiple touches appear.

We should also mention that distributed groupware systems, such as Beach, GroupKit and MAUI, almost always present mice coordinates relative to the working window. This is because these systems only have one local mouse. They take the window-relative coordinates returned by that mouse (as captured through the standard GUI toolkit) and send these coordinates (usually identified by participant id) to the remote computers.

Recommendation: The toolkit should present a uniquely identified input API with pointer coordinates relative to the screen and/or working window.

2.3.3 Provide Multiple Cursors for Each Mouse

When developing applications that support multiple mice on a single computer, users expect a cursor to move in accordance with the movements of their mouse. These cursors need to be responsive and efficient: the application should not be compromised because of cursor movement. The first problem is that the low-level APIs do not provide cursors. The result is that end programmers spend considerable effort reinventing cursors. Often the programs developed are sluggish due to inefficient cursor drawing implementations. The second problem is that even when multiple cursors are provided, end programmers often cannot customize these cursors for different applications. This is important because like traditional cursors its shape reflects important meanings (e.g., mode information). As well, multiple cursors need to be distinguished from one another so participants know whose cursor belongs to whom [Greenberg, et al., 1996].

Amulet, MID, the Smart Technologies SDK and the Diamond Touch SDK do not automatically draw multiple cursors for the end programmer. Researchers were able to still develop interesting interactions using customized cursors although this had to be

built from the ground up. Both Colt and MMM implemented multiple cursors. Colt provided simple coloured cursors by saving the background image underneath the cursor prior to drawing the cursor on top of the background. However, when certain cooperatively controlled objects (e.g., a shared line) were dragged, the image underneath the cursor would change. As a result, old copies of the background image would remain, thus leaving "mouse trails" or unwanted pixels that should have been erased instead of saved and later redrawn. This problem would not happen if cursors were implemented as windows superimposed on top of the working application like in the MMM architecture. In MMM, each user's mouse was also represented with a fixed colour cursor. Cursors were drawn as top-level windows and updated by continually refreshing the cursor window and the windows underneath each time the mouse moved.

PebblesDraw [Myers et al., 1998] was a drawing application designed using the Amulet toolkit that provided different shaped cursors (e.g., a star, a cross, a snake, etc.) for each mouse. Attached to each cursor was a large coloured square that represented the selected colour in the drawing application. KidPad [Bederson et al., 1996] used an interaction technique that dynamically changed the cursor image to reflect the current tool used.

Both the GroupKit and MAUI toolkits allowed end programmers to develop customized cursors for each mouse. GroupKit [Roseman and Greenberg, 1995] allowed programmer to add cursors with a single line of code. However, if programmers wanted to modify the cursor properties they had to understand the existing cursor drawing code. The recent MAUI Toolkit allowed end programmers to rapidly explore different cursors by including a set of built-in graphical cursors (e.g., arrow and block). The MAUI toolkit also allowed arbitrary icons to be used as the graphical cursor.

Recommendation: The toolkit should automatically provide multiple cursors for the end-programmer. These graphical cursors should be efficient and customizable.

2.3.4 Support for Different Orientations on a Table

Research is growing in how multiple people interact with table top displays. Unlike traditional upright displays where people are all seated in front of it at the same

orientation, tables afford seating at different sides. This means that people will view and interact with it at different orientations. The problem is that computers today assume that there will be only a single orientation for all mouse input. To overcome these problems, SDG programmers of tables need to rotate incoming coordinate data for each mouse and the cursor image and captions for each cursor to reflect the seating position of its owner.

MID, MMM, Colt, Amulet, and distributed groupware systems such as GroupKit, Beach and MAUI do not support different orientations on a table. They all assume a fixed edge, usually the 'south' position in a vertical display. However, the issue of pointer orientation is finessed when the coordinates provided by the input device are absolute positions of the touch surface. For example, the Smart Board SDK (http://www.smarttech.com) and Diamond Touch SDK [Dietz and Leigh, 2001] provide coordinates relative to their absolute position on the board so there is no need for special rotation transformations as the touch coordinates returned are identical regardless of where people are seated. The catch is that these systems do not rotate the cursor image to reflect seating position and they cannot do so unless they know what orientation each user in the system is located on.

The Diamond Spin Toolkit [Shen et al., 2004] is the only one that pays some attention to orientation, although not at the level of a mouse or cursor image since it is a touch surface. It allows arbitrary orientation of graphical widgets such as documents or menus. These widgets change orientation when they were moved into particular regions of the workspace, for example, a widget located on the left side of a table would be oriented so that it was upright for the person sitting on the left.

Recommendation: The toolkit should reorient mouse input and cursor images for different seating positions of the user on a table.

2.3.5 Handling the System Cursor

Most SDG systems work by identifying multiple mice and intercepting any mouse movements and button presses. At the same time, the operating system responds to these mouse activities by merging move events, by moving the system cursor, and by passing on mouse activities to conventional widgets such as window controls, buttons and menus.

The problem is that this active system cursor competes with SDG cursors. For example, moving and clicking an SDG cursor will also move and reposition the system cursor to an arbitrary region of the screen (e.g., the application close button). This causes unexpected results when a mouse button is clicked, thus most end programmers have to write code to explicitly handle the system cursor or to mitigate its effects.

The easiest (but perhaps least practical) way around this problem is to simply avoid using the mouse as the SDG pointing device. Both Marmots and the Diamond Touch SDK do this. They do not affect the system cursor since their hardware is not recognized by the operating system as mouse compatible input devices.

The toolkit designer can also rewrite the windowing system to recognize and respond appropriately to multiple system cursors. This is the approach used by MMM [Bier and Freeman, 1991]. The catch is that it requires each window and widget to be redesigned from the ground up.

Another solution, used by both MID [Bederson and Hourcade, 1999] and Colt [Bricker, 1998], hides the system cursor and fixes its location to a spot that would not cause any unwanted operations. Unfortunately, this means that it is impossible to use conventional widgets (e.g., the close button for a regular window) as the system mouse cannot be moved to it. Because the SDG cursors cannot interact with normal widgets and window controls, MID and Colt applications are usually implemented without standard widgets, and are presented as a single window sized to cover the entire screen.

Distributed groupware toolkits such as Beach, GroupKit and MAUI avoid the system mouse problem because the local application only deals with a single mouse i.e., the system cursor and the local user's groupware cursor are one and the same. Other cursor locations are received through the network, and they are just drawn atop the groupware application. If the local person's cursor leaves the window, then the groupware cursor is usually left drawn on its edge or made invisible.

Recommendation: The toolkit should prevent the system cursor from causing unexpected results when multiple mice are used simultaneously.

2.3.6 Keyboard Focus

When a user types on the keyboard the operating system directs this input to the text area that corresponds to the current keyboard focus. The text area that has keyboard focus is usually indicated by a flashing vertical bar known as a text caret. The keyboard focus can be changed when the user mouse clicks on different text areas. The problem is that existing windowing systems only provide a single keyboard focus, thus input from multiple keyboards is directed to the same text area. To solve this problem an end programmer would need to provide a focus for each keyboard and map that focus to a particular mouse in a multi mouse application.

SDG systems have a poor record for handling this problem. Most systems do not even attempt to manage multiple keyboards: MID, Colt, the Diamond Touch SDK and the Smart Board SDK. MMM [Bier and Freeman, 1991] maps a single keyboard to two users, where they share the keyboard through a turn taking mechanism. A person could click a button to take control of the keyboard and all subsequent text input would be directed to that person's keyboard focus. While this works, it means that simultaneous input is not possible.

Distributed groupware systems such as GroupKit, Beach and MAUI automatically provide a keyboard focus for each person by using separate computers. Since there is only one keyboard per computer, it can interoperate with the widget that has the active keyboard focus. Changes within that widget are propagated to distributed counterparts, and the text is updated accordingly. However, most of these systems cannot show the text carret in the non-local widgets, as this carret is normally displayed only when that widget has the focus. Amulet [Myers, et al., 1998] is somewhat similar, except that it implements multiple keyboard foci within an SDG environment by associating one keyboard focus with each PDA. Text entered into the PDA would be directed to its corresponding keyboard focus. In all these systems the keyboard focus mapping is managed by using a separate computer for each user.

Recommendation: The toolkit should provide multiple keyboard foci, where each keyboard focus is mapped to an individual mouse.

2.3.7 SDG Widget Infrastructure

Almost all single user toolkits supply widgets that can be dropped into an application. Similarly, SDG toolkits should supply SDG widgets that recognize and support multiple people. Because SDG widgets are still novel, researchers are striving to design and evaluate appropriate widgets. Yet constructing SDG widgets is not easy. Recall that the system cursor normally communicates its events to conventional single user widgets (Section 2.2.5). The problem is that this does not work for multiple mice or keyboards, thus an alternative method is needed to allow SDG widgets to receive multiple input events. While the end programmer can write specialized code to detect and forward input events to SDG aware widgets, it is tedious to do and leads to programs that are hard to port and/or maintain. To overcome this problem, SDG programmers need a unified widget infrastructure that automatically identifies SDG widgets and forwards appropriate inputs events.

SDG widget infrastructure is mostly undeveloped in the current generation of SDG toolkits. The Diamond Touch SDK and the Smart Board SDK do not provide any infrastructure for communicating events to widgets. Widgets designed with MID are done by brute force, i.e., the programmer must build widgets without any assistance from the toolkit. Colt provided examples of how widgets could be built from the ground up, meaning that end programmers would have to copy and modify the existing Colt code. While MMM did not provide a widget infrastructure, it was possible to communicate with widgets if they were designed as windows that could receive events.

Several toolkits do provide a programmatic widget interface, although none of these are systems that deal with multiple mice. GroupKit simply allows end programmers to develop widgets by listening to input events from multiple users. While crude, quite a few interesting widgets were built. MAUI, Amulet and the Diamond Spin Toolkit provide an object-oriented widget infrastructure, where a common base class identifies and redirects input events to widgets that inherit from that class. In the Diamond Spin Toolkit, users would inherit from a `DsContainer` class that would handle basic input captured from the Diamond Touch SDK. In Amulet, each widget would contain an Interactor whose responsibility was to handle input from different PDAs. In MAUI widgets would

inherit from a GControl class and the MAUI toolkit would ensure that the appropriate input events were directed to each widget which inherited from that class.

Recommendation: The toolkit should provide an SDG widget infrastructure that handles all underlying '"plumbing"', i.e., where it automatically detects and sends input events to SDG widgets.

2.3.8 SDG Widget Primitives

The section above deals with low-level '"plumbing"' issues involved in associating events from multiple input devices to SDG widgets. Another higher-level issue is how end-programmers can conveniently develop and/or modify actual widgets. When developing a new SDG widget, most programmers begin with example (typically complex) source written by someone else (if such examples even exist) or write code that, while resembling a widget in appearance, is actually hard-wired into the application. The problem is that end programmers will spend excessive time "re-inventing the wheel" either by re-implementing basic widget code from scratch or by modifying fairly complex widget implementation code. To overcome this problem SDG programmers need a set of basic easily extensible widgets.

Similar to what was said in the previous section, MMM, MID, Beach, the Diamond Touch SDK and the Smart Board SDK do not include any example widget primitives. Thus, programmers had to develop widgets from scratch.

Several toolkits provided widget source. The Colt toolkit included a simple widget class whose source could be copied and modified to create different widgets. Colt did not support widget identification, thus programmers had to modify the original source to direct input events to particular SDG widgets. The main release of GroupKit did not provide widget primitives, although it did include source code for a number of widgets.

A few distributed groupware toolkits did include extensible widgets. An experimental version of GroupKit supported inheritance, meaning that the widgets included within it could be extended. The recent MAUI toolkit provided a basic control class, which was a simple square region that received input events. MAUI also included

a stock set of extensible widgets, but since the source is not available to other researchers these widgets can be extended but not modified.

Recommendation: The toolkit should provide a set of basic extensible widget primitives that interface with the input API

2.4 Summary

The goal of this chapter was to develop a list of requirements for building a Single Display Groupware Toolkit from existing literature. In order to accomplish this goal I first explained my motivation for exploring Single Display Groupware, then I provided some background to key input systems for SDG. Finally, I examined these systems in detail to develop a list of requirements for the SDG Toolkit.

I first explained my motivation for exploring Single Display Groupware. Here I talked about how people use computers in a collaborative setting despite the fact that the computer is not aware of this collaboration. Next, I discussed studies that show there are benefits for being able to work together on the same display. Finally, I explained how the advent of large displays naturally affords interaction by multiple users.

Next, I described styles of input for SDG systems. I began with a discussion of workarounds that people use to share a single computer among multiple people. Then, I discussed specialized input devices that were created to support multiple users. Next, I described the use of multiple mice on a single computer. Next, I discussed different ways of interacting with a large vertical display and table top display. Finally I looked at several distributed toolkits as they have design considerations that are useful for developing an SDG toolkit.

Finally, I developed a list of requirements for an SDG toolkit by examining ten existing systems for SDG. I have summarized the issues and recommendations in Table 2.1 and have provided a summary of the features offered by each system discussed in Table 2.2.

Issue	Recommendation
Low level input from multiple mice and keyboards	The toolkit should layer system level API dependencies, where it guarantees that input from multiple devices can be passed onto higher layers as separate streams. When new system-level API dependencies are used, the layer can be rewritten and substituted without affecting higher layers.
Uniquely identified pointer events with coordinates relative to the screen or working window	The toolkit should present a uniquely identified input API with coordinates relative to the screen and/or working window.
Multiple cursors for each mouse	The toolkit should automatically provide multiple cursors for the end-programmer. These graphical cursors should be efficient and customizable.
Different orientations on a table	The toolkit should reorient mouse input and cursor images for different seating positions of the user on a table.
The system cursor	The toolkit should prevent the system cursor from causing unexpected results when multiple mice are used simultaneously.
Keyboard focus	The toolkit should provide multiple keyboard foci, where each keyboard focus is mapped to an individual mouse.
SDG widget infrastructure	The toolkit should provide an SDG widget infrastructure that handles all underlying ‘“plumbing”’, i.e., where it automatically detects and sends input events to SDG widgets.
SDG widget primitives	The toolkit should provide a set of basic extensible widget primitives that interface with the input API.

Table 2.1. Recommendations for developing an SDG Toolkit

System	Mice	Key-board	Relative Window Coordinates	Multiple Cursors © = custom	Support for table orientations	Dealing with system mouse	Keyboard focus	Widget Primitives	Widget Example	Other Input
MMM	2 serial	1	No	Yes	No	N/A	Switched in software	No	Closed System	N/A
MID	256 USB	1	No	No	No	Hidden	N/A	No	Separate App	Also supports Diamond Touch
Colt	2 USB	1	No	Yes	No	Hidden	N/A	Yes	Yes	N/A
Amulet	N/A	N/A	No	Yes ©	No	N/A	Each PDA has own keyboard	No	Separate App	PDAs
Diamond Touch SDK	N/A	N/A	No	No	Yes	N/A	N/A	No	No	Diamond Touch
Diamond Spin	N/A	N/A	Yes	No	Yes	N/A	N/A	Yes	Yes	Diamond Touch
Smart Board SDK	N/A	N/A	No	No	Yes	N/A	N/A	No	No	Smart's Digital Vision Board
Beach	1	1	Yes	No	No	N/A	N/A	Yes	Separate App	Distributed
GroupKit	1	1	Yes	Yes ©	No	N/A	N/A	No	Yes	Distributed
MUG	1	1	Yes	Yes ©	No	N/A	N/A	No	Yes	Distributed
SDG Toolkit	Serial/ USB	Serial/ USB	Yes	Yes ©	Yes	Super mouse, Hidden	One per mouse	Yes	Yes	Diamond Touch, DViT, 3D Input

Table 2.2. Overview of features offered by key systems

Chapter 3. The Implementation of the SDG Toolkit[1]

In this chapter, I discuss the development of a toolkit to simplify the design of Single Display Groupware applications. This chapter continues from the requirements developed in Chapter 2, and further elaborates on how these requirements can be implemented (albeit with some tradeoffs). Collectively, these implementations form the SDG Toolkit. As we will see, the SDG Toolkit implementation is divided into seven main components. First, I will discuss the issue of gaining input from multiple devices connected to the same computer as separate streams. Second, I will discuss the transformation of pointer data to window coordinates. Third, I will explore the issue of presenting these streams of input as uniquely identified input events for programmers. Fourth, I will discuss the drawing of multiple cursors for each user. Fifth, I will discuss the implications of supporting table top and vertical displays. Sixth, I will describe the handling of the system cursor. Finally I will discuss the management of multiple keyboard foci. This chapter closes with descriptions of several basic applications; these help illustrate, summarize, and connect the SDG Toolkit implementation from the perspective of an end-programmer.

While this chapter describes what some may consider 'routine' software development, I stress it contributes much to SDG research. In particular:

---

[1] Portions of this chapter are published in:

Tse, E., Greenberg, S. (2004) **Rapidly Prototyping Single Display Groupware using the SDG Toolkit.** Proc. Fifth Australian User Interface Conference*, Conferences in Research and Practice in Information Technology (CRPIT),* Vol 28, Dunedin, New Zealand, pp. 101-110.

1. Each section articulates the basic requirements and technical challenges that face all designers of toolkits for single display groupware. This is important as it helps others understand the needs and pitfalls in SDG development *a priori* rather than after-the-fact discoveries through trial and error.
2. I detail solutions to these problems as implemented in SDG Toolkit. While our descriptions are within the context of the Microsoft Windows platform and .NET, our strategies generalize to other platforms and thus help developers of SDG toolkits.
3. The principles of the SDG Toolkit are also applicable to other novel input devices. I will discuss several extensions of the SDG Toolkit to other novel input devices in Chapter 5.
4. I describe how end programmers would process and use SDG input events. This is important as it supplies a conceptual model to other toolkit builders about how a toolkit for SDG should present itself.
5. I provide SDG Toolkit as a fully documented downloadable resource for other SDG researchers so they can immediately begin rapidly prototyping their ideas.

As I introduce the various principles and implementation details of the SDG Toolkit, I will refer to Figure 3.1, which represents the overall implementation architecture of the toolkit. It shows how the SDG Toolkit converts low level Raw Input Data (bottom) into a form that is easy for the end programmer to use (top).

3.1 Low Level Input from Multiple Mice and Keyboards

As discussed in Chapter 2, the one user per computer assumption in current windowing systems prevents programmers from easily accessing input from multiple mice and keyboards as separate streams. Operating systems sometimes provide low level APIs to access multiple mice as separate streams but they are difficult to program and often change in ways that are not backwards compatible when the operating system is updated.

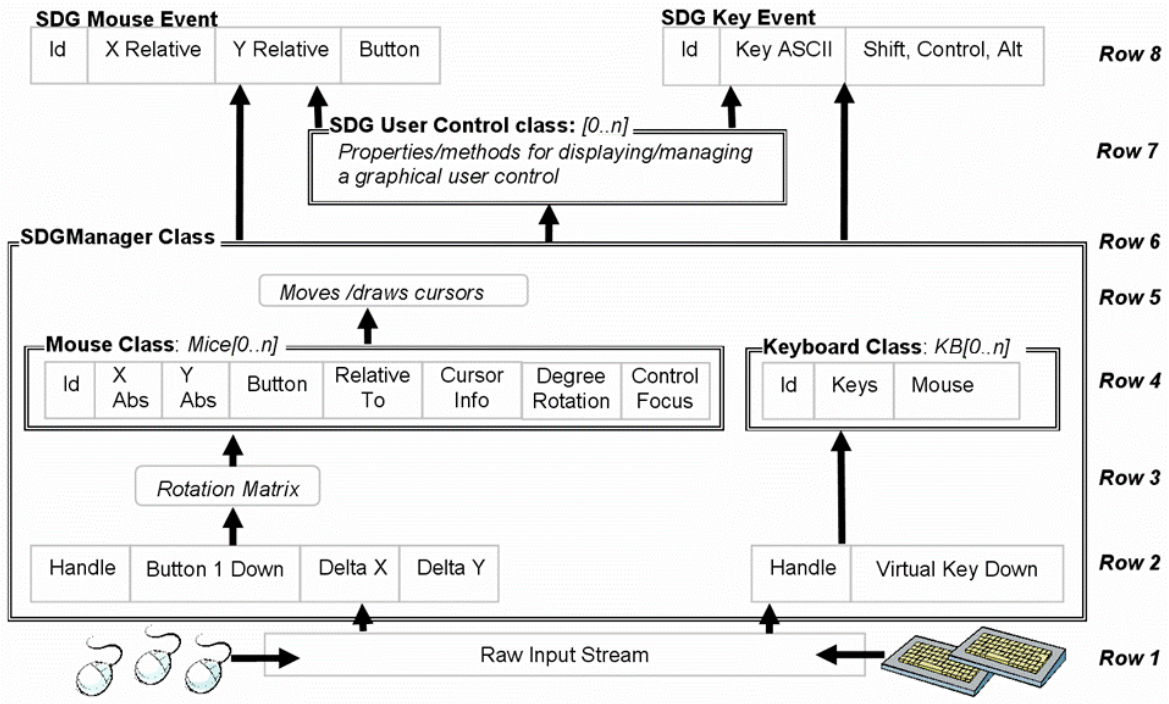


Figure 3.1. The SDG class and event structure, showing how Raw Input turns into SDG events

This is why neither the Colt [Bricker, 1998] nor MID Toolkit [Bederson and Hourcade, 1999] are useable today.

To insulate against these low level dependencies, an SDG toolkit should isolate the input layer so that it is accessible only through a well defined application programmer's interface. When new system-level API dependencies are used, the layer can be rewritten and substituted without affecting the code written by end programmers. The remainder of this section describes how all this is done by the SDG Toolkit.

At the lowest level the SDG Toolkit currently uses *Raw Input*, a somewhat difficult-to-program utility for low-level input management in Microsoft Windows XP, as illustrated by the following example. A programmer can query Raw Input to gain a list of all attached input devices. On any keyboard or mouse input, Raw Input then provides a generic input stream. The programmer must parse this stream to identify the device that generated the input and its particular arguments. For example, the source code below illustrates how a Raw Input event stream is captured and analyzed.

```
LRESULT CALLBACK WndProc(UINT msg, WPARAM wParam, LPARAM lParam){
if (msg == WM_INPUT) { //A Raw Input Event is identified
     RAWINPUT ri; //The Raw Input Data Structure
     GetRawInputData(lParam…); //lParam points to an input structure
     if(RIM_TYPEMOUSE == ri.Header.dwType) { //input device type
       int mouseHandle = ri.Header.hDevice; //32 bit handle to a mouse
       int xDelta = ri.mouse.lLastX; //Distance from last mouse event
       int yDelta = ri.mouse.lLastY;
}}}
```

Figure 3.2. A Raw Input window event

Each event is tagged by a data structure (`RAWINPUT ri`) identifying the input port, the input device type (e.g., mouse, keyboard), and its parameters (e.g., x and y delta coordinates).

The SDG Toolkit uses Raw Input as the lowest level building block for retrieving input from multiple keyboards and mice as illustrated in Row 1 of Figure 3.1. It hides the fact that it uses Raw Input within a layer, where the SDG Manager class (the box contained between Rows 2 - 6) captures, transforms and wraps data from the Raw Input stream into a form more convenient to the end programmer (Rows 2 - 4). When the programmer creates the SDG Manager instance, the instance automatically queries Raw Input (Row 1) to discover the attached mice and keyboards. The SDG Manager then automatically creates one or more instances of the SDG Mouse and Keyboard classes (Row 4), each matched to a particular input device by storing its handle (Row 2). Finally, the SDG Manager monitors the incoming Raw Input stream (Row 1), and parses its information (operation in Row 2), where it stores the mice/keyboard data in the appropriate Mouse and Keyboard instances (Row 4). Details of how this data is transformed will be discussed in later sections. Note that this is a general strategy: the same approach can and has been used to extend the SDG Toolkit to handle other types of input devices.

Furthermore, the SDG Manager maintains a collection of all Mouse and Keyboard instances so that the programmer can query and modify this collection at any time. For example, the programmer can easily find out how many mice are attached to the computer, enumerate through them, and set their initial coordinates:

```
// Set all initial mice positions to (0,0)
  foreach (Mouse this_mouse in sdgMgr.Mice) {
     this_mouse.X = 0; this_mouse.Y = 0;
  }
```

In summary, the SDG Manager is responsible for managing all input devices. Using system level facilities, it detects what devices are available. It captures and transforms each input stream and encapsulates it in object that represents that device. The SDG Manager hides any system dependencies, and provides a reasonable mechanism for substituting other system level facilities if needed.

3.2 Translating Pointer Data to Window Coordinates

Low level mouse pointer input APIs usually deliver delta values that are relative to the last mouse movement. Yet windowing environments usually work with either absolute screen coordinates, or (more preferably) with absolute coordinates that are relative to the to left corner of an application window. To do this we need to first convert the delta coordinates into absolute screen coordinates. While useful for full screen applications, it does not suffice for windowed applications. For example, if absolute screen coordinates were fed into a standard line drawing function within a window the pointer would be

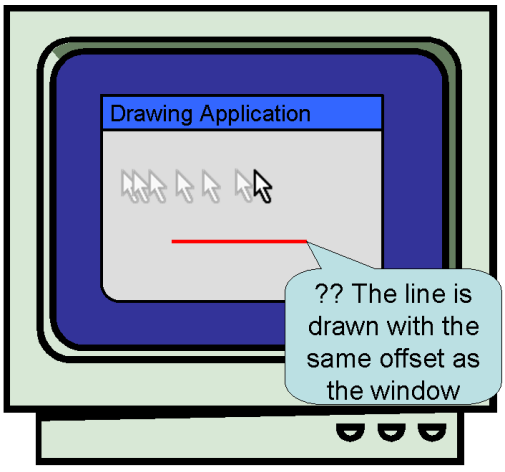


Figure 3.3. The issue of drawing in absolute screen coordinates: Drawings are offset since programs expect to be working in coordinates relative to the top left of the window. The problem is that we do not know the context of where our SDG events occur.

positioned with an offset as seen in Figure 3.3. Consequently, we need to further transform the absolute screen coordinates so they are relative to the top left corner of the application window.

The SDG Toolkit does both these transformations. By default, the SDG Toolkit transforms Raw Input delta values into absolute screen coordinates that are stored in the X abs and Y abs properties of the SDG Mouse Instance (Figure 3.1, Row 4). With Raw Input this is done by adding the latest X and Y delta values to the current mouse position. Unless otherwise instructed, the SDG Manager uses these screen coordinates when it raises an SDG Mouse Event, which is useful if the programmer is creating a full-screen application.

SDG Toolkit programmers can explicitly associate mouse instances to both standard windows and controls. Specifically, they can set the mouse class's `RelativeTo` property to the desired window/widget. Or, a global `RelativeTo` property makes all mice coordinates relative to a particular window. When an event is to be fired the absolute screen coordinates are subtracted from the top left corner to the form specified in the `RelativeTo` property. Because the SDG Manager does the coordinate transformation on the fly at run time, the `RelativeTo` property can be changed any time during program execution. For example `SdgManager1.Mice[0].RelativeTo = Form;` instructs the SDG Manager to translate and return mouse coordinates relative to the form window for that the first mouse (Figure 3.1, Row 4; see Mouse Class).

In Chapter 4, I will describe how our SDG User Control class defines controls that receive events from the SDG Manager, and how these controls automatically translate the event screen coordinates to control-relative coordinates.

In summary, the SDG Toolkit transforms pointer data received as delta coordinates into absolute coordinates relative to either the screen or the application window specified by the end programmer.

3.3 Presenting Uniquely Identified Input Events

When a programmer receives an input event from the toolkit, he or she needs to know which mouse or keyboard generated that event. Since traditional mouse and keyboard event handlers do not provide this information, the toolkit needs to present identifier information in a way that is easy for the end programmer to understand. We saw in Chapter 2 that one approach used by MID [Bederson and Hourcade, 1999] was to provide an ID parameter in the standard mouse event, where multiple mice were identified with ordinal integers starting at 0. End programmers could call a function to obtain the total number of mice connected to the system.

A similar method has been used in the implementation of the SDG Toolkit. The SDG Manager generates ID's for each device instance as ordinal integers starting at 0, and stores any associated data with that instance. This means that programmers can use this ID to index the SDG Manager's Mouse and Keyboard collection, where they can easily query or set the properties of a particular instance. For example, a programmer can display the coordinates of the first mouse by:

```
MessageBox.Show (sdgMgr.Mice[0].X + "," + sdgMgr.Mice[0].Y);
```

Whenever new input is detected from any mouse or keyboard, the SDG Manager raises an *SDG Mouse Event* or *SDG Key Event*. This is presented to programmers in a style that mimics conventional single user mouse and keyboard events (Figure 3.1, Row 8). For example, SDG Mouse Events follow the standard `MouseDown`, `MouseUp`, `MouseMove` and `MouseClick` naming conventions, and contain all of the expected parameters typically found in standard mouse events, e.g., X and Y coordinates, button state, and so on. Similarly, the SDG Key Events include `KeyUp`, `KeyDown` and `KeyPress`. The most important difference from standard events is that SDG events include the ID parameter (Row 8). The result is that programmers can create event handlers that easily identify the mouse or keyboard that fired the event.

To show how this works, Figure 3.4 compares how a C# programmer would register and write a standard non-SDG mouse event handler (Figure 3.4 top) versus an SDG Toolkit mouse event handler (Figure 3.4 bottom). While examples are in C#, the

```
// a traditional mouse event
Form.MouseDown += new MouseEventHandler(OnMouseDown);
…
OnMouseDown (object sender, MouseEventArgs e){
    mbox.show ("X,Y,button is: " + e.X + e.Y + e.Button);
}
```

```
// an SDG mouse event – differences are bolded
sdgMgr.MouseDown += new MouseEventHandler(OnMouseDown);
…
OnMouseDown (object sender, SdgMouseEventArgs e){
    mbox.show ("ID, X, Y, button is:" + e.ID + e.X + e.Y + e.Button);
}
```

Figure 3.4. Comparing traditional and SDG mouse events

SDG Toolkit works with any .NET language e.g., Visual Basic, Managed C++ and so on. As seen, they are quite similar in style. The important differences are the inclusion of a mouse ID, the different typing of the event argument (`SdgMouseEventArgs e`) and that the SDG Manager generated the event (`sdgMgr.MouseDown`) instead of the window (`Form.MouseDown`).

A simple yet inefficient way to deliver mouse movement events is raise an event each time a Raw Input callback occurs. The problem with this approach is that thousands of callbacks can be received each second and since many programmers write the bulk of their code in a mouse movement event handler therefore this can compromise application performance. This problem is exacerbated when multiple mice are moved simultaneously. For example, moving two mice simultaneously doubles the number of Raw Input callbacks.

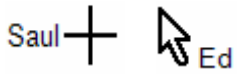
To resolve this issue, the SDG Toolkit ensures that no more than 120 mouse movement events are fired each second regardless of the number of mice connected to the system. Mouse movement events are queued and cumulatively added together when the event is to be fired. For example, if the queue contains two mouse movements – one that moves the mouse right by one pixel and another that moves the mouse right by two pixels – the mouse movement event would see that the mouse was moved three pixels to the right. The advantage of this method is that mouse movement events occur less frequently, although they often have larger coordinate changes. Also, this system maintains application performance even when multiple mice are connected to the system.

In summary, the SDG Toolkit offers the programmer two methods for accessing uniquely identified input events. First, the SDG Toolkit provides a Mouse property that can be polled to obtain the current X and Y coordinates of the mouse and a Key property that can be polled to obtain the last character typed. Second, mouse and keyboard events are automatically generated in a style that is familiar to programmers. However, the event mechanism adds an additional ID parameter that allows the programmers to uniquely identify the device that generated it. The SDG Toolkit queues up mouse movement events to improve application performance even when multiple mice are connected.

3.4 Displaying Multiple Cursors

People using conventional windowing systems expect a cursor to move in accordance with the movements of their mouse. SDG systems should provide support for this familiar interaction. To accomplish this, an SDG toolkit must automatically provide multiple cursors for the end-programmer. These graphical cursors should not compromise application performance. In Chapter 2, we saw how GroupKit provided cursors that supported different images and text captions. In this section I will describe how cursors are drawn in the SDG Toolkit, how these cursors are customizable, and how cursor drawing has been made efficient.

By default, every pointing device seen by the SDG Toolkit displays an associated cursor. No extra end-programming is needed to get basic multiple cursors. The SDG Manager implements this (Figure 3.1, Row 5) by leveraging the capabilities of top-level transparent windows, where one is created for each Mouse instance. The SDG Manager draws the cursor within this window, and repositions the window after the mouse is moved to the correct position. As long as cursors are of modest size (i.e., smaller than a 64 by 64 pixel icon), they perform well, especially if the computer uses video cards that process transparent windows in hardware.




```
SDG Manager1.Mice[0].Cursor = Cursors.Cross;
SDG Manager1.Mice[0].Text   = “Saul”;
SDG Manager1.Mice[0].TextCardinalPosition = West
SDG Manager1.Mice[1].Cursor = Cursors.Arrow;
SDG Manager1.Mice[1].Text   = “Ed”;
```

Figure 3.5. Customizing cursor images and labels

SDG Toolkit cursors are also highly customizable. The programmer can set the various cursor properties contained in each mouse instance as stylized in the cursor info property of Figure 3.1, Row 4. In reality, cursor info includes the cursor shape, its hot spot or target area of the cursor (e.g., the hot spot of a cross cursor is in the centre and the hot spot for a regular arrow cursor is on the top left of the cursor), its visibility, and even its transparency. The programmer can also add a text label to the cursor, and can adjust the text font, size, color and location relative to the cursor graphic. For example, the code snippet in Figure 3.5, creates two visually distinctive cursors identified by their owner’s name, shown at the top.

The rapid movement of windows requires a large amount of processing power from computers, which do not support hardware acceleration of moving partially transparent windows. To alleviate this problem we use the event monitoring system mentioned in Section 3.3 and only move the cursor when a mouse movement event is fired. This method significantly reduces the amount of processor time spent moving windows around the screen, and because of the monitoring system, the cursor moves in a smooth manner.

In summary, the SDG Toolkit displays a cursor for each connected mouse. Both the text caption and the cursor image can be customized by the end programmer. To ensure that cursor movement does not compromise application performance, the SDG Toolkit only moves the cursor window when a mouse movement event is to be fired.

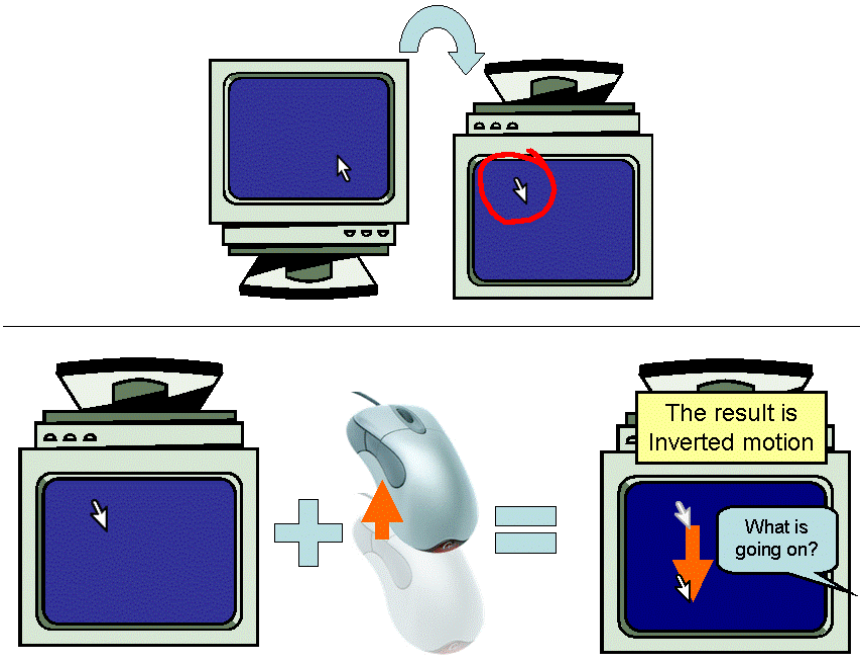


Figure 3.6. Issues with rotation: When a monitor is upside down and the mouse moves up, motion is inverted.

3.5 Supporting Different Orientations on a Table

We saw in Chapter 2 that computers today assume that there will be only a single orientation for all mouse input. This causes two problems for developers of table top SDG applications using multiple mice. First, when working on opposite ends of a table, one user is located on the upright position of the display and the other is located on an inverted position. Figure 3.6 shows that when the person in the inverted position moves his or her mouse upward the result is that the cursor instead moves downward. The second problem is that cursor images and text labels are incorrectly oriented for the inverted participant's cursor. This makes it difficult to read one's own text caption and it makes interaction awkward since we do not normally work with inverted cursors.

The SDG Manager solves the first problem by transforming the deltas produced by Raw Input through a rotation matrix (Figure 3.1, Row 3) whose angle is specified by the programmer. The programmer sets this orientation angle using the Mouse instance's `DegreeRotation` property (Figure 3.1, Row 4). To solve the second problem, the cursors and movement are adjusted accordingly to give the cursor the correct look and the mouse

the correct feel. Thus, the SDG Toolkit also adjusts the rotated cursor image so that it appears in the correct orientation for the inverted user. For example, if one person is sitting across from the other, the code in Figure 3.7 could be used to correctly orient the inverted participant's cursor.

```
SDG Manager1.Mice[1].Text = "Ed";
SDG Manager1.Mice[1].DegreeRotation = 180;
```

Figure 3.7. Setting the orientation of a cursor

The cursor and text caption would be flipped 180 degrees, and cursor movements would be inverted.

In summary, the SDG Toolkit simplifies table top programming by automatically rotating incoming mouse movement information to a specific orientation specified by the end programmer. The SDG Toolkit also reorients graphical cursors so they are upright for the person's orientation around the table.

3.6 Handling the System Cursor

We saw in Chapter 2 how if all SDG mice move the system cursor, it will not track correctly (as it reacts to the combined forces on it). The system cursor moves around the screen in strange ways. While one could make this cursor invisible, it is still active, i.e., a click with any SDG mouse will also generate a click event on the system cursor: this could mysteriously activate the window or widget under the system mouse or unintentionally close an application. This leads to problems for SDG developers as they need to find ways to circumvent the system mouse. I forewarn that there are no elegant solutions. Instead, I list various approaches one could take and show how each mitigates problems caused by the system mouse, albeit with some tradeoffs.

One possible solution is to continuously move the system mouse to the location of the most recently used SDG mouse, i.e., to give the momentary illusion that any SDG mouse could control a non-SDG window or control. Unfortunately, this does not work well in practice. Time and location dependencies in how a system mouse interpreted

concurrent click/move/release actions generated by multiple mice meant that one user's mouse action could easily interfere with another user's mouse action.

A much better solution is to bind the system mouse to directly follow a single SDG mouse and its cursor. This 'super mouse' will have both SDG and standard capabilities. While not a democratic solution, it is pragmatic. This solution is implemented by SDG Toolkit, where the programmer can ask the SDG Manager to bind the system mouse to a single SDG mouse, for example:

```
SDGMgr.MouseToFollow = 1 // follow 1st mouse
```

However, a serious side effect of having an enabled system mouse results from windowing systems maintaining only a single active window as the input focus. A super mouse click outside the SDG window causes the system mouse to raise a non-SDG window and the SDG application to lose the input focus. Other SDG mice will no longer respond. Still, it is a reasonable approach for full screen applications.

To remove this side effect, one can 'turn off' the system mouse. Programmers can do this with the SDG Toolkit by ensuring that the system mouse never moves from some unused corner of a window and by making it invisible; they do this by setting the `ParkSystemMouseLocation` property of the SDG Manager to any screen location. While excellent for managing pure SDG applications, it does mean that the end user cannot use standard window controls (close, resize), any standard widgets (buttons, scrollbars), or switch to other non-SDG windows. This can be confusing because people's naïve conceptual model is that their cursor represents both an SDG mouse and a system mouse.

Yet it would be convenient for programmers to exploit non-SDG widget capabilities with a parked mouse. Programmers could do this manually by examining if any SDG mouse event occurs over a widget and programmatically calling the widgets callback function. The SDG toolkit simplifies this task by telling the event handler what widget appears under it in each SDG mouse event. That is, whenever the SDG Manager sees a mouse event, it examines what user control (if any) is immediately under that coordinate position. It then returns it as the `sender` argument to the event handler, e.g., as shown below:

```
OnMouseDown(object sender, SdgMouseEventArgs e);
```

Of course, this does not completely solve the problem as it remains the programmer's responsibility to activate a particular widget's functions. For example, if a user clicked over a non-SDG button then the programmer could identify this button in the `sender` argument and use it to interpret the event within the context of the button. However, the programmer would have to somehow activate the button—its graphical behaviour and its callback—as the button had never received this event.

The choice between the solutions implemented by SDG Manager—the single 'super mouse', mouse parking, using the sender argument—is a trade off between the desired nuances of the SDG application and its effect on the end user.

In summary, the SDG Toolkit provides three strategies for handling the system mouse. First, it allows end programmers to specify a 'super mouse' that has control of the system cursor. Second, it allows the system mouse to be parked in a particular location and hidden from the view of the end user. Third, if the parked mode is used the end programmer can activate the callback of activated widgets by examining the sender argument of the SDG mouse event.

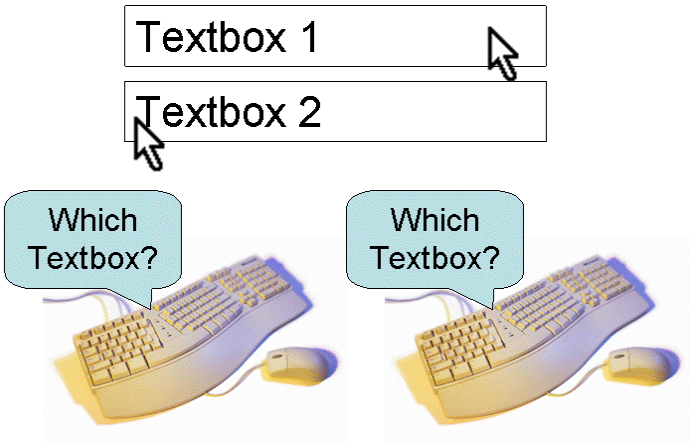


Figure 3.8. An issue with multiple keyboards: It is unknown which textbox keyboard input should go

3.7 Keyboard Focus

As seen in Chapter 2, existing windowing systems only provide a single keyboard focus. Thus, if there are two or more keyboards connected to a computer, the windowing system will not know which text widget should receive keyboard input. An example is seen in Figure 3.8. In SDG applications there needs to be multiple text foci so that input from multiple keyboards can be directed to appropriate text widgets.

The SDG Toolkit associates multiple text foci for all keyboards and mice as follows: First, it associates each keyboard with a mouse. Second, when a user mouse-clicks over a control to indicate their text input focus, the Mouse instance automatically stores a pointer to that control in its `ControlFocus` property (Figure 3.1, Row 4). Third, the programmer writes a keyboard key event handler that just looks up the `ControlFocus` of its corresponding mouse and directs the text towards that control.

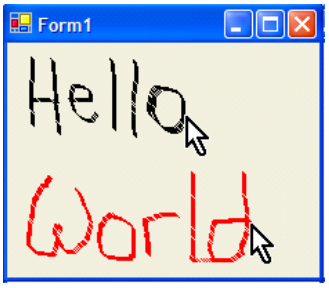
By default, Keyboard 0 is automatically mapped to Mouse 0, Keyboard 1 to Mouse 1 and so forth. Programmers can customize this mapping by changing the 'mouse' property of the keyboard instance, for example, `SDGMgr.Keyboard(5).Mouse = 0` causes the sixth keyboard to track the first mouse, where keyboards are enumerated starting with the number zero.

In summary, the SDG Toolkit provides a focus for each keyboard connected to the computer. The keyboard focus can be changed by clicking on a widget with the automatically mapped mouse. This mapping can be modified by the end programmer, so different keyboards can be mapped to different mouse foci.

3.8 What the Programmer Sees

This section illustrates how a programmer would actually create an SDG application using the SDG Toolkit. For clarity, these examples are deliberately simple to minimize non-SDG complexity. Code excludes setup and "housekeeping code" standard to all SDG and non-SDG Windows programs in C#.

3.8.1 Hello World – Mouse Drawing




```
1  private void InitializeComponent () {
//'this' refers to the top level window form
2   Form1.sdgMgr.RelativeTo = this;
3   this.sdgMgr.MouseMove +=
          new SdgMouseEventHandler(this.sdgMgr_MouseMove);
4    …
5  }
6  private void sdgMgr_MouseMove(object sender, SdgMouseEventArgs e) {
7    Graphics g = this.CreateGraphics();
8    Pen penColour = Pens.Black;
9    if (e.ID > 0) penColour = Pens.Red;
10   if ((e.Button & MouseButtons.Left) > 0)
11     g.DrawLine(penColour, new Point(e.X-1, e.Y-1),
                             new Point(e.X+1, e.Y+1));
12 }
```

Figure 3.9. SDG hello world drawing: 'Hello' is in black, 'World' is in red.

The first "hello world" example is a very simple concurrent drawing application involving two users and two mice, illustrated in Figure 3.9. It illustrates how the SDG manager is included and SDG mouse events are handled via callbacks. To build this, the programmer takes the following steps.

1. *Add the SDG Manager using the Visual Studio interface builder to make the program SDG aware*. Drag an SDG Manager component from the Visual Studio toolbox onto the application. The SDG Manager is implemented as a non-visible control used by the programmer in exactly the same way as other standard controls. One adds it to a window by drag and drop, sets its many properties and event handlers through form-filling, and handles events in the normal way

2. *Make all mouse coordinates relative to the window*. In the standard `InitializeComponent` routine (Figure 3.9, line 1) that initializes the top level window, add a line of code that first sets the `relativeTo` property of the SDG Manager to the form (line 2). This can also be done via the properties window.

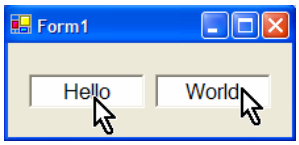
3. *Register an event handler to the SDG MouseMove event* (line 3). One can type this into the program or, alternatively, one can set the event handler without coding by using the SDG Manager's property window.

4. *Write the callback for the `sdgMgr_MouseMove` event* (lines 6-12). If the previous step was done through the properties dialog, lines 6 and 12 of the callback would have been automatically added to the program. Create a black drawing pen (line

8), but change its color to red if the Mouse ID is greater than 0, i.e., if it's not the first mouse (line 9). The programmer then checks to see if the left button is depressed for that mouse (line 10), and if so draw a 2x2 pixel around the current X and Y coordinates of the mouse (line 11).

These few lines of code illustrate the simplicity of the SDG Toolkit. In contrast, building the same program without the SDG Toolkit (atop of Raw Input) required 588 lines of complex code!

3.8.2 Hello World – Keyboard Text



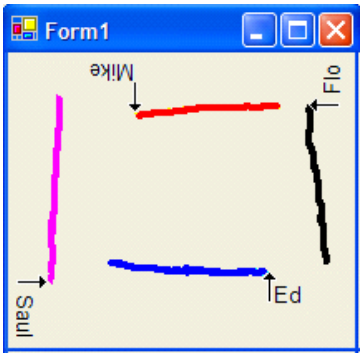

```
1  private void sdgMgr_KeyPress(object sender,
                                SdgKeyPressEventArgs e){
2    TextBox this_textbox;
3    if (sdgMgr.Mice[e.ID].ControlFocus is TextBox) {
4      this_textbox = (TextBox) sdgMgr.Mice[e.ID].ControlFocus;
5      this_textbox.Text = this_textbox.Text +
                                    e.KeyChar.ToString();
6  }}
```

Figure 3.10. SDG hello world keyboard application

The second 'hello world' example has two textboxes and also works with two people (Figure 3.10). When a user clicks on a textbox, that user's keyboard input will be redirected. If two people click different textboxes as in Figure 3.10, their typing will be directed appropriately (even if they type simultaneously). If both click the same text box, their input is merged.

The code in Figure 3.10 shows only the `KeyPress` event handler, which illustrates how one associates `KeyPress` events from multiple keyboards to the different text widget foci. Adding the SDG manager and registering the callback is similar to the previous example. Section 3.7 describes how each mouse remembers what control it last clicked (the focus) in its `ControlFocus` property. When the `KeyPress` event is raised from either keyboard, the event handler (lines 1-6) looks to see if that mouse has a Textbox control in its ControlFocus property. If so, the event handler adds a character to the Textbox (line 5).

3.8.3 Table top Drawing




```
1 Public Form1 (){  // The Form constructor
2   String[] sdgText = {"Ed", "Saul", "Mike", "Flo"};
3   int[]    sdgDegreeRotations = {0, 90, 180, 270};
4   for (int i=0; i < sdgMgr.Mice.Count && i < 4; ++i) {
5     sdgMgr.Mice[i].Cursor = Cursors.UpArrow;
6     sdgMgr.Mice[i].Text = sdgText[i];
7     sdgMgr.Mice[i].DegreeRotation = sdgDegreeRotations[i];
8   }}
9 private void sdgMgr_MouseMove(object sender,
                                  SdgMouseEventArgs e) {
10  Graphics g = this.CreateGraphics();
11  Color[] colors = {Color.Blue, Color.Magenta,
                      Color.Red, Color.Black};
12  if((int)(e.Button & MouseButtons.Left) > 0) {
13     Graphics g = this.CreateGraphics();
14     g.DrawLine(new Pen(colors[e.ID]),
15                new Point(e.X-1, e.Y-1),
                   new Point (e.X+1, e.Y+1));
16  }
```

Figure 3.11. SDG table top drawing. All user marks are in different colors

The third example illustrates a drawing application designed for a square table top with four seated people, one per side. As Figure 3.11 shows, cursors and text labels are oriented appropriately. What is not visible is that each person's mouse will also behave correctly given their seating orientation.

The initialization code shows how the programmer deals with an unknown number of mice (up to 4 in this example – line 4), sets mouse properties such as cursors and their text labels (lines 5–6), and correctly orients the cursors and returned coordinates (line 7). The MouseMove event handler (lines 9-16) is similar to Figure 3.9, except that it shows a better way to assign different line colors to each user.

3.9 Conclusion

This chapter summarized the basic implementation of a toolkit for supporting Single Display Groupware applications. It provided an implementation for the requirements developed in Chapter 2. I started by explaining the seven main components that form the core implementation of the SDG Toolkit. I have summarized the issues and solutions presented in this chapter in Table 3.1.

Next, I described three very simple example applications that illustrate how these solutions work in practice. The first drawing example showed how input events are used

by the end programmer. The second text example showed how multiple keyboard foci were supported by the toolkit. Finally, the third example showed how cursors could be customized and reoriented for different positions around a table.

The following chapter discusses the implementation of the widget layer of the SDG Toolkit.

Issue	Solution
Input from multiple mice and keyboards needs to be accessed as separate streams	Using system level facilities, the SDG Toolkit detects what devices are available. It captures and transforms data from each input stream and encapsulates it in an object that represents the device. This method hides any system dependencies, and provides a reasonable mechanism for substituting other system level facilities if needed.
Pointer Data needs to be converted into window coordinates	The SDG Toolkit transforms pointer data received as delta coordinates into absolute coordinates relative to either the screen or the application window specified by the end programmer.
Presenting Uniquely Identified Input Events to the end programmer	The SDG Toolkit offers the programmer two methods for accessing uniquely identified input events. First, the SDG Toolkit provides a Mouse property that can be polled to obtain the current X and Y coordinates of the mouse and a Key property that can be polled to obtain the last character typed. Second, mouse and keyboard events are automatically generated in a style that is similar to how programmers handle input events. However, the event mechanism adds an additional ID parameter that allows the programmers to uniquely identify the device that generated it. The SDG Toolkit queues up mouse movement events to improve application performance even when multiple mice are connected.
Multiple cursors need to be displayed for each mouse	The SDG Toolkit displays a cursor for each connected mouse. Both the text caption and the cursor image can be customized by the end programmer. To ensure that cursor movement does not compromise application performance, the SDG Toolkit only moves the cursor window when a mouse movement event is to be fired.

Different orientations on a table need to be supported	The SDG Toolkit simplifies table top programming by automatically rotating incoming mouse movement information to a specific orientation specified by the end programmer. The SDG Toolkit also reorients graphical cursors so they are upright for the person's orientation around the table.
The system cursor can cause unexpected results in SDG applications	The SDG Toolkit provides two strategies for handling the system mouse. First, it allows end programmers to specify a 'super mouse' that has control of the system mouse. Second, it allows the system mouse to be parked in a particular location and hidden from the view of the end user. In this mode the end programmer relies on the sender argument of the SDG mouse to determine if a widget was activated
Multiple keyboard foci are not provided by windowing systems	The SDG Toolkit provides a focus for each keyboard connected to the computer. The keyboard focus can be changed by clicking on a widget with the automatically mapped mouse. This mapping can be modified by the end programmer, that is different keyboards can be mapped to different mouse foci.

Table 3.1. Overview of SDG development issues and solutions implemented in the SDG Toolkit

Chapter 4. The Implementation of the SDG Widget Layer[2]

In this chapter, I discuss the development of a layer that supports the creation of reusable interface widgets (known as controls in our implementation environment), which in turn supports the rapid prototyping of SDG applications. I continue from the requirements developed in Chapter 2, and elaborate how these requirements are implemented in the SDG Toolkit. As we will see there are two main components to the implementation of the SDG widget layer. First, an SDG widget infrastructure supports the identification of SDG controls and forwards the appropriate SDG events to the appropriate control. Second, SDG widget primitives provide the basic building blocks that end programmers can extend when developing their own SDG controls. To show how all this works in practice, I include two illustrative examples of how an end programmer would develop and use SDG controls.

4.1 The SDG Widget Infrastructure

The widget infrastructure in current windowing systems is built around the notion that controls only respond to a single pointer and keyboard. Since this is not the case for SDG, we need a new widget infrastructure to support multiple input devices. That is, we need a way for SDG input to be collected and communicated to the appropriate SDG control. To do this, the infrastructure must first provide a way to discover what controls

[2] Portions of this chapter are published in:

Tse, E., Greenberg, S. (2004) **Rapidly Prototyping Single Display Groupware using the SDG Toolkit.** Proc. Fifth Australian User Interface Conference*, Conferences in Research and Practice in Information Technology (CRPIT),* Vol 28, Dunedin, New Zealand, pp. 101-110.

on a window (or form) are SDG capable. The infrastructure must then compose a list of all SDG capable controls on a form. Finally, the infrastructure must take input from multiple devices and direct it to the appropriate SDG control on this list.

The infrastructure needs a standard way to communicate input events to a particular control. The widget layer of the SDG Toolkit does this by creating two interfaces that define the minimum set of capabilities that any SDG control must understand. The first `ISdgMouseWidget` interface (whose implementation will be discussed in the following section) defines the mouse capabilities, where we insist that any SDG control object must implement methods (with arguments) corresponding to the four normal SDG mouse events described in the previous chapter, e.g., `OnSdgMouseDown`, `OnSdgMouseMove`, `OnSdgMouseUp`, `OnSdgMouseClick`. For example:

```
void OnSdgMouseMove(SdgMouseEventArgs e); //An SDG method
```

The second `ISdgMouseAndKeyWidget` interface extends this interface to include the keyboard events `OnSdgKeyDown`, `OnSdgKeyPress`, and `OnSdgKeyUp`. For example:

```
void OnSdgKeyDown(SdgKeyEventArgs e); //An SDG keyboard method
```

If graphical controls contain one of these interfaces, then the SDG Manager can exploit it to make the control SDG-aware by invoking the appropriate events through this interface.

Next, we need a way of discovering what controls on a form are SDG-capable, for this information is needed before we can decide how to pass the input to them. A naïve approach to detecting SDG controls in an application would be to look at all the immediate children of the top level window and see if they implement the ISdgMouseWidget or ISdgMouseAndKeyWidget interfaces. The problem with this approach is that SDG widgets may be embedded within a non-SDG control, and these would not be detected. To solve this problem the SDG Manager runs a recursive algorithm to parse through all of the controls when the application is initialized. The algorithm is described in the code segment in Figure 4.1, which produces a list of SDG controls in a variable called SdgWidgetList

```
private void WalkControlHierarchy(ArrayList SdgWidgetList, Control c) {
  if (c is ISdgMouseWidget || c is ISdgMouseAndKeyWidget) {
    SdgWidgetList.Add(c); //SDG Control found
  } else { //An SDG widget may be found in the children of this control
    foreach(Control child in c.Controls)
    { //c.Controls contains all the children of the current control
      WalkControlHierarchy(SdgWidgetList, child); //Recurse children
    }
  }
}
```

Figure 4.1. Recursive algorithm for finding SDG controls in a form

The third and final step is to take the input and direct it to the appropriate SDG control. For mice input, each time an SDG mouse event is received it is checked against the SdgWidgetList to see if the event occurred over a particular SDG control. If it does, then the appropriate callback is invoked in that control as defined in the ISdgMouseWidget interface. For example, when we receive a mouse down event we check to see if the coordinates of this event occur over an SDG control in the SdgWidgetList, and when a match is found the SDG Manager calls the corresponding OnSdgMouseDown event within that control using coordinates that are relative to the top left corner of the widget (whose location is stored in the SdgWidgetList).

Sending keyboard events to SDG controls is a slightly more complex procedure. In Chapter 3, I discussed two aspects of providing multiple keyboard foci in an SDG application. First, a `ControlFocus` property is associated with each mouse connected to the computer. This property points to the control that has the focus i.e., the most recently clicked SDG control that included the ISdgMouseAndKeyWidget interface. Second, each keyboard is associated with one of the multiple mice, and that mouse's control focus is the target of SDG Keyboard widget events. Figure 4.2 illustrates this in a UML activity diagram [Rumbaugh, et al., 2004]. When a keyboard event is received (Row 1) the SDG Manager checks to see if a mouse has been associated with that keyboard (Row 2). If no mouse is associated, then no keyboard event is fired within the control, instead a standard keyboard event is fired within the SDG Manager (Row 4). If a mouse is associated with the keyboard, the SDG Manager will fire a keyboard event within the control that currently has the focus of that mouse (Row 3) and then fire a keyboard event within the SDG Manager (Row 4).

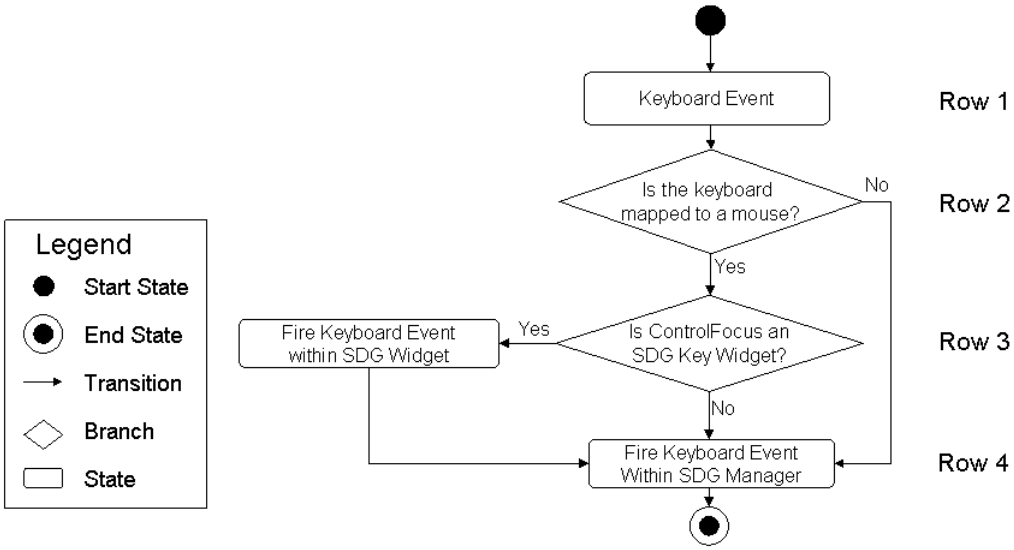


Figure 4.2. UML activity diagram of SDG keyboard events

In summary, the SDG Toolkit provides an infrastructure for rapidly prototyping SDG widgets in three steps. First, it provides a standard way of communicating input events to controls by having them inplement the ISdgMouseWidget and ISdgMouseAndKeyWidget interfaces. Second, it obtains a list of controls that implement these SDG widget interfaces by recursively searching through all controls on a form. Finally, the SDG Toolkit takes input from multiple mice and keyboards and directs it to the appropriate SDG control.

4.2 SDG Widget Primitives

While the ISdgMouseWidget and ISdgMouseAndKeyWidget interfaces help provide the internal mechanism underlying SDG widgets, they are still too low level to be convenient building blocks for an SDG widget developer. Instead, we give the SDG widget developer an inheritable object (also known as a widget primitive) that has all the expected behaviours of a conventional widget, and that also implements the basic SDG interface. The SDG Toolkit provides two types of widget objects. The SDG Control is the most basic: it is a widget primitive that represents a square region that the user can click on. In contrast the SDG User Control includes the functionality of the SDG Control, but it also allows programmers to use the interface builder and graphical

designer for developing widgets. Since the implementation of both these widget primitives is similar, I will only describe the development of the SDG user control class as it provides more functionality and convenience to the end programmer.

Microsoft .NET supplies a special object called a `UserControl` that is the building block for many conventional controls. To make this object SDG-aware, I created an `SdgUserControl` class as follows, with the complete code shown in Figure 4.3 (note that this is an internal implementation; the end programmer does not have to do this).

1. I defined a class that inherits from the standard `UserControl`, and declared that it implements the `ISdgMouseAndKeyWidget` (Figure 4.3, line 1). Inheriting the standard `UserControl` means it has all the methods, properties and event capabilities of a normal user control (e.g., properties that define its location, extents, background and foreground colors, and font). It also means that programmers can access this user control through the .NET interface builder in the same way they access non-SDG controls.
2. The `SdgUserControl` then implements the SDG interfaces (e.g., lines 3 and 16). If the SDGManger finds this user control under the current mouse coordinates, it invokes its SDG methods with the arguments filled in.
3. In turn, the `SdgUserControl` raises its own event corresponding to the received SDG event (e.g., lines 4 and 17). This new event is thus available to the end-programmer.

While this may sound complicated, this generic control was easy to create given our design logic. As seen in Figure 4.3, the complete class definition is handled in twenty one lines of programmer written code and handles both mouse and keyboard events. The SdgUserControl interface is a very basic implementation of the ISdgMouseAndKeyWidget Interface and a standard user control. The SDG User Control implements each of the required mouse and keyboard events (e.g., OnSdgMouseMove) and fires a corresponding event in the control. This means that each time a mouse movement event is obtained by the SDG Manager, the control itself will fire an event that can be used by both the widget designer and end programmers using the SDG control. Because the SDG User Control provides these as public SDG events, widget developers can now access SDG events in the interface builder.

```
1   public class SdgUserControl : UserControl, ISdgMouseAndKeyWidget{
2     // First add the SDG Mouse Events
3     public void OnSdgMouseMove(SdgMouseEventArgs e){
4       if (SdgMouseMove != null) SdgMouseMove(this, e);}
5
6     public void OnSdgMouseUp(SdgMouseEventArgs e){
7       if (SdgMouseUp != null) SdgMouseUp(this, e);}
8
9     public void OnSdgMouseDown(SdgMouseEventArgs e){
10      if (SdgMouseDown != null) SdgMouseDown(this, e);}
11
12    public void OnSdgMouseClick(SdgMouseEventArgs e){
13      if (SdgMouseClick != null) SdgMouseClick(this, e);}
14
15    //Now add the SDG Keyboard Events
16    public void OnSdgKeyDown(SdgKeyEventArgs e){
17      if (SdgKeyDown != null) SdgKeyDown(this, e);}
18
19    public void OnSdgKeyUp(SdgKeyEventArgs e){
20      if (SdgKeyUp != null) SdgKeyUp(this, e);}
21
22    public void OnSdgKeyPress(SdgKeyEventArgs e){
23      if (SdgKeyPress != null) SdgKeyPress(this, e);}
24
25    //Now add the corresponding events
26    public event SdgMouseEventHandler SdgMouseUp;
27    public event SdgMouseEventHandler SdgMouseDown;
28    public event SdgMouseEventHandler SdgMouseMove;
29    public event SdgMouseEventHandler SdgMouseClick;
30    public event SdgKeyEventHandler SdgKeyUp;
31    public event SdgKeyEventHandler SdgKeyDown;
32    public event SdgKeyEventHandler SdgKeyPress;
33  }
```

Figure 4.3. The SDG user control implementation

For the end programmer, creating SDG-aware control is now easy. Programmers can inherit and extend the SDG User Control class to create their own novel widgets, as this class provides all of the basic methods required to support input from multiple mice and keyboards. For example:

```
Public class MySdgClass : SdgUserControl
```

creates an empty SDG class with methods such as SdgMouseUp and SdgKeyUp that handles both mouse and keyboard events.

Microsoft .NET also supplies a generic object called a `Control` that is the most basic implementation of a widget. The implementation of the SDG Control is identical to the implementation of the SDG User Control seen in Figure 4.3, except that it inherits

from `Control` instead of `UserControl` in line 1. The `SdgControl` is a lightweight version of the SDG User Control that widget programmers can use when they need more control over the behaviour of an SDG widget. For example, programmers may find it easier to develop oddly shaped rotatable widgets using a SDG Control rather than trying to program against the assumptions made in an SDG User Control that all widgets need to be placed in the upright position.

In summary, the SDG Toolkit provides two inheritable widget objects for SDG widget programmers. Both the SDG Control and User Control implement the basic SDG mouse and keyboard widget interfaces so that programmers can immediately take advantage of the SDG functionality provided. The SDG Control provides basic functionality, while the SDG User Control allows programmers to develop widgets using the Microsoft Visual Studio interface builder.

4.3 Example: Creating an SDG Colour Mixer Control

Using the `SdgUserControl`, programmers can easily create their own SDG widgets using techniques familiar to them. To illustrate this, I will show how we can implement a trivial SDG colour-mixing widget that fully responds to two mice. This example could have been created for an arbitrary number of mice, but has been deliberately simplified. Figure 4.4 shows how two end users would see the square shaped control located in a window. The control is white if no one presses on it (left), blue if only the first person is pressing it (middle left), yellow if only the second person is pressing it (middle right), and green if both are pressing it at the same time (right). The title of the window also reflects the button down state of these mice. From a programmer's perspective, the colour mixer widget extends the SDG User Control by including a 'colour changed' event (triggered when the colour has changed) and two boolean properties: Mouse0Down and Mouse1Down, which are true if the respective mouse button is pressed on the widget. First, I describe how an end programmer would use the SDG colour mixer control in their own application. Second, I will show how an end programmer would actually develop this control.

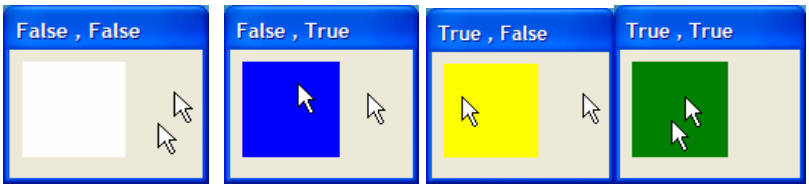


Figure 4.4. The SDG colour mixer application, the colour of the widget changes based on the number of people clicking on the widget at any given time.

4.3.1 The End Programmer's Use of the SDG Colour Mixer Control

The development of the SDG colour mixing application is trivial with the Microsoft Visual Studio designer. The SDG colour mixer widget is presented to the end programmer as an easily reusable SDG control with its own icon and identifying name. As with normal controls, it appears in the Visual Studio designer (Point 4, Figure 4.5). The entire application can be completed by writing only a single line of code. To create the SDG colour mixer application using the Mixer widget from the designer, the end programmer would need to do the following steps as illustrated in Figure 4.5:

1. Create a new windows form application
2. Add an SDG manager to the form using the graphical designer (Point 2, Figure 4.5)
3. Set the SDG manager's coordinates to be relative to Form1 (not shown)
4. Add the SDG Mixer control from the Toolbox (Point 4, Figure 4.5)
5. Add a callback for the ColourChanged event (Point 5, Figure 4.5)
6. In the ColourChanged callback add the following line of code to change the title of the top-level window

```
this.Text = sdgMixer1.Mouse0Down + "," + sdgMixer1.Mouse1Down;
```

All of these steps should be familiar to end programmers since SDG widgets can be added using the same toolbox that is used for single user widgets. Once an SDG widget has been designed, its use is as simple as adding a conventional widget to a form.

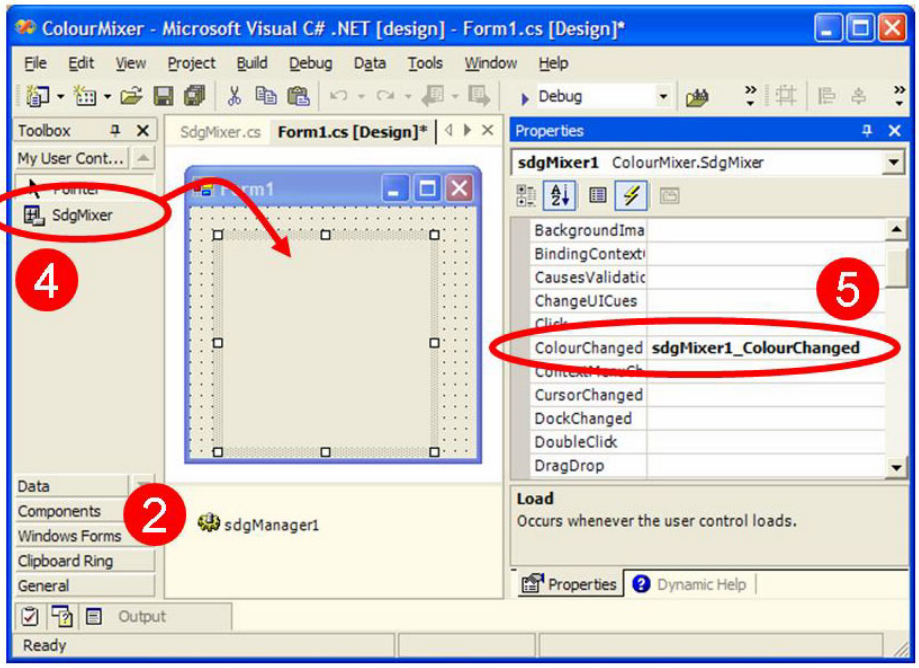


Figure 4.5. Using the designer to add your own SDG mixer widget

For completeness, Figure 4.6 shows the code that an end programmer would need to write in order to create the SdgMixer widget application without the designer. As previously mentioned, the control is implemented as a class with two Boolean properties Mouse0Down and Mouse1Down that indicate which mice are currently clicked on it. A 'colour changed' event is fired each time the colour of the widget is modified. As previously shown, if the interface designer were used, then the programmer's job is relegated only to adding a single line of code. By adding the SDG Manager and an SDG Mixer widget to the form constructor (`public Form1()`) we would have a button that changed colour based on how many mice were clicking on the widget like in Figure 4.4, without the changing title. By using the colour changed event and the Mouse0Down and Mouse1Down properties we can change the application title so that it represents the current state of the mixer widget. The end result of this application is seen in Figure 4.4.

```
public class Form1 : System.Windows.Forms.Form
{
   //All of this code except for one line automatically generated
   // by the Interface Builder

   private SDG Manager SDG Manager1;
   private SdgMixer sdgMixer1; //Our new sdg aware widget

   public Form1() {
       SDG Manager1 = new SDG Manager();
       sdgMixer1 = new SdgMixer();
       SDG Manager1.RelativeTo = this;
       sdgMixer1.Location = new Point(10,10);
       sdgMixer1.ColourChanged += new EventHandler(MixerColourChanged);
       this.Controls.Add(sdgMixer1);
       this.ClientSize = new Size(200,180);
   }

   private void MixerColourChanged(object sender, EventArgs e) {
      //This line of code is written by the end programmer
       this.Text = sdgMixer1.Mouse0Down + "," + sdgMixer1.Mouse1Down;
   }
}
```

Figure 4.6. The SDG colour mixer application source code

4.3.2 Developing the SDG Colour Mixer Widget

Developing this widget using the SDG Toolkit is quite simple. Figure 4.7 provides the complete code of our SDG widget example. To explain its logic, the array named "press" contains two elements, each holding the 'button press' state of the first and second mouse. The `SdgMouseDown` event handler sets the appropriate `press` element to `true`, while the `SdgMouseUp` handler sets it to `false`. Both call the `Draw` method, which is a simple state machine that calculates which mouse or combination of mice are currently pressing the widget, and sets the background color accordingly.

While simple, this example illustrates that the SDG Toolkit makes SDG widget development and reuse straight-forward. First, it allows programmers to develop reusable widgets that can be added to an application in the same way that single user widgets such as buttons are added to conventional windowing applications. Second, using SDG widget primitives, widget programmers can use SDG events to rapidly develop collaboration aware widgets.

```
public class SdgMixer : SdgUserControl {
  private  Boolean [] press = new Boolean [2];
  private SdgMixer() {
    this.SdgMouseDown += new SdgMouseEventHandler(MixerMouseDown);
    this.SdgMouseUp += new SdgMouseEventHandler(MixerMouseUp);
  }
  private void MixerMouseDown(object s, SdgMouseEventArgs e){
    press [e.ID] = true;
    Draw ();
  }
  private void MixerMouseUp(object s, SdgMouseEventArgs e){
    press [e.ID] = false;
    Draw ();
  }
  private void Draw () {
    //Fire the ColourChanged event
    if (ColourChanged != null) ColourChanged(this, new EventArgs());
    //Change the widget Colour
    if ((press[0] || press [1]) == false)
      this.BackColor = Color.White;
    else if (press[0] && press [1])
      this.BackColor = Color.Green;
    else if (press[0]) this.BackColor=Color.Yellow;
    else this.BackColor = Color.Blue;
  }
  //public properties available to end programmers
  public Boolean Mouse0Down {
    get { return press[0]; }
  }
  public Boolean Mouse1Down {
     get { return press[1]; }
  }
  public event EventHandler ColourChanged;
}
```

Figure 4.7. The SDG colour mixer widget source code

4.4 Example: Creating an SDG Text Placement Widget

Using the `SdgUserControl`, programmers can easily create SDG widgets that are aware of multiple keyboards and mice. To illustrate this, I will show how we can implement a trivial SDG text placement widget that fully responds to two mice and two keyboards. Figure 4.8 shows the full application. The SDG Text Placement widget is a white rectangular shaped control (Figure 4.8) that allows people to click on a region of the screen to specify where they would like to place their text. The location where text is to appear is visualized by a small coloured circle seen on the top left region of each mouse called the text placeholder (Figure 4.8, middle). The text drawn is of a different colour,

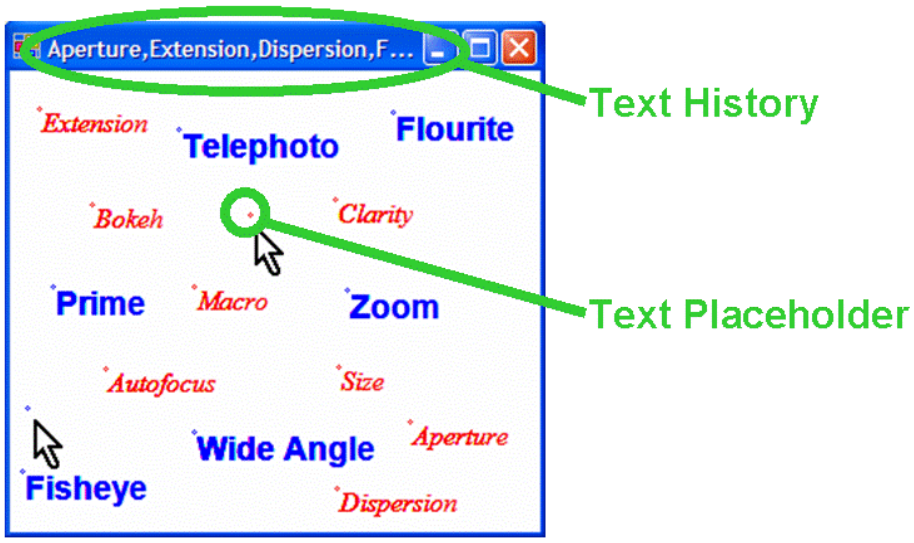


Figure 4.8. The SDG text place application, users can place bits of text in various locations within the Text Place widget

size and font depending on which keyboard was typing. Text entered is stored in a list and a history is shown in the application title (top). From a programmer's perspective, this widget extends the SDG User Control and includes a 'Text Entered' event (trigged when either user presses the return key) and a 'Last Text Entered' string property, which obtains the most recently entered phrase by either keyboard. While it is possible to support an arbitrary number of mice and keyboards, this application is deliberately simplified to handle two mice and two keyboards.

4.4.1 The End Programmer's Use of the SDG Text Placement Widget

The development of this SDG Text place application is trivial with the Microsoft Visual Studio designer. As with the SDG Colour Mixer, the SDG text place widget is presented to the end programmer as a reusable SDG control with its own icon and identifying name. To create the SDG Text Place application using the Text Place widget from the designer, the end programmer would need to perform the following steps:

Steps 1 to 3 are the same as in the SDG Colour Mixer example.

4. Add the SDG Text Place control from the Toolbox
5. Add a callback for the TextEntered event
6. In the TextEntered event callback add the following line of code to provide text feedback to the end user while they are typing in the buffer.

```
this.Text = sdgText1.LastTextEntry + “,” + this.Text;
```

As mentioned earlier, the SDG Text Place widget contains one string property (LastTextEntry) that indicates the most recent text entered by either keyboard, and a ‘Text Entered’ event that is fired each time the string buffer is changed in the text place widget. For completeness, Figure 4.9 shows the code that an end programmer would need to write in order to create this Text Placement application without the designer.

```
public class Form1 : System.Windows.Forms.Form
{
   //All of this code except for one line automatically generated
   // by the Interface Builder
   private SDG Manager SDG Manager1;
   private TextPlaceWidget sdgText1; //Our new sdg aware widget

   public Form1() {
       SDG Manager1 = new SDG Manager();
       sdgText1 = new SdgMixer();

       SDG Manager1.RelativeTo = this;
       sdgText1.Location = new Point(10,10);
       sdgText1.TextEntered += new EventHandler(TextEntered);

       this.Controls.Add(sdgText1);
       this.ClientSize = new Size(300,250);
   }
   private void TextEntered(object sender, EventArgs e) {
      //This line of code is written by the end programmer
      this.Text = sdgText1.LastTextEntry + “,” + this.Text;
   }
}
```

Figure 4.9. The SDG text place application source code

```
public class TextPlaceWidget : SdgUserControl {
  private PointF[] TextEntryPoints = new PointF[2];
  private string[] TextBuffer = new string[2];
  private string LastText;
  Graphics g;

  private TextPlaceWidget() {
     this.SdgMouseDown += new SdgMouseEventHandler(TextPlaceMouseDown);
     this.SdgKeyDown += new SdgKeyEventHandler(TextPlaceKeyDown);
     g = this.CreateGraphics();
     this.BackColor = Color.White;
  }
  private void TextPlaceMouseDown(object s, SdgMouseEventArgs e){
     TextEntryPoints[e.ID] = new PointF((float) e.X, (float) e.Y);
     //Draw the Text Placeholder
     if (e.ID > 0)  g.DrawEllipse(Pens.Blue, e.X, e.Y, 2, 2);
     else           g.DrawEllipse(Pens.Red, e.X, e.Y, 2, 2);
  }
  private void TextPlaceKeyDown(object s, SdgKeyEventArgs e){
     if (Keys.Enter == e.KeyCode) {
         LastText = TextBuffer[e.ID];
         TextBuffer[e.ID] = string.Empty;
         //Fire the Text Entered Event
         if (null != TextEntered) {
             TextEntered(this, new EventArgs());
         }
     }
     else if (Keys.ShiftKey != e.KeyCode) {
         TextBuffer[e.ID] = TextBuffer[e.ID] + (char) e.KeyValue;
         //Write the text to the screen
         Font f = new Font("Times", 12, FontStyle.Italic);
         Color c = Color.Red;
         if (e.ID > 0) {
              f = new Font("Arial", 14, FontStyle.Bold);
              c = Color.Blue;
         }
         g.DrawString(TextBuffer[e.ID], f, c, TextEntryPoints[e.ID]);
     }
  }

  //public property and event available to end programmers
  public string LastTextEntry {
    get { return LastText; }
  }
  public event EventHandler TextEntered;
}
```

Figure 4.10. The SDG text place widget source code

4.4.2 Developing the Text Placement Widget

Developing this widget using the SDG Toolkit is quite simple. Figure 4.10 provides the complete code for the text placement widget. To explain its logic, an array named `TextEntryPoints` holds the location where text is to be entered into the widget (note that all coordinates in this example are relative to the top left corner of the widget). Each time a mouse down event is received on the widget, the text entry point is changed and a small coloured circle is drawn at the text entry point. Another array called `TextBuffer` holds the text that has been entered in by the user's keyboard and writes this buffer out to the screen each time a character is typed. If the enter key is pressed then the last text entered is stored in a variable called `LastText` and the `TextBuffer` is emptied. Both the `TextPlaceMouseDown` and `TextPlaceKeyDown` contain code to change the colour of the drawings (and fonts in the Key event) based on the originating ID of the event. This application uses the assumption that Mouse 0 is mapped to Keyboard 0 and Mouse 1 is mapped to Keyboard 1.

This example illustrates how the SDG Toolkit simplifies the development of more complex widgets that are aware of multiple keyboards and multiple mice. It showed the use of coordinates relative to the widget and it made use of the automatic keyboard to mouse mapping provided by the SDG Toolkit.

4.5 Towards an SDG Widget Library

Although the SDG Toolkit does not yet include a library of SDG widgets, many widget examples have been developed using the toolkit. For example, student Rob Diaz-Marino created several novel SDG versions of conventional widgets such as radio buttons and sliders, and I have developed SDG versions of innovative menu systems such as flow menus and ToolGlasses [Tse, et al., 2004]. These examples are available to the programmer on the SDG Toolkit web site, but more work needs to be done in order to make a cohesive library of widgets available to end programmers. This is beyond the scope of this thesis.

Still, in Chapter 5, I will describe the development of several SDG widgets to illustrate the power of the SDG widget layer by example.

4.6 Summary

In this chapter I described the development of an SDG widget layer. First, I showed how the SDG Toolkit identified and sent input events to SDG widgets. Second, I described the implementation of two inheritable widget objects, the SDG Control and the SDG User Control. Finally, I provided two illustrative examples of how a programmer would use the widget layer of SDG Toolkit to develop their own customized widgets and widget applications. I have summarized the issues and solutions presented in this Chapter in Table 4.1.

Issue	Solution
SDG Widget Infrastructure	The SDG Toolkit provides an infrastructure for rapidly prototyping SDG widgets in three steps. First, it provides a standard way of communicating input events to widgets through the ISdgMouseWidget and IsdgMouseAndKeyWidget interfaces. Second, it obtains a list of widgets that implement the SDG widget interfaces by recursively searching through all controls on a form. Finally, the SDG Toolkit takes input from multiple mice and keyboards and directs it to the appropriate SDG widget.
SDG Widget Primitives	The SDG Toolkit provides two inheritable widget objects for SDG widget programmers. Both the SDG Control and User Control implement the basic SDG Mouse and Keyboard widget interfaces so that programmers can immediately take advantage of the SDG functionality provided. The SDG Control provides basic functionality, while the SDG User Control allows programmers to develop widgets using the Microsoft Visual Studio interface builder

Table 4.1. Overview of SDG development issues and solutions implemented in the SDG widget layer

The goal of Chapters 3 and 4 was to detail the development of all the components of the SDG Toolkit in such a way that another researcher could easily replicate this work if they needed to. I accomplished this goal by presenting a set of problems with SDG development (both at the input level and the widget level) and described my solution to

these problems through the implementation of the SDG Toolkit. While all these solutions are written in Microsoft Visual C#, they should be general enough to be reimplemented on other modern graphical user interface systems and programming environments.

Chapter 5. Validating the SDG Toolkit

In Chapter 1, I described two thesis goals. The first was to allow average programmers (with no system level programming experience) to easily develop applications using the SDG Toolkit. The second was to enable average programmers to create novel SDG widgets using the widget layer of the SDG Toolkit. This chapter provides validation that SDG applications and SDG widgets can be easily designed using the SDG Toolkit. I do this by detailing several case studies of how the SDG Toolkit has been used in practice. I first describe case studies of my own development experiences using the SDG Toolkit, and use these as a self evaluation. I then describe the experiences of other SDG researchers using the SDG Toolkit and its widget layer.

In Chapter 3, I explained that even though the principles of the SDG Toolkit deal specifically with multiple mice and keyboards, they would generalize to other novel input devices. In the second part of this chapter, I further validate the design of the SDG Toolkit through case studies where I show that the principles of the SDG Toolkit do generalize to other novel input devices.

5.1 Personal Case Studies using the SDG Toolkit

In this section I examine how the SDG Toolkit has been used for my own research in SDG. This self-evaluation is valuable, for if problems and issues arose during my own use of it, it is almost certain that others would have similar problems. Successes also reveal that certain SDG design goals are reachable, at least in principle. I begin by discussing a novel SDG version of a single user strategy game that was designed as an example to test the SDG Toolkit in its early stages of development. Next, I discuss a menu system called flow menus that I implemented to test the development of SDG widgets. Finally, I describe how I used the SDG Toolkit to rapidly prototype SDG software needed for a study on spatial partitioning behaviour.

5.1.1 SDG Rush Hour

SDG Rush Hour is a collaborative puzzle solving game that I developed as a program to test the rapid prototyping capability of the SDG Toolkit during its early design stages (i.e., before the availability of SDG User Controls). I originally played an online web game called Rush Hour and decided to reimplement this game so that multiple people could solve these puzzles simultaneously. The principle of the game is that the players need to move a special red card to an exit marker (top left, Figure 5.1). The problem is that the red car is stuck in traffic and other cars only have limited space to travel forwards or backwards. The challenge of this game is to rearrange these cars so that the red car can get through to the exit.

The entire SDG Rush Hour application was built from scratch in only two days. To do this, I first created a single user application that constrained moveable picture boxes of cars to simulate the physics of the game. I then made it possible to load and save different game levels, and to detect when the game was won. Finally, I transformed the single user version of my application into a multiple user version in less than one hour of straight-forward programming (although some extra programming was required to add collision detection for cars moving at the same time in the same position).

The SDG Toolkit allowed a simple SDG game to be rapidly prototyped in a weekend. Most of the effort spent in the development of this application was on logic specific to the game rather than the details of making the game handle multiple inputs.

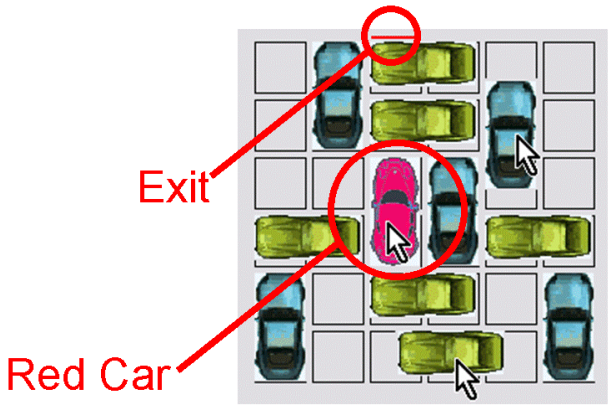


Figure 5.1. The SDG rush hour application

Since this application was developed without the SDG Widget layer, extra code was needed to identify widgets and forward input events to them. Instead of identifying the widgets in the application, I hard coded them into an array structure, I then wrote code to explicitly send messages to widgets when an event had occurred over their space. This meant that each time I added a widget I would need to modify my code so that the new widget would be recognized in the event handler. Since SDG Rush Hour uses many different car widgets I spent about two hours during the weekend implementing functionality that simulated SDG widgets.

5.1.2 Pie Menus and Flow Menus

Seeing that much of my development time in the SDG Rush Hour application was spent identifying and communicating with SDG widgets, I decided to add the widget layer to the SDG Toolkit, as described in Chapter 4. I tested this layer by creating SDG versions of novel menu systems. The first was a crude variant of the Pie Menu. As seen in Figure 5.2, multiple people can raise their own pie menu, and select an item or cycle through items using the mouse scroll wheel to change the colour and thickness of their drawings. Scrolling the mouse wheel selects the next or previous option. Pressing the scroll wheel or the left mouse button commits the selection made.

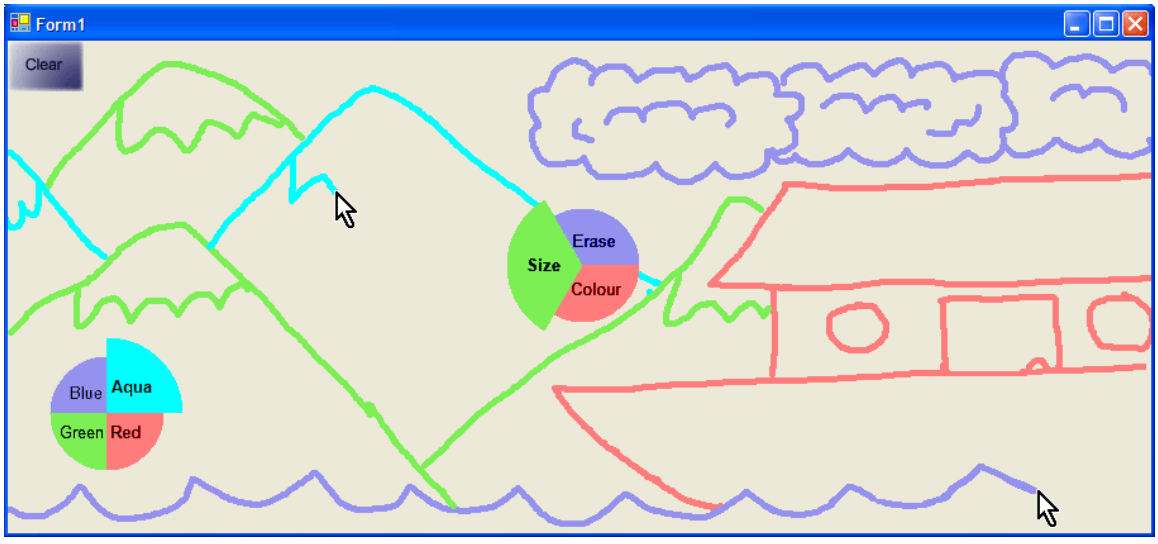


Figure 5.2. SDG Pie menu: Items are selected with the mouse wheel, four mice are shown in this example

This particular interaction technique was first prototyped as an SDG User Control widget that would visualize a menu in a circular form. I then added code to indicate the currently selected menu item by making the pie slice larger. Next, I added interaction behaviours to make the pie menu properly respond to movements of the mouse wheel and mouse clicks. Finally, I created a simple SDG drawing application using this SDG Pie Menu, where multiple people could draw concurrently and use their own pie menu for colour selection as seen in Figure 5.2.

The original version of the Pie Menu was developed using the SDG User Control. This proved problematic for pop-up menus since one unintended property of a User Control is that they erase any line drawings made underneath a window when moved (they are normally fixed to a permanent location). Since the Pie Menu was implemented as an SDG User Control it would act like a large eraser when moved around the screen. Consequently, I created a new widget control using a transparent window that ensured the drawing would not be erased when the menu was moved; it also allowed semitransparent menus to be developed. The addition of the mouse widget interface to the windows form class required only four simple lines of code.

I emphasize that even though the SDG Toolkit did not provide the widget primitive needed for these menus, it provided the tools needed to rapidly build a custom widget primitive.

It took only two hours to convert the single user version of the Pie Menu into a multi-user version. This included the time required to create my own SDG widget primitives using a semi transparent windows. Since the Pie Menu took 16 hours to develop, 12.5% of the total development time was spent working on the multiple user aspects of the pie menu. Of course, since it is now a widget the pie menu could now be trivially reused in other applications.

The Pie Menu was just an experiment, as I did not expect people to use the mouse wheel instead of the pointer to make selections from a menu. For a more realistic widget, I developed Guimbretière and Winograd's Flow Menu (2000) as an SDG interaction technique. Figure 5.3 shows Flow Menus with an SDG drawing application that allows multiple users to draw different coloured lines of different widths. The specialized flow

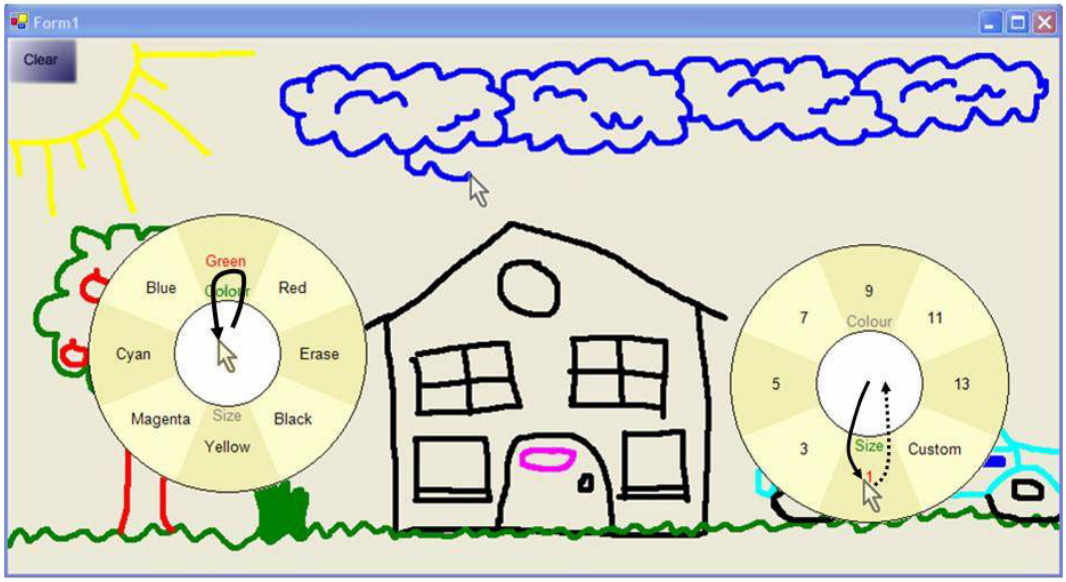


Figure 5.3. An SDG drawing application. Two users are selecting a drawing colour and size from their individual flow menu, as the third is drawing. Gestures are shown with black arrows.

menus give users colour and line options while a clear button is provided on the top left hand side to clear the entire screen. Users can also erase by selecting the erase colour of the flow menu.

Flow menus use gestures as the primary interaction method [Guimbretière and Winograd, 2000]. The idea is that a user raises the menu as a popup window (with the click of a button or a specialized gesture) and moves outside of the centre white circle to make a first level menu selection and back in to make a second level menu selection. For example, Figure 5.3 shows a user on the right side moving outside the centre white circle to select the size menu; the dotted line represents the path that would be taken if the user was to select a pen size of 1. The idea is that experienced users would learn specific gestures over time and be able to perform them so rapidly that the menu may not even appear on the screen.

Using the source code from the pie menu application I was able to create the flow menu in a single day. Since the SDG Toolkit code could be reused from the Pie Menu example, I only had to develop new code to handle the interaction and drawing of Flow Menus. Of course, the flow menu can now be easily added to any application.

In summary, the SDG Toolkit allowed for several novel widgets to be developed in a very short period of time. The Pie menu example showed how most of the development time was spent implementing a visualization technique for menu items and creating the menu interaction. The Flow Menu example showed how existing code could be easily adapted to a new visualization and interaction technique with very little extra programming effort. While neither used the SDG User Control, they illustrate how the underlying infrastructure supports the development of new widget primitives through the SDG widget interfaces. The SDG Window widget has now been added in the current version of the SDG Toolkit.

5.1.3 Study on Spatial Partitioning

As a side project, Stacey Scott, Jonathan Histon, Saul Greenberg and I conducted a study to observe the use of spatial partitioning in various tracing and drawing tasks [Tse et al., 2004]. This study required me to build a simple drawing application that loaded bitmapped images and let two people trace preloaded images simultaneously (e.g., Figure 5.4). The application also recorded all multiple mouse movements and click information for analysis. Using this data we discovered that people naturally partition their workspaces into separate areas often influenced by their own seating positions.

As usual with study designs, we went through several radically different systems and much fine tuning before we were convinced that we were studying the right thing.

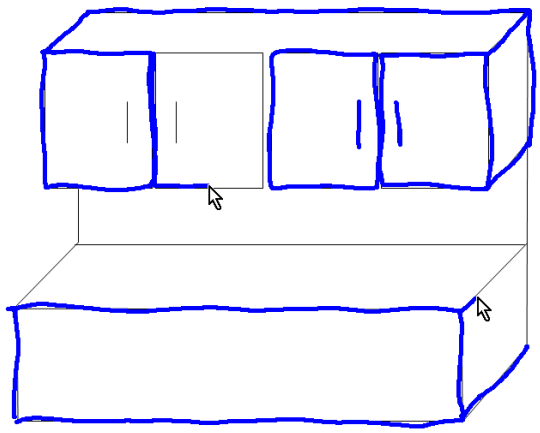


Figure 5.4. The spatial partitioning study application

The SDG Toolkit was used for prototyping the various iterations of this study. In an early iteration of our study prototype we wanted to see how using buttons with different visual behaviour affected its simultaneous use. To do this, we first designed buttons with specialized visual behaviours. Figure 5.5b shows a button that is raised when no mouse is clicking on it (left), half depressed when only one mouse is clicked on it (middle) and fully depressed when both mice are clicking on it (right). Using the SDG User Control it was trivial to implement the different visual behaviours of the buttons. To increase the likelihood of two participants using a button at the same time in our tracing program, we created paintbrushes that needed to be refilled after a certain amount of drawing was done on the tracing canvas. As seen in Figure 5.5a, using the multiple customizable cursors provided by the SDG Toolkit I rapidly prototyped a refill meter for each user that indicated the total amount of ink remaining.

Figure 5.5. Features seen in an early iteration of the SDG Study a) A meter indicating the total amount of ink remaining b) Different visual behaviours for a refill button

The SDG Toolkit allowed many iterations of our study prototype to be developed with minimal turnaround (usually around a day, sometimes hours, sometimes minutes). In particular, the provision of customizable cursors simplified the design of our refill meter prototype, the SDG widget layer trivialized the development of our novel widget visualizations, and the SDG event model mouse tracking easy. Perhaps even more importantly, the SDG Toolkit was reliable. We ran our study on twenty-four pairs of participants without ever running into technical problems.

The parked mouse strategy was problematic for our SDG study application. We wanted to have a configuration screen that would allow us to specify the participant's ID number and the corresponding study scenario. To use this screen we had to start the application without SDG support so that single user widgets would work, after which we invoked SDG support when the study was to begin. This problem emphasizes that our strategies for dealing with the system mouse comes with tradeoffs.

In summary, through the development of our SDG study application we saw how the SDG Toolkit provided a rapid turnaround of our prototype ideas. The application developed with the toolkit was reliable enough to be run on a large number of participants without technical problems. Finally, although the parked mouse strategy made it difficult to use conventional widgets, the SDG Toolkit could be easily stopped so that single user widgets would function properly.

5.2 Case Studies of Others Using the SDG Toolkit

The following case studies describe applications developed using the SDG Toolkit by researchers other than myself. I begin by describing a project that I did in collaboration with a student at the University of Saskatchewan. Then, I detail an example SDG widget library and sample application developed by a summer intern in our lab. I then describe a research application that combines both distributed groupware and single display groupware. Finally, I discuss a graph navigation tool that was made to support multiple users in a very short amount of time.

5.2.1 SDG ToolGlass

After reading about Bier et. al.'s notion of a ToolGlass (1993), Nicole Stavness (a student at the University of Saskatchewan) and I decided to implement ToolGlasses as an SDG interaction technique. While I was her partner on this project, she did most of the end-programming and design details. ToolGlasses were originally designed to exploit two handed input by a single user: one hand moved the ToolGlass over a surface (perhaps transforming how the underlying objects are displayed), while the other hand would 'click through' the ToolGlass to assign a property to the underlying object. For example, the ToolGlass could contain a palette of colors, and each user could position a particular color over an object and assign that color to it by clicking through it. The SDG re-creation provides all users with their own ToolGlass (Figure 5.6). What is especially interesting about this is that each user has two pointing devices: one to move the lens (the cursor is the small hand in the bottom left corner of each Magic Lens) and one to click through (the arrow cursor). To our knowledge, this is the first ToolGlass example to have

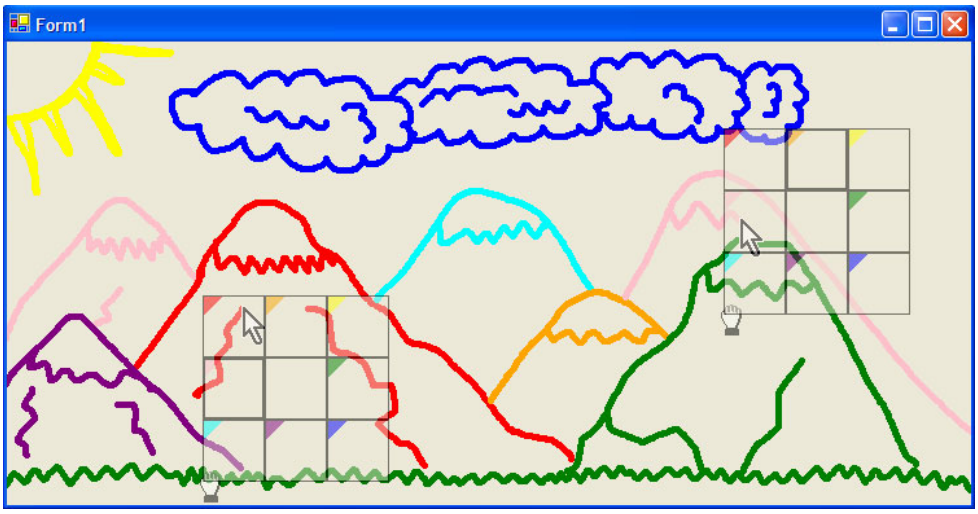


Figure 5.6. SDG ToolGlass, showing two users and their moveable click through tool palettes

been equipped with SDG semantics. This could easily be extended into a collaborative tool [Druin et al., 1997], e.g., where people can mix and select new colors by placing their lens atop of one another.

To create the ToolGlass, Nicole created a number of transparent buttons inside an SDG User Control (as described in Chapter 4). These buttons fire an event when they are clicked so that the corresponding colour can be changed in the drawing application. Finally, code was then added to make each ToolGlass follow the movements of a corresponding SDG mouse. Since a user control was originally used for the implementation of the SDG ToolGlass it would erase all of the drawings made underneath it when moved; thus making the tool glass similar to a giant eraser. This was solved by placing each tool glass into a transparent window above the actual application as in the flow menu example (Section 5.1.2).

Nicole encountered two problems when developing the SDG ToolGlass example as seen in the email correspondence. The first was implementing the `ISdgMouseWidget` interface (three lines of code described in Chapter 4) and learning how to create a transparent window using Visual Studio .NET. The second problem was that she did not know how to communicate with the buttons she placed inside the SDG user control. Specifically, she did not understand how the `sender` argument of the widget mouse event indicated the currently clicked button. This is an issue that may not be clear to other

SDG Widget programmers. After I provided some of the answers to her questions in email, Nicole was able to complete the SDG ToolGlass example. This experience suggests that better tutorial documentation for advanced SDG designs would be helpful. This has not yet been added to the SDG Toolkit but it is planned for a future release.

Since Nicole was located in Sasketchewan during the development of the SDG ToolGlass, I provided minimal support during her development time, mostly in the form of troubleshooting and interface design discussion. In addition to learning how to use the SDG Toolkit, Nicole had to familiarize herself with the Microsoft Visual Studio C# programming environment. Nicole started by learning the basic SDG drawing application (available on the SDG Toolkit web page) and then she learned the basic SDG widget example. To exacerbate Nicole's situation I was away at a conference during the time when she was developing the SDG ToolGlass example! Nicole spent about 1 week learning how to use the Microsoft Visual Studio programming environment and learning the SDG Toolkit. It took her about 3 weeks to develop the ToolGlass example. In total, Nicole spent about 20% of her total time learning how to use Visual Studio and the SDG Toolkit (assuming a total of 5 weeks development time). Of course, since this project was done as a class exercise the development time was mixed with many other responsibilities of a full course load – this was not 5 weeks of dedicated effort.

In summary, the SDG Toolkit allowed Nicole Stavness to quickly redevelop and modify an existing interaction technique, even though she had no prior experience using the SDG Toolkit nor the Microsoft Visual Studio programming environment.

5.2.2 SDG Widget Library

Rob Diaz-Marino, a summer intern at the University of Calgary, developed a library of three multi-user analogues of single user widgets: a radio button, a check box and a slider. From a programmer's perspective, the SDG widget library appeared in the programmer's toolbox and was used in the same way that single user widgets are used (like the widget examples in Chapter 4) except that some initialization is required (discussed later). From an end-user's perspective, the widgets in the SDG widget library appeared and behaved like single user widgets except that each user's selection was

associated with a colour and extra information was provided to the end user when their cursor was below it. I will describe the three SDG widgets below. For simplicity each example will use the assumption that there are only two users.

Figure 5.7 shows the different states of a multi-user checkbox widget. When it is not selected the checkbox is white (Figure 5.7a). If one cursor clicks on the checkbox a light grey check appears in the middle of the box and the checkbox is filled with a faded colour corresponding to the colour of the cursor (b). If two cursors select the checkbox the check turns black and each half of the widget is filled with the faded colour of the two cursors (c). More information is revealed if the user places their cursor over the multi-user widget, where the background colour turns a solid colour and a list of users who have selected this widget appears at the bottom (d). If multiple mice hover over the checkbox a list of all hovering cursors appears below the checkbox so that each participant knows whether or not they have selected the multi-user widget (e).

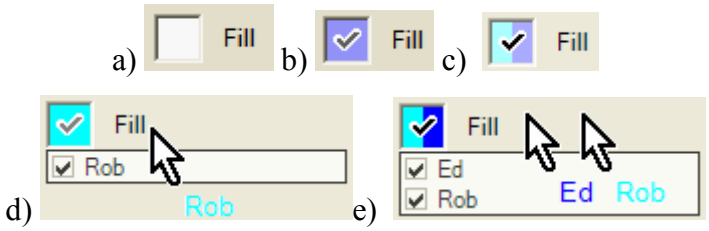
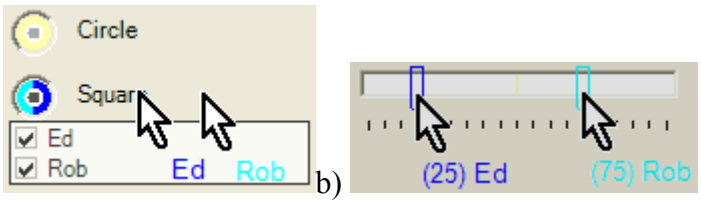


Figure 5.7. Different states of a novel SDG checkbox a) none selected b) one user selected c) two users selected d) one user selected their cursor is over the fill widget e) one user selected and two cursors are over the fill widget

The SDG radio button behaves in a similar manner, except that in any given group of radio buttons only one radio button can be selected by each user. The visual behaviour of the radio button is identical to the checkbox except that a small black dot is used instead of a check. Figure 5.8a shows a group of two radio buttons where the top radio button has been selected by one cursor and the bottom radio button has been selected by two cursors. Like the checkbox example, if a user moves their cursor over the radio button widget their name appears in a list of users who have selected the widget.

The multi-user slider in Figure 5.8b behaves like a single user slider except that each cursor's slider position is marked by a different colour. Figure 5.8b shows two slider handles that are outlined in the colour of their respective cursor with the numeric



a) b)

Figure 5.8. Other multi-user widgets a) SDG radio button b) SDG slider.

position of their cursor written in the cursor's text caption (bottom). To minimize clutter, the slider handle is replaced with a thin coloured line when the cursor is not over the slider widget.

Using these widgets, Rob Diaz-Marino developed a simple shape manipulating application that allowed up to four participants to manipulate the shape and sizes of different coloured objects on the screen. By selecting the fill checkbox, the object would be filled with the user's cursor colour.

While the development of this application was fairly straightforward, the design of the SDG Toolkit added some complexity to the development of the SDG widget examples. Each time the SDG widgets were used, extra code had to be written in order to modify the colour and text caption of each cursor so that a colour could be associated with each cursor. Also, a reference to the SDG Manager had to be passed in to each SDG slider so that it could modify the cursor text caption to show the actual value of the slider when a mouse hovered over the widget. This complexity is caused by the widget-like implementation of the SDG Manager. That is, just as two widgets cannot communicate with each other, SDG widgets cannot (easily) communicate with the SDG Manager. Rob Diaz-Marino's experience with developing widgets has led me to modify the SDG Toolkit so that widgets can access the SDG Manager through the sender argument of any SDG widget event.

Rob Diaz-Marino had no prior experience with the SDG Toolkit, and only a few months of experience using the Microsoft Visual Studio development environment. I asked Diaz-Marino about his experience learning to use the SDG toolkit and he told me in email correspondence that "the basic functionality of the SDG Toolkit was very easy to use, and was almost trivial to understand." While this speaks about the learnability of the SDG Toolkit I also wanted to see how long it took to develop these fairly sophisticated

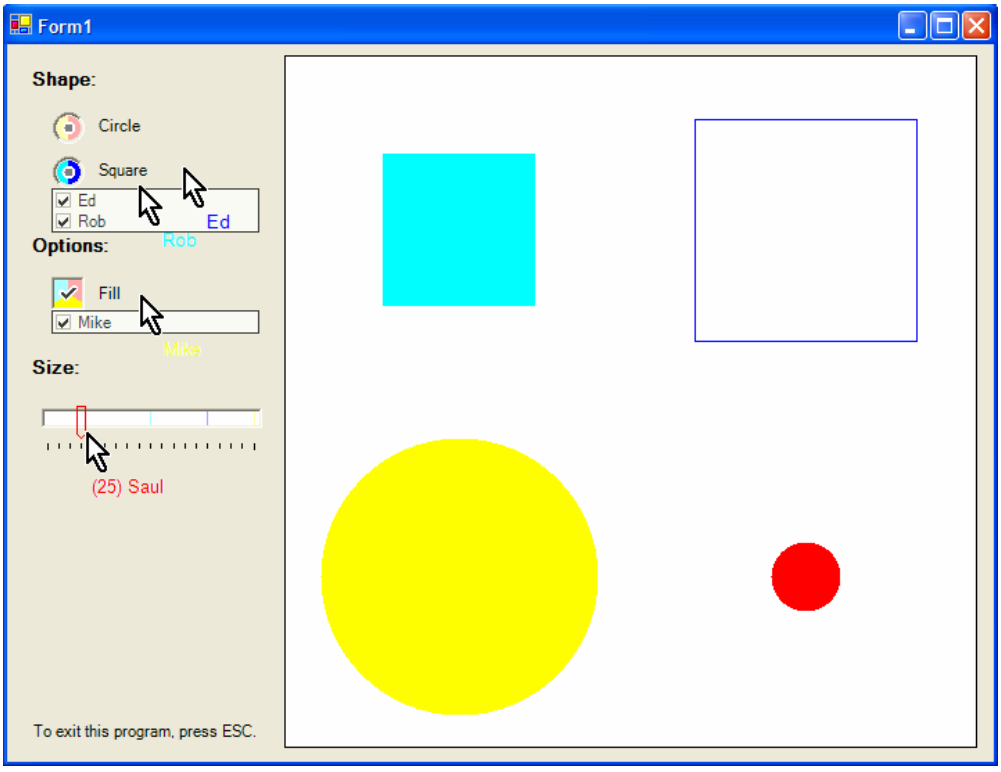


Figure 5.9. A shape manipulating application developed using the SDG widget library

widgets. The SDG widget library and the sample application took only three days to develop and "only 10% of (his) time was spent figuring out the SDGToolkit, the rest was spent on the graphics and logic for the controls". Even despite the issues Rob Diaz-Marino experienced getting his widgets to communicate with the SDG Toolkit, the majority of his development time was still focused on the implementation of his ideas for multi-user widgets.

In summary, the SDG widget library showed how a summer undergraduate student with no prior experience with the SDG Toolkit or SDG in general could develop several novel analogues of single user widgets in a short amount of time. His development experience also revealed a potential weakness of the SDG Toolkit that has been resolved in a recent version.

5.2.3 MPG and Digital Arm Shadows

Tang et al. (2004) developed several applications in a new area of research known as Mixed Presence Groupware (MPG), the combination of single display groupware and

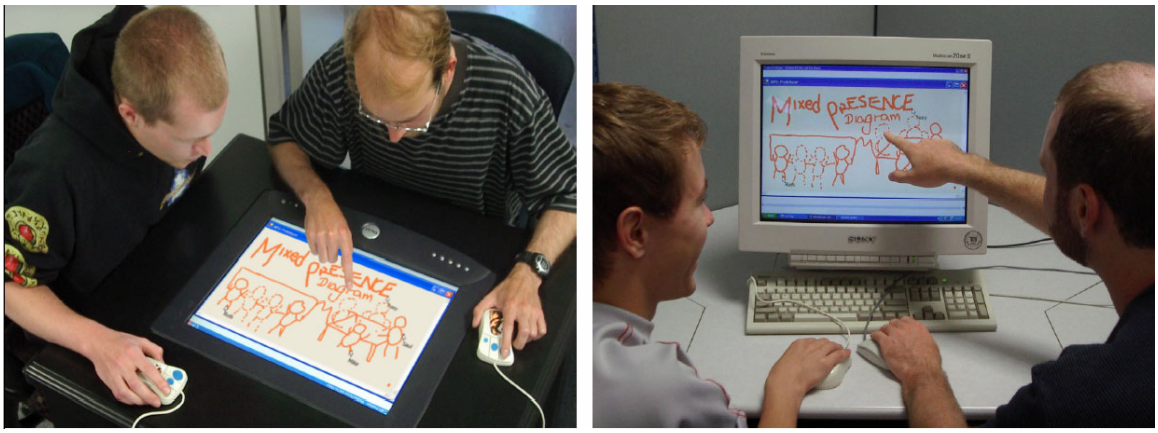


Figure 5.10. The initial MPG application (photo from Tang et al., 2004)

distributed groupware. The first example MPG application allowed a drawing canvas to be shared over a network by groups of SDG participants. This example application provided correctly rotated cursors and text captions for each participant working over a table or upright display and allowed a group of people working on a table top display to collaborate with those working on an upright display (Figure 5.10).

To develop this application Tony Tang used the SDG Toolkit to rapidly prototype a simple drawing application that worked over upright and table top displays and included cursors with different text captions. Tang used a distributed toolkit called the Collabrary (Boyle, 2002) to allow multiple people to share their drawings over a network, and used the SDG mouse class to provide a separate cursor for each remote participant.

After the development of this sample application Tony Tang found that participants that were not sharing the same display had very little interaction with each other. For example, the participants located around a table would rarely communicate with the participants located around the upright display, even if they were in the same room. Tang felt that this difference may have been caused by a decreased perceived presence of the remote participants. While local participants could use their arms and hands to gesture on the screen, remote participants had only a small white cursor to gesture with. To resolve this problem Tony Tang implemented another shared drawing application where the small SDG cursors were replaced with large semitransparent arm shadows with videos of each participant placed on a respective side of the drawing space (Figure 5.11). This application also included a feature that would make the semi-transparent arm shadows

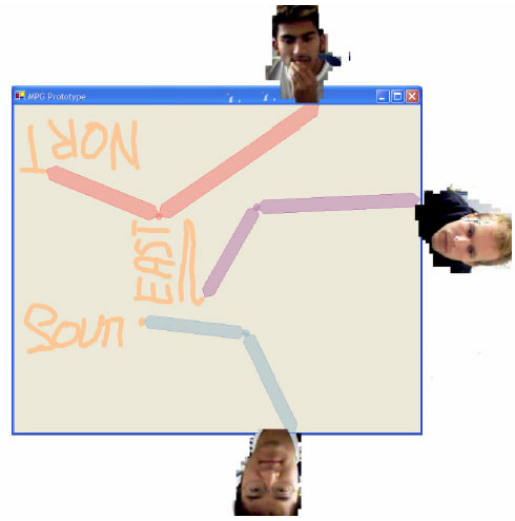


Figure 5.11. The arm shadows application (photo from Tang et al., 2004)

slowly fade away if they were unused so that they would not clutter up the drawing space. To provide a video of each participant, tools provided by the Collabrary were used. The arm shadows application detected when a participant had left their seat through an embedded light sensor phidget (or physical widget) provided by the Phidgets Toolkit [Greenberg and Fitchett, 2001].

The arm shadows application extended the initial MPG example and replaced SDG cursors with coloured arms drawn on semi-transparent windows. Since Tang developed this application while the widget layer was being developed, he had to manually communicate with each coloured arm by redirecting input from the SDG event stream.

The complexity of this application is quite impressive considering the fact that Tang had no prior experience with Microsoft Visual Studio, the SDG Toolkit, the Collabrary Toolkit or the Phidgets Toolkit. When asked about the difficulty of learning to use the SDG Toolkit, Tang responded "the design decisions made it fairly easy to pick up." Since Tang was learning so many toolkits at the same time he was not able to give an estimate of the total time spent working with the SDG Toolkit. Instead Tang commented on how the SDG Toolkit was an enabling factor in the development of his prototype applications: "Without the toolkit, I probably wouldn't have thought to build [the arm shadows application]. The difficulty in building the multiple-mice thing would have been prohibitive".

In summary, a new area of research has been founded through the development of several novel applications combining SDG and distributed systems. Using the SDG Toolkit and the Collabrary Toolkit as enabling technologies, Tang was able to rapidly prototype a distributed SDG drawing application, evaluate it and iterate on the design to create a novel arm shadows drawing application [Tang, et al., 2004].

5.2.4 SDG EdgeLens.NET

The master's thesis work of Nelson Wong deals with the viewing of large complicated graphs with many nodes and edges. In these situations edge congestion can occur causing some nodes or edges to appear occluded. The bottom of Figure 5.12 shows a straight line with 3 nodes. We do not actually know how many edges are connected to each node as the edges may be overlapping. The edge lens technique bends edges around a point of focus and fades distant edges so that occluded edges can be revealed [Wong, et al., 2003].

SDG Edge Lens.NET is a toolkit for building applications that allow multiple edge lenses to be moved on a graph (Figure 5.12). It was developed as a curiosity-driven side project by Nelson Wong and Tony Tang, where it was ported to Microsoft Visual C# as a reusable library. The graphical component of this port allowed the end programmer to load graphs from files and view them using the edge lens, and would visualize each point of focus with a coloured dot (Figure 5.12, middle). The SDG EdgeLenses.NET application used the input from the SDG Toolkit to control the multiple focal points of the EdgeLenses.NET control. The visualization of the SDG cursors was disabled since a coloured cursor was already provided by the EdgeLenses.NET control.

This entire application (including the EdgeLens.NET toolkit) was developed in only two days. When asked about the total amount of time spent using the SDG Toolkit Tony Tang responded "Generating the SDG EdgeLens.NET application was trivial -- probably somewhere in the neighbourhood of 15 minutes." Since Tony Tang developed this application several months after using the SDG Toolkit for the Arm Shadows application (Section 5.2.3), this example shows that SDG Toolkit developers can quickly recall their experiences with the SDG Toolkit to create new and novel multi-user

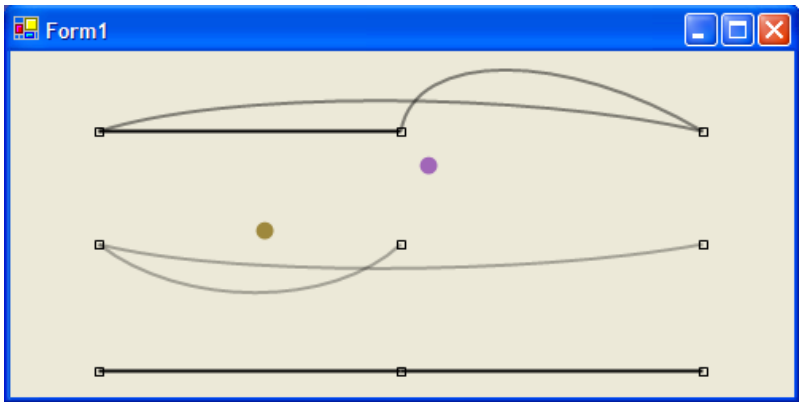


Figure 5.12. SDG edge lenses .NET. Users move the coloured dots (edge lenses) to reveal hidden edges in graphs like the one on the bottom.

applications. To quote Tony Tang "the design decisions (of the SDG Toolkit) were fairly appropriate for the applications that we have built… I think SDGT as it stands right now is fairly good for what it does, and the kinds of applications that we build with it".

In summary, this example shows how quickly a single user application can be made to support multiple users using the SDG Toolkit. It also shows how programmers can quickly reapply their skills with the SDG Toolkit to new novel applications even after months of inactivity.

5.3 Case Studies of Generalizing the SDG Toolkit Principles

This section validates the internal architectural principles behind the SDG Toolkit (described in Chapters 3 and 4) by showing how they can be applied to the design of toolkits for large vertical and horizontal displays using alternate pointing devices. This is important because it shows that the SDG Toolkit design philosophy is not strictly bound to multiple mice and keyboard systems.

I first describe how the SDG Toolkit principles can be generalized to large display pointing devices. Next, I give two case studies where I extended the SDG Toolkit principles to a camera based 3D wand input system, and to the Digital Vision Technology

Column	1	2	3	4	5	6	7
	Input Provided by the Low Level SDKs		Features Provided by the SDG Toolkit				
Input System	Low Level Event Mechanism	Input provided by SDK using Low Level Mechanisms	Input Provided to Programmer as Unique Events	Input Feedback	Table Top Support	System Mouse Handling	SDG Widget Layer
Multiple Mice	Windows Callback	Delta X, Y Mouse ID Buttons Pressed	Screen or Window X, Y Mouse ID Buttons Pressed	Multiple Cursors	Added	Added	Added
Open CV	Windows Callback	Absolute X, Y, Z within an arbitrary 3D space Point ID	Absolute X, Y, Z within an arbitrary 3D space Point ID	None	None	N/A	None
DViT Smart Board	Class Method Callback	Absolute Window X, Y Point ID Point Size	Absolute Window X, Y Point ID Point Size Bounding Region	Simple Cursor for each point Bounding Box for each point size Bounding Box for bounding region	None	N/A	Added
Diamond Touch	Continuous Polling	Absolute Board X, Y User ID Bounding Region Array of Grid Values	Absolute Window X, Y User ID Bounding Region Array of Grid Values	None	None	N/A	None

Table 5.1. Comparison of different input systems and their respective SDG Toolkits

(DViT) Smart Board. Finally, I describe a case study where another researcher extended the SDG Toolkit principles to the Diamond Touch surface.

5.3.1 Generalizing the Principles of the SDG Toolkit

Although the principles of the SDG Toolkit appear tuned to managing multiple mice and keyboards, they can be generalized to support other input devices. To foreshadow what is to come, Table 5.1 summarizes the input provided by different SDG input systems and how they translate to the features provided by the SDG Toolkit. I use Table 5.1 for comparison and reference when discussing the generalization of the SDG Toolkit principles.

Multiple Input and Identification: The toolkit needs to layer low-level SDK and system level dependencies, where it guarantees that input from multiple devices can be passed onto higher layers as separate streams (Column 2, Table 5.1). Most large display pointing devices provide Software Development Kits (SDKs) that allow programmers to access input through specialized low level event mechanisms (Column 1, Table 5.1). The lower layers of the toolkit need to handle these SDKs so that they have a minimal impact on higher layers.

Uniquely Identified Input Events with Absolute Coordinates Relative to the Screen or Working Window: The toolkit presents a uniquely identified input API with absolute coordinates relative to the screen and/or working window specified by a `RelativeTo` property. Different systems deliver point input in different styles: absolute board coordinates, relative delta coordinates, etc. These must be transformed as needed (Column 3, Table 5.1).

Extra Input Event Information: Extra features provided by the SDKs, peculiar to that input device, such as pointer sizes and bounding regions should be made available to the end programmer as additional parameters in each input event. While some SDKs require the programmer to write extra code to access extra event information the toolkit should make this information readily available to the programmer (Column 3, Table 5.1).

Input Feedback: The toolkit should automatically provide real time graphical feedback of the input that is efficient and customizable. While it makes sense for input from multiple mice to be shown as multiple cursors, we can adapt the graphical feedback so it is appropriate to the input provided by large display input systems. For example, the SDG Toolkit for the DViT Smart Board automatically includes optional graphical feedback of the bounding region and point sizes as an overlay on top of the application window (Column 4, Table 5.1)

Supporting Horizontal and Vertical Displays: The toolkit should reorient pointer input for different seating positions of the end user on a table. While this is an issue for low level SDKs such as Raw Input that provide delta coordinates; no reorientation of the input is needed by the toolkit if the pointing devices provide absolute coordinates. However, if the input feedback provided is orientation sensitive (e.g., with an arrow cursor) then the input feedback needs to be rotated for each user accordingly (Column 5, Table 5.1).

Dealing with the System Mouse: The toolkit should prevent the system cursor from causing unexpected results when multiple people interact simultaneously. If the input pointing device does not contend for control of the system mouse, then this will not be a large problem. Of course, the problem of having certain window system features responding only to the system mouse will remain (Column 6, Table 5.1).

SDG Widgets: The toolkit should provide an SDG widget infrastructure that handles all underlying '"plumbing"', i.e., where it automatically detects and sends input events to SDG widgets. These input events should be aware of the extra features provided by the low level SDKs. The toolkit should also provide a set of basic extensible widget primitives. For example, the SDG Toolkit extension for the DViT Smart Board includes a set of widget primitives that is fully aware of the point sizes and bounding regions provided by the low level Smart Board SDK (Column 7, Table 5.1).

Now that we have explored how the principles of the SDG Toolkit generalize to novel input devices we will examine three case studies of how these principles have been applied in practice.

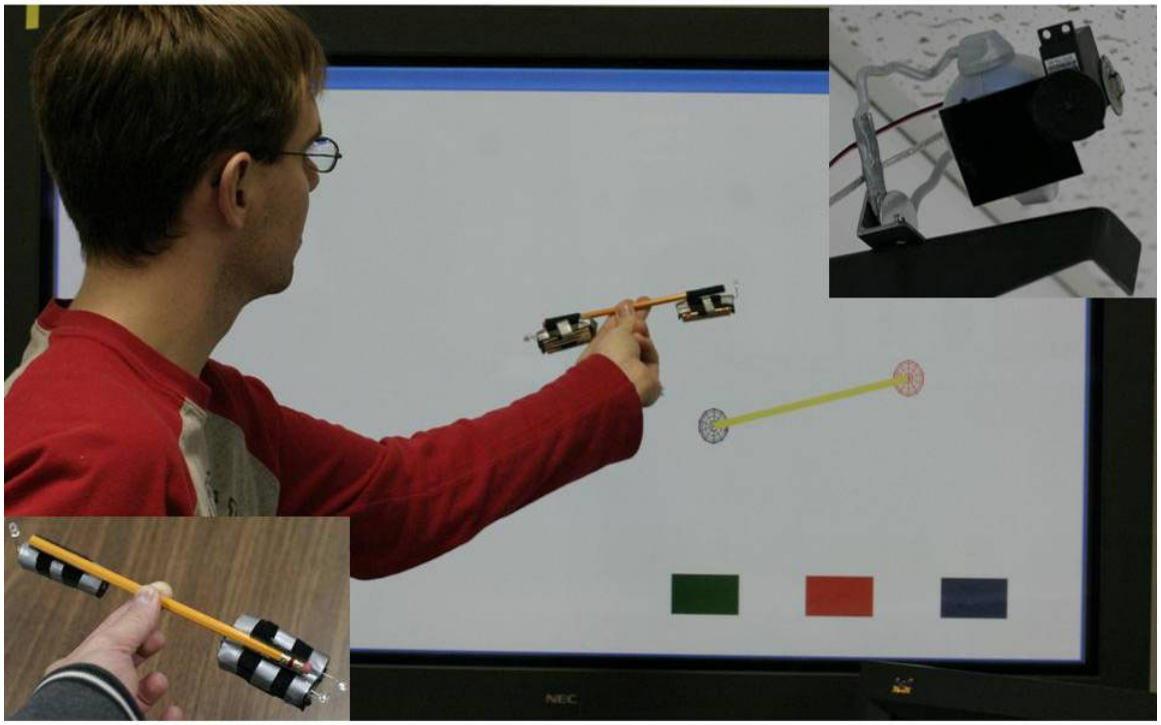


Figure 5.13. The 3D wand system: Infrared LEDs (bottom left) are tracked by two web cameras with infrared filters (top right) allowing LEDs to be tracked in three dimensional space as seen in this simple application tediously built from scratch without the Open CV Toolkit.

5.3.2 Three Dimensional Wand Input

In a separate project, a student researcher within our laboratory prototyped a 3D wand as an input device for a plasma display. This was implemented by using a vision system to track two infrared light emitting diodes (IR LEDs) at either end of the wand using two web cameras, as seen in Figure 5.13. The vision software was complex and difficult to use as it revealed the underlying workings of the Open CV vision library.

I was asked to simplify the 3D wand interface. To do so, I wrapped the functionality of the Open CV utility application. In addition to providing X, Y, Z coordinates and IDs for each IR LED tracked, I added functionality to configure cameras and to adjust the number of IR LED points to track. The 3D wand input is accessible through the following SDG Toolkit functionality:

1. **Multiple Input and Identification**: The 3D Wand Input wraps the complexity of Open CV by presenting input from the 3D wand LEDs as separate streams of X, Y and Z coordinates and an associated ID.

2. **Uniquely Identified Input Events and Extra Event Information**: The 3D Wand Input converts the input (a Microsoft Windows callback mechanism) into uniquely identified input events. These events are similar to those provided for SDG Mice except that a Z component is added. Since this application simply wraps the events provided by Open CV the event coordinates are not relative to the screen or working application window, they are absolute coordinates within an arbitrary 3D space.

3. **Supporting Tables and Vertical Displays**: Since all the coordinates provided by the Open CV wrapper are absolute, this input can be used on a display projected onto a table. However, if the graphical feedback is orientation sensitive then it would need to be rotated for each user. No graphical feedback was provided in this version of the 3D wand input.

While I had no prior experience with the Open CV libraries, I built this wrapper in only two days using my experience in developing the SDG Toolkit for multiple mice.

As a class project undergraduate Anand Agarawala was able to prototype a 3D graph application called Super Skewer using the 3D Wand Input. The bottom right of Figure 5.14 illustrates how he visualized the three dimensional wand. His application

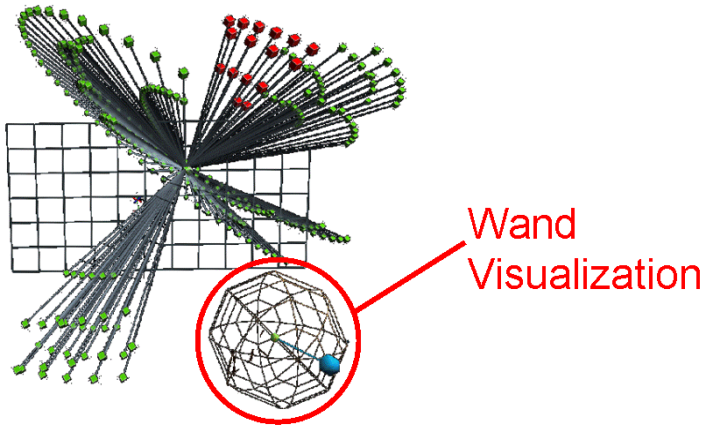


Figure 5.14. SuperSkewer: A 3D graph can be rotated and manipulated using two sets of IR LEDs. A 3D visualization of the infrared LEDs is shown in the bottom right of this application.

allows an entire graph to be rotated and moved, selected nodes can be moved and manipulated using 3D interaction techniques. While this application is designed for a single user, interaction with multiple wands is possible (in principle) with the 3D Wand Input.

5.3.3 The DViT Toolkit

Members of our lab wanted to study interaction techniques involving multiple people over a large table surface. To do this we purchased a Digital Vision Technology (DViT) Smart Board, a large interactive whiteboard that supports multiple points of contact (Figure 5.15). The DViT Board can track the 2D position and size of up to two touches and unlike the 3D Wand Input, the DViT board supports both finger and pen input. This technology can be used on plasma sized displays and large projected screens.

To explain how this works, the DViT has an infrared camera on each corner of the board that observes an of array infrared LEDs on each side of the display (Figure 5.15, bottom left). When a pen or finger is moved over the board, changes are observed by the four infrared cameras. Using these images the DViT Hardware calculates the position and size of each point and makes this information available through the low level Smart

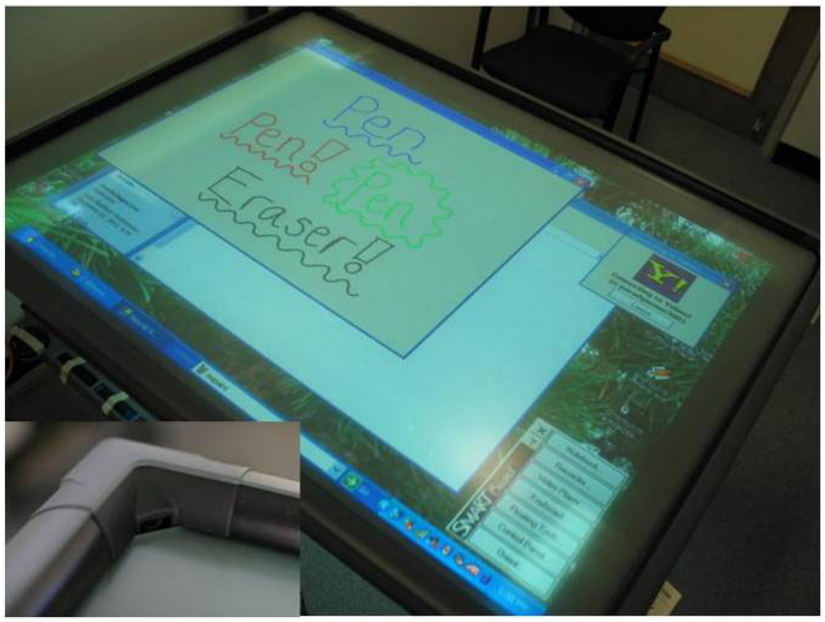


Figure 5.15. The DViT board: Four Infrared Cameras are located on each corner of the display. On each side of the display is an array of infrared LEDs

Board SDK.

One of the limitations of the DViT Board is that it cannot uniquely identify users. This means that the ID provided by the Smart Board SDK may change if a person lifts their finger or pen from the display. This greatly increases the difficulty of tracking individual movements.

We wanted to exploit the multiple pointer capabilities of the DViT by wrapping the complexity of the Smart Board SDK. Using the principles of the SDG Toolkit I developed the DViT Toolkit during an internship at Smart Technologies. The DViT Toolkit is accessible through the following SDG Toolkit functionality:

1. **Multiple Input and Identification**: The DViT Toolkit wraps the complexity of the Smart Board SDK by presenting input from multiple contact points as separate streams of X and Y coordinates with an associated ID.
2. **Uniquely Identified Input Events**: The DViT Toolkit converts the input (a set of class method callbacks) into uniquely identified input events with absolute coordinates relative to the screen or working window. These events are similar to those provided for SDG Mice (e.g., Touch Down, Touch Up, Touch Move and Double Touch).
3. **Extra Event Information**: The point sizes are presented as additional parameters in each DViT input event. To simplify the tracking of scaling gestures, I provided an event that represented changes in the bounding region between the two points of contact. Also, an event indicating the currently selected pen is fired when a pen is removed or returned from the pen tray.
4. **Translating Pointer Data to Window Coordinates**: The DViT Toolkit provides two dimensional coordinates relative to the screen or application window through the use of a `RelativeTo` property like in the SDG Toolkit.
5. **Supporting Tables and Vertical Displays**: Since the Smart Board provides absolute coordinates, the input does not need to be modified for use on a table. However, if the graphical feedback was orientation specific, then it would need to be reoriented for each person's seating position around the table.

6. **Graphical Feedback**: Instead of providing multiple cursors, the DViT Toolkit provides graphical feedback in the form of a simple cursor for each point, a bounding box for each point size (Figure 5.16) and, a bounding box for the bounding region (Figure 5.17)
7. **SDG Widgets**: Finally, a specialized widget layer for the DViT Toolkit was also provided. This widget layer is the same as the SDG Toolkit widget layer except that it uses DViT events with extra properties such as the size of each point of contact.

To test the DViT Toolkit, I developed a special white board application that used pointer sizes to distinguish between a pen and an eraser. This application mimics the behaviour of a real whiteboard, pens can be picked up from the pen tray and used to draw different colours on the Smart Board. If a point on the DViT is larger than a pen tip it is recognized as an eraser of the corresponding size, for example, Figure 5.16 shows a hand used as an eraser. This means that a finger would become a small eraser while an arm would be a very large eraser.

This application made use of the pointer size visualization provided by the DViT Toolkit. As seen at the bottom left of Figure 5.16, the hand is covered by a rectangle that approximates the actual size of the hand. This visualization allows the end user to see the areas of the screen that will be affected when the eraser is used. Using the DViT Toolkit

Figure 5.16. A DViT drawing application: This drawing application is unique in that the size of the pointer is used to determine if the user is currently using a pen or using an eraser on the board. This image also shows the pointer size visualization with the red box over the hand on the left.

this entire application took only two hours to develop.

Using the widget layer of the DViT Toolkit I developed a widget that could be scaled, rotated and translated using a gesture with two hands. As seen in Figure 5.17, a person could perform these transformations by moving the corners of an image. The widget could be moved by touching anywhere in the middle of the widget.

During the development of this application I noticed that I would often miss the small target of the corners of the image. Using the visualization provided by the DViT Toolkit (Figure 5.17) I was able to quickly determine that this problem was caused by jittery input from Smart Board SDK. To resolve this issue, I made the targets on the corners of the image larger but realized that some smoothing of the input from the DViT Board would be needed in a future version of the toolkit. This entire application was developed over three days in which almost all of the effort was spent doing the math for scaling, rotating and translating an image. Only two hours of my development time were spent making my image manipulation application respond to two inputs on a DViT board.

In summary, this case study illustrates how most of the principles of the SDG Toolkit can be applied to novel input devices in practice. It showed that principles such

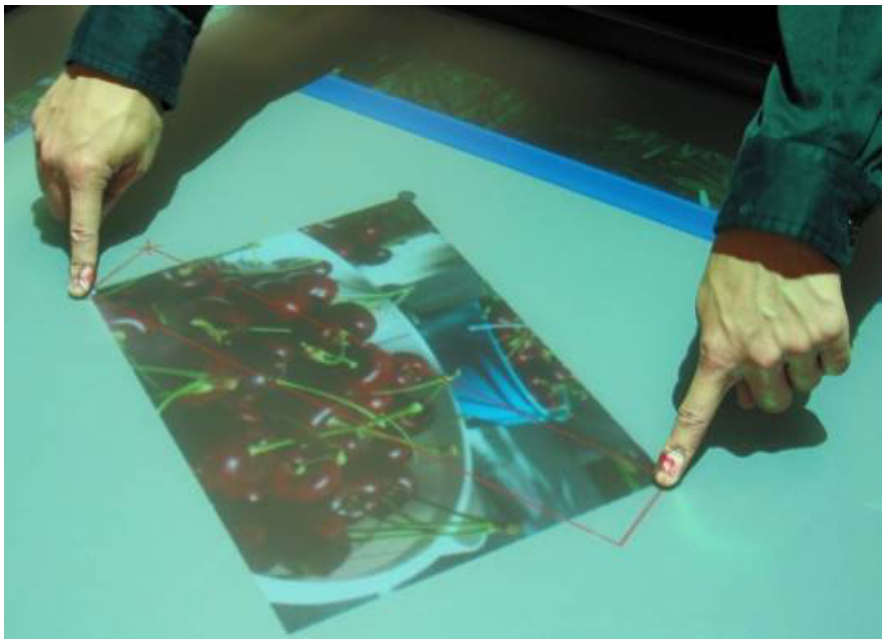


Figure 5.17. A DViT picture manipulation example: Users can scale, rotate and translate the image by moving two of its corners. This image also shows the bounding region visualization of the DViT Toolkit with the red box between the two fingers.

as the provision of graphical feedback could be generalized to novel input devices. Through two example applications I showed how the DViT Toolkit simplified the prototyping and debugging of applications using the DViT Smart Board.

5.3.4 The Diamond Touch Toolkit

Members of our lab wanted to be able to be able to uniquely identify multiple users at a table top display. To do this we obtained a touch sensitive board known as Diamond Touch [Dietz and Leigh, 2001] that could identify up to four different users with no limitations on the number of points of contact.

To explain how this works, the Diamond Touch emits a small electrical signal through a number of capacitive pads. As seen in Figure 5.18, each user of the system comes in contact with a different signal by sitting on a capacitive pad. This signal is detected by an array of antennas located just below the Diamond Touch surface. These signals are processed in the computer and made available to the programmer through the low level Diamond Touch SDK. One of the limitations of the Diamond Touch board is that its size is limited to a display slightly smaller than a plasma display.

Rob Diaz-Marino, a summer intern in our lab, was asked to use the principles of the SDG Toolkit to simplify the design of applications with the Diamond Touch [Diaz-

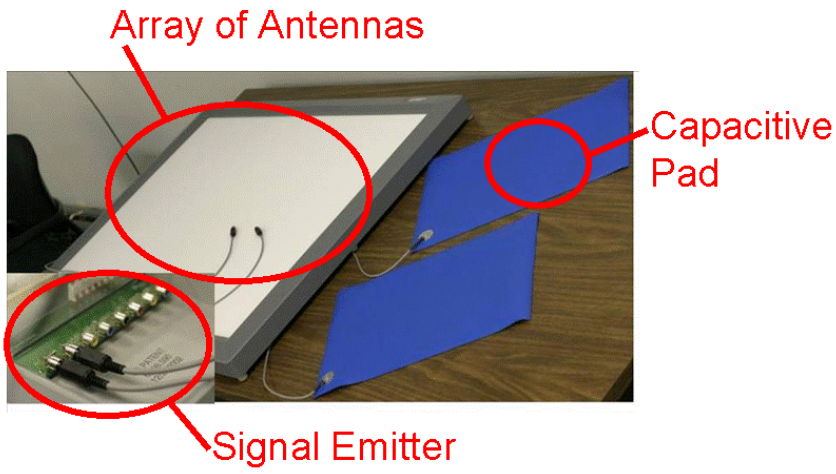
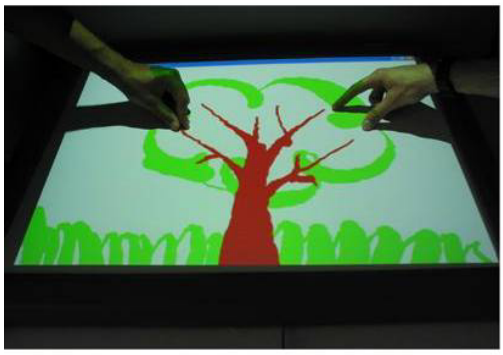


Figure 5.18. The Diamond Touch board and capacitive pads: Each pad is connected to a signal emitter located on the back of the board (bottom left) and the signal that passes through the body is received by an array of sensors under the surface of the board.

Marino, et al., 2003]. I emphasize that Diaz-Marino had no prior experience with the SDG Toolkit principles, the Diamond Touch SDK or the Microsoft Visual Studio Programming environment. Despite these hurdles Diaz-Marino was able to develop a prototype version of the toolkit in only one week. His toolkit includes the following SDG Toolkit features:

1. **Multiple Input and Identification**: The Diamond Touch Toolkit wraps the complexity of the Diamond Touch SDK by continually polling the Diamond Touch hardware and presenting input from multiple users as separate streams of X and Y coordinates with an associated ID.
2. **Uniquely Identified Input Events**: The Diamond Touch toolkit converts the input into uniquely identified input events with absolute coordinates relative to the screen or working window. These events are similar to those provided for SDG Mice (Touch Down, Touch Up, Touch Move and Double Touch). Events are fired regularly using a timer so that the end programmer does not need to manually poll the input device.
3. **Extra Event Information**: The Diamond Touch Toolkit also provides a bounding region event that is useful for tracking scaling gestures and two arrays representing the Raw Input seen by the antennas for each user.
4. **Translating Pointer Data to Window Coordinates**: The Diamond Touch Toolkit handles the conversion of board coordinates to absolute coordinates relative to the screen or application window through the use of a `RelativeTo` property like in the SDG Toolkit.



```
private void dtManager_TouchBoxMove( TouchEventArgs e )
{
    Graphics g = this.CreateGraphics();
    if (e.User.LastDown.Timestamp < e.User.LastBox.Timestamp)
    {
      Pen drawpen = new Pen(Color.Black, 1);
      drawpen.StartCap = System.Drawing.Drawing2D.LineCap.Round;
      drawpen.EndCap = System.Drawing.Drawing2D.LineCap.Round;
      Rectangle lastbox = e.User.LastBox.Box;
      Rectangle newbox = e.Box;
      int thickness = (newbox.Width + newbox.Height) / 10;
      if (thickness < 2) thickness = 2;
      Point lastpoint = new Point(lastbox.X + (lastbox.Width / 2),
          lastbox.Y + (lastbox.Height / 2));
      Point newpoint = new Point(newbox.X + (newbox.Width / 2),
          newbox.Y + (newbox.Height / 2));
      drawpen.Color = e.User.Color;
      drawpen.Width = thickness;
      g.DrawLine(drawpen, lastpoint, newpoint);
    }
}
```

Figure 5.19. The scribble draw application and source code: Programmers manipulate the widget of their coloured drawing by adjusting the bounding region that is made by their hands. Photo from Diaz-Marino, et al., 2003

5. **Supporting Tables and Vertical Displays**: Since the Diamond Touch provides absolute coordinates, the input does not need to be modified for use on a table. However, if the graphical feedback was orientation specific, then it would need to be reoriented for each person's seating position around the table.

Using the Diamond Touch Toolkit, Rob Diaz-Marino developed several compelling example applications to illustrate its ease of use. The first example seen in Figure 5.19 is a drawing application called Scribble Draw that uses the size of the bounding region for each user to determine the thickness of their lines. The tree and grass drawing seen on the left hand side of Figure 5.19 was created by varying the line thicknesses using two fingers.

Another application developed by Rob Diaz-Marino using the Diamond Touch Toolkit was a two-player memory game. In Figure 5.20, we see two people interacting with the memory game. Each player could flip up to two cards by either touching the surface of a card on the display or scaling the card as seen at the bottom of Figure 5.20. If a player flipped over two identical cards a point would be awarded and these cards would be removed from the pile. When all of the cards had been removed from the pile, the player with the most points would be declared the winner.

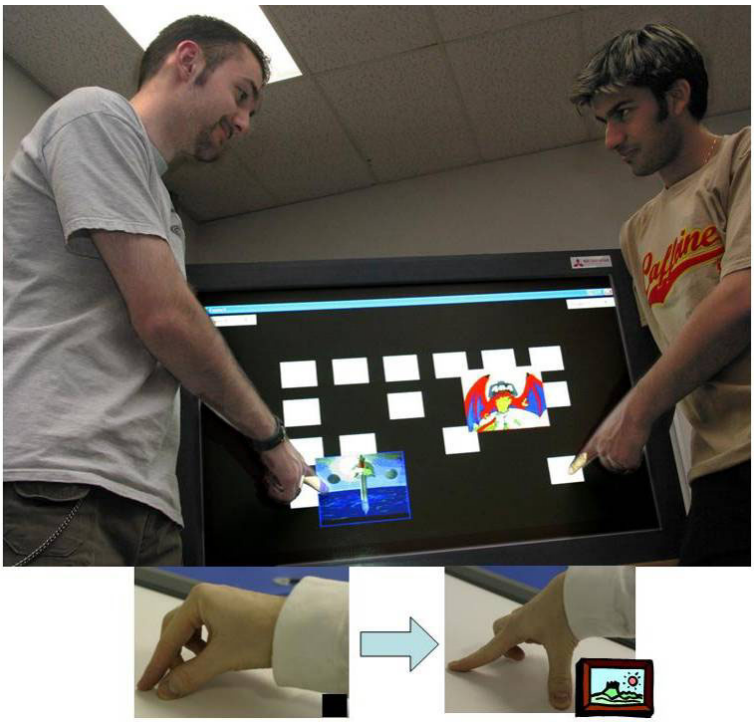


Figure 5.20. The Diamond Touch memory game: Users select cards by touching the board and flip and size them using a finger pushing gestures like the ones seen at the bottom of this Figure.

Finally, Rob Diaz-Marino developed a musical application called Sound Board that used the Diamond Touch to control the volume and balance various tracks in a song. In this application, a number of spheres representing sound tracks (e.g., percussion, bass) in a song are animated using a physics engine that handles collisions against walls and other spheres. As seen in the top right corner of Figure 5.21, the motion of an unselected ball is visualized using a blue fading trail of spheres.

These spheres could be controlled by either touching them or surrounding the sphere through a person's bounding region. When caught, the spheres become enlarged with a different colour for each participant. Selected spheres move with the person's hand movement and are animated with semi transparent spheres that fade to black as they move away from the selected sphere. The horizontal position of the sphere controls the balance of the sound track and the vertical position of the sphere controls the volume of the sound track. For example, a sphere placed on the top left of the screen is heard

exclusively on the left speaker with a high volume, whereas a sphere placed in the

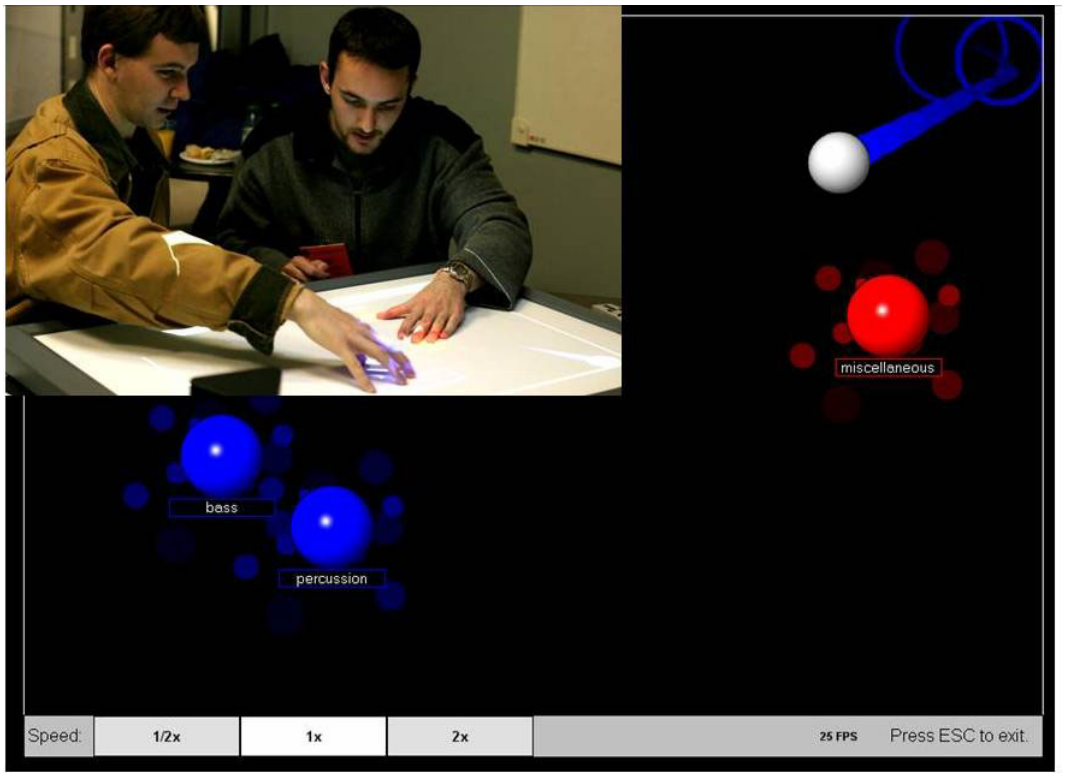


Figure 5.21. The Diamond Touch sound board application: Multiple users can catch moving balls that represent different tracks of a song. Each track can be panned to the left or right speaker by moving the balls left or right. Each track can be increased or decreased in volume by moving the balls up or down.

bottom right of the screen is very faintly heard on the right speaker.

In summary, the Diamond Touch Toolkit case study illustrates how other researchers have been able to extend the principles of the SDG Toolkit to other novel input devices. It also shows how a device that presents input through a complex polling procedure can be presented to the end programmer in a way that is similar to mouse events. Finally, the case study examples show how the development of the Diamond Touch Toolkit allowed applications to be rapidly prototyped using the Diamond Touch surface.

5.4 Conclusion

My goal in this chapter was to validate design of the SDG Toolkit. I first validated how SDG applications could be prototyped quickly and easily using the SDG Toolkit. I did this by delivering several case studies of how I and others used the SDG Toolkit to develop SDG applications and widgets. All case studies revealed that researchers using the SDG Toolkit were able to focus on the design of their applications rather than worrying about the technical issues of handling multiple input devices. I then validated the generalizability of the SDG Toolkit infrastructure by describing how its design could be adapted to three input devices for large displays (3D wand input, the DViT Smart Board, and the Diamond Touch board).

Finally, to validate how the principles of the SDG Toolkit extend to other novel input devices I described three case studies. In the first case study I described how a few of the principles of the SDG Toolkit had been applied to the 3D wand input. Even though this system was different from multiple mice and keyboards, many of the same principles still apply. Next I described a toolkit for the DViT Smart Board that implemented almost all of the principles of the SDG Toolkit. This case study showed the full potential of the SDG Toolkit principles in practice. I then described the Diamond Touch Toolkit developed by Rob Diaz-Marino. This case study showed how others have used the principles of the SDG Toolkit for large display input devices.

These case studies serve to show that applications can be rapidly prototyped using the SDG Toolkit (thesis goals 1 and 2) and that its principles extend to other large display input devices (claimed in Chapter 3).

Chapter 6. Conclusion

This chapter concludes the thesis with discussions of my research contributions. First, I recap the thesis problems discussed in Chapter 1. This is followed by a detailed description of how each thesis problem is solved in the SDG Toolkit. Next, I detail some of the future work I have done and some of the future work that I plan on pursuing. I then conclude this thesis with a discussion of my research and its viability in the future.

6.1 Thesis Problems

In Chapter 1, I outlined two problems that arise for the development of single display groupware applications.

1. **We do not have a means to quickly and rapidly develop SDG applications using multiple mice and keyboards.** The assumptions present in today's operating systems hinder the development of SDG applications by presenting the following hurdles to SDG development:

    a) ***Multiple input and identification***: Since computers assume that there will only be one mouse or keyboard per system, applications that support multiple mice and keyboards often require programmers to build their own device drivers or use low level APIs to develop even the most basic SDG applications.

    b) ***Cursor drawing***: Even after these basic SDG input problems have been resolved there is still a need to draw an individual pointer or cursor for each mouse. Often manually drawing a cursor for each mouse adds additional work for developers and results in sluggish application performance if not done efficiently.

c) ***Table top Support***: Current systems assume a single orientation for each mouse, but SDG researchers need to support different seating orientations when designing table top applications.

2. **We do not have a means to develop SDG aware widgets**: Just as widgets encapsulate common single-user interaction techniques, so should SDG interaction widgets contain multi-user interaction methods. Yet SDG widgets will be quite different, for they need to understand concurrent use. To create such widgets, a suitable infrastructure is needed.

   a) ***SDG widget infrastructure***: In order for SDG widgets to work there must be a way for SDG widgets to communicate with the SDG input API. A low level infrastructure must exist to identify SDG widgets and send notifications of multiple mouse movements and keyboard events. Currently, researchers write extra code to allow SDG widgets to communicate with SDG input. This results in extra programming effort required by the programmer and the risk of sluggish application performance if done crudely.

   b) ***Basic SDG widget primitives***: To prevent all widgets from being developed from scratch we need to provide basic SDG widget building blocks as a starting point for developing more complex widgets. We do not have the basic widget building blocks needed to rapidly prototype simple and novel SDG widgets.

6.2 Thesis Contributions

In this thesis, I present the following research contributions, each corresponding to a solution of the above problems. A comparison of the SDG Toolkit and SDG input systems can be found on Table 2.2.

6.2.1 The SDG Toolkit

We do not have a means to quickly and rapidly develop applications using multiple mice and keyboards: This thesis detailed the design, implementation and validation of a

toolkit for rapidly prototyping Single Display Groupware applications called the SDG Toolkit. It provides the following features:

a) ***SDG input API:*** In Chapter 3, I described how the SDG Toolkit interface allows programmers to access input from multiple mice and keyboards as separate streams. Using a layered architecture the SDG Toolkit hides any system dependencies; this also means that other input devices can be substituted if needed. Mouse and keyboard events are generated in a style that is similar to how programmers handle input events except it adds an additional ID parameter that allows the programmers to uniquely identify the device that generated it.

b) ***Automatically draws multiple cursors:*** In Chapter 3, I described how the SDG Toolkit displays a cursor for each connected mouse so that programmers will not need to add cursor drawing code into their own applications. Both the text caption and the cursor image can be customized by the end programmer. The SDG Toolkit ensures that cursors are drawn efficiently.

c) ***Provides table top support:*** In Chapter 3, I described how the SDG Toolkit provides a means for programmers to design applications that recognize different orientations of multiple mice over a table top display. This is done by rotating incoming mouse movement information and reorienting graphical cursors and text captions so they are upright for the person's orientation around the table.

The above points mention 'efficiency', and this should be clarified. While it could be argued that today's fast hardware makes this point moot in comparison to earlier SDG systems, this is not the case. My first implementation of SDG Toolkit used a brute force technique, where I simply responded to all Raw Input events as they occurred, and where I did not take any special care in how cursors were drawn. In retrospect, this proved naïve as frequent cursor movements and end programmer mouse movement routines compromised application performance. The throttling cursor movements and input events were necessary to ensure that SDG applications were responsive even when multiple mice were moved simultaneously. Similarly, the implementation choice of cursors as transparent windows was taken because window movement is handled in hardware on modern video cards.

The SDG Toolkit also provides programmers with workarounds for handling issues caused by the one user model of modern windowing system. In Chapter 3, I described how the toolkit provides several alternatives for handling the actions of the system cursor, where it disables system cursor functionality by default. Finally, the SDG Toolkit provides a focus widget property for each mouse and provides facilities to allow each keyboard to be mapped to a particular mouse focus. By default each keyboard is associated with the corresponding mouse of the same identification number.

In Chapter 5, I validated the effectiveness of the toolkit for prototyping SDG applications and showed how the SDG Toolkit allows programmers to focus on the design of SDG systems rather than worrying about the technical issues of handling multiple input devices.

6.2.2 An SDG Widget Layer

We do not have a means of easily prototyping SDG aware interaction widgets: To allow programmers to rapidly prototype SDG widgets I added a widget layer to the SDG Toolkit. The widget layer provides the following features:

a) ***An SDG widget infrastructure:*** The infrastructure automatically detects SDG widgets and sends notifications of input events. It provides a standard way of communicating with widgets through SDG aware interfaces.

b) ***SDG widget primitives:*** The widget layer provides two inheritable widget primitives that can be used as building blocks for novel SDG widgets. Both the SDG Control and User Control implement the SDG widget interfaces so that programmers can immediately take advantage of the SDG functionality provided. The SDG Control provides basic functionality, while the SDG User Control allows programmers to develop widgets using the Microsoft Visual Studio interface designer.

In Chapter 5, I validated the effectiveness of the widget layer for prototyping novel SDG widgets through several case studies. The case studies showed that most of the development effort was spent working on interaction issues rather than worrying about

communicating with widgets, and that new widget classes could be rapidly constructed to handle unanticipated needs.

6.2.3 From Replication to Empiricism

Gaines' (1986) BRETAM phenomenological model of developments in science technology states that technology begins with an insightful and creative *B*reakthrough, followed by many (often painful) *R*eplications and variations of the idea. *E*mpiricism occurs when people draw lessons from their experiences and formalize them as useful generalizations. This continues to *T*heory, *A*utomation and *M*aturity.

Within this context, the primary contribution of this thesis is to move SDG technical work from the painful replication stage (where it is now) into the empiricism stage. I did this through several mechanisms. First, I articulated the technical requirements and challenges of SDG software that face many designers by examining common features provided by other toolkits in the literature in Chapter 2. These are the empirical lessons obtained by looking at a number of breakthrough and replication systems. Second, I detailed solutions to these problems in Chapters 3 and 4. These solutions codify a conceptual model for other toolkit builders about how a toolkit for SDG should present itself. Finally, I provided the SDG Toolkit itself as a resource so that SDG application developers can focus on the nuances of SDG and SDG interaction techniques rather than replicate SDG "plumbing".

6.3 Future Work

The future work of the SDG Toolkit follows several threads, I have already begun some of them in this thesis.

First I would like to develop a complete SDG Widget library that includes multiple user analogues of all single user graphical widgets, as well as innovative new approaches to SDG interaction. To do this, I will need to study how multiple user widgets are used in a collaborative setting to determine the ideal method for designing such widgets.

Second, I would like to extend the SDG Toolkit to other pointing devices for large displays. This has already been done for 3D Wand Input, the DViT Smart Board and the Diamond Touch Board, but it could also be extended to, say, multiple laser tracking technologies such as those developed at the University of British Columbia [Vogt, et al., 2003]. All of the existing extensions of the SDG Toolkit have been distinct toolkits; it would be more efficient to have a generalized architecture for handling SDG input.

6.4 Conclusion

In this thesis I explored issues related to the development of applications that support multiple people. I described the problems associated with developing Single Display Groupware applications today and I developed a conceptual model for building a toolkit to resolve these issues. I implemented this solution in a software toolkit called the SDG Toolkit and made it available to other researchers so that they could focus on the design of their applications rather than worrying about the underlying "plumbing" involved.

If one looks down the road a few years from now, it is hard to imagine future computers that are not SDG-capable. While this functionality could be achieved through an add-on such as the SDG Toolkit, at some point, our windowing systems should have SDG built into them as a fundamental component. Perhaps the concepts introduced in this thesis will influence how this is done.

References

1. Bederson, B. and Hourcade, J. (1999), **Architecture and implementation of a Java package for Multiple Input Devices (MID)**, *HCIL Technical Report No. 9908*, http://www.cs.umd.edu.hcil.

2. Bederson, B., Hollan, J., Druin, A., Stewart, J., Rogers, D. and Proft, D. (1996), **Local Tools: An Alternative to Tool Palettes**, *Proceedings of the ACM Conference on User Interface and Software Technologies (UIST '96)*, Seattle, pp.169-170.

3. Bier, B. and Freeman, S. (1991), **MMM: A user interface architecture for shared editors on a single screen**, *Proceedings of the ACM Conference on User Interface and Software Technologies (UIST '91)*, Hilton Head, pp. 79-86.

4. Bier, E., Stone, M., Pier, K. and Buxton, W. and DeRose, T. (1993), **Toolglass and Magic Lenses: The See-Through Interface**, *Proceedings of the ACM Conference of Computer Graphics and Interactive Techniques (SIGGRAPH '93)*, Anaheim, pp. 73-80.

5. Bishop, G. and Welch, G. (2000), **Working in the office of "Real Soon Now"**, *IEEE Journal of Computer Graphics and Applications*, Volume 20 Issue 4, July/August, pp. 76-78.

6. Boyle

7. Boyle, M. and Greenberg, S. (2002), **GroupLab Collabrary: A Toolkit for Multimedia Groupware***, Extended Abstracts of the ACM Conference on Computer Support Cooperative Work (CSCW '02), Workshop on Network Services for Groupware*, New Orleans.

8. Bricker, L., Baker, M., Fujioka, E. and Tanimoto, S. (1999), **A System for Developing Software that Supports Synchronous Collaborative Activities**, *Proceedings of EdMedia*, pp. 587-592.

9. Bricker, L. (1998), **Cooperatively Controlled Objects in Support of Collaboration,** *Ph.D. Thesis*, University of Washington, Department of Computer Science and Engineering, Seattle, March, 1998.

10. Cole, K., (1995), **Equity Issues in Computer-Based Collaboration: Looking Beyond Surface Indicators**, *Proceedings of the ACM Conference on Computer Supported Cooperative Learning (CSCL '95)*, Bloomington, pp. 67-74.

11. Diaz-Marino, R.A., Tse, E, and Greenberg, S. (2003), **Programming for Multiple Touches and Multiple Users: A Toolkit for the DiamondTouch Hardware,** *Extended Abstracts of the ACM Conference on User Interface and Software Technologies* (UIST '03), Vancouver.

12. Dietz, P. and Leigh, D. (2001), **DiamondTouch: A multi-user touch technology**, *Proceedings of the ACM Conference on User Interface and Software Technologies (UIST '01)*, Orlando, pp. 219-266.

13. Druin, A., Stewart, J., Proft, D., Bederson, B. and Hollan, J. (1997), **KidPad: a design collaboration between children, technologists, and educators**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '97)*, Atlanta, pp. 463-470.

14. Elrod, S., Bruce, R., Gold, R., Goldberg, D., Halasz, F., Janssen, W., Lee, D., McCall, K., Pedersen, D., Pier, K., Tang, J., and Welch, B. (1992), **Liveboard: A large interactive display supporting group meetings, presentations and remote collaboration**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, Monterey, pp. 599-607.

15. Gaines, B. and Shaw, M. (1986), **A Learning Model for Forecasting the Future of Information Technology**, *Future Computing Systems*, Volume 1(1), pp. 31-69.

16. Greenberg, S., Gutwin, C., and Roseman, M. (1996), **Semantic Telepointers for Groupware**. *Proceedings of the Australian Conference on Computer-Human Interaction (OzCHI '96)*, Hamilton, New Zealand, pp. 54-61.

17. Greenberg, S. (1999), **Designing Computers as Public Artifacts**, *International Journal of Design Computing*: *Special Issue on Design Computing on the Net (DCNet'99)*, University of Sydney, November 30 – December 3.

18. Greenberg, S., Boyle, M. and LaBerge, J. (1999), **PDAs and Shared Public Displays, Making Personal Information Public, and Public Information Personal**, *Personal Technologies*, Volume 3(1), Elsevier, pp. 54-64.

19. Greenberg, S. and Fitchett, C. (2001), **Phidgets: Easy Development of Physical Interfaces through Physical Widgets**, *Proceedings of the ACM Conference on User Interface and Software Technologies (UIST '01)*, Orlando, pp. 209-218.

20. Guimbretiere, F. and Winograd, T. (2000), **FlowMenu: Combining Command, Text, and Data Entry**, *Proceedings of the ACM Conference on User Interface and Software Technologies (UIST '00)*, San Diego, pp. 213-216.

21. Hill, J. and Gutwin, C. (2003), **Awareness Support in a Groupware Widget Toolkit**. *Proceedings of the ACM Conference on Supporting Group Work (Group '03)*, Florida.

22. Hourcade, J.P. (2003), **User Interface Technologies and Guidelines to Support Children's Creativity, Collaboration, and Learning**. *PhD Thesis*, University of Maryland, College Park.

23. Inkpen, K., Booth, K., Gribble, S. and Klawe, M. (1995), **Give and Take: Children Collaborating on One Computer**, *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI '95)*, Denver, pp. 258-259.

24. Inkpen, K., Booth, K. S. and Klawe, M. & Upitis, R. (1995), **Playing Together Beats Playing Apart, Especially for Girls**, *Proceedings of the ACM Conference on Computer Support for Collaborative Learning (CSCL '95)*, Bloomington.

25. Inkpen, K., McGrenere, J., Booth, K. and Klawe, M. (1997), **The effect of turn-taking protocols on children's learning in mouse-driven collaborative environments**, *Proceedings of the Graphics Interface Conference (GI '97)*, Kelowna, B.C., pp. 138-145.

26. Inkpen, K.M., Ho-Ching, W., Kuederle, O., Scott, S.D. and Shoemaker, G.B.D. (1999), **"This is fun! We're all best friends and we're all playing.": Supporting children's synchronous collaboration**, *Proceedings of Computer Supported Collaborative Learning (CSCL '99)*, Stanford, pp. 252-259.

27. Myers, B., Stiel, H. and Gargiulo, R. (1998), **Collaborations using multiple PDAs connected to a PC**, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'98)*, Seattle, pp. 285-294.

28. Nardi, B. and Miller, B. (1990), **An ethnographic study of distributed problem solving in spreadsheet development**, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, Los Angeles, pp 197-208.

29. Olsen, D. and Neilsen, T., (2001), **Laser Pointer Interaction**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '01)*, Seattle, pp. 17-22.

30. Roseman, M. and Greenberg, S. (1995), **Building Real Time Groupware with GroupKit, A Groupware Toolkit**, *ACM Transactions on Computer human Interaction (TOCHI '95)*, March, Volume (3)1, pp. 66-106.

31. Rumbaugh, J., Jacobson, I. and Booch, G. (2004), **Unified Modeling Language Reference Manual, 2nd Edition**, *Addison Wesley Object Technology Series*, Pearson Education, ISBN 0321245628.

32. Scott, S.D, Shoemaker, G.B.D, and Inkpen, K.M. (2000), **Towards Seamless Support of Natural Collaborative Interactions**, *Proceedings of Graphics Interface (GI '00)*, Montreal, pp .103-110.

33. Shen, C., Vernier, F., Forlines, C. and Ringel, M. (2004), **DiamondSpin: An Extensible Toolkit for Around-the-Table Interaction**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '04)*, Vienna, Austria, pp. 167-174.

34. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S. and Tatar, D. (1987), **WYSIWIS Revised: Early experiences with multi-user interfaces**, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '87)*, Austin, Texas, pp. 276-290.

35. Stewart, J., Bederson, B. and Druin, A. (1999), **Single display groupware: a model for co-present collaboration**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburgh, pp. 286-293.

36. Stewart, J., Raybourn, E. M., Bederson, B., and Druin, A. (1998), **When two hands are better than one: Enhancing collaboration using single display groupware**, *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI '98)*, pp. 287-288.

37. Streitz, N., Gießler, J., Holmer, T. and Konomi, S. (1999), **i-LAND: An interactive Landspace for Creativity and Innovation**, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburgh, pp. 120-127.

38. Tandler, P., (2003), **The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments**, *Journal of Systems & Software, Special Edition on Application Models and Programming Tools for Ubiquitous Computing*, October, 2003.

39. Tang, A., Boyle, M. and Greenberg, S. (2004), **Display and Presence Disparity in Mixed Presence Groupware**, *Proceedings of the Fifth Australasian User Interface Conference*, *In Conference in Research and Practice in Information Technology (CRPIT)*, Volume 28, Dunedin, New Zealand, pp. 73-82.

40. Tatar, D., Foster, G. and Bobrow, D. (1991), **Design for conversation: lessons from Cognoter***, Proceedings of the ACM Conference on Computer Supported Cooperative Work and Groupware (ECSCW '91)*, Amsterdam, pp. 55-79.

41. Tse, E. and Greenberg, S. (2002), **SDGToolkit: A Toolkit for Rapidly Prototyping Single Display Groupware**, *Extended Abstracts of the ACM Conference on Computer Supported Cooperative Work (CSCW '02)*, New Orleans, pp. 173-174.

42. Tse, E. and Greenberg, S. (2004), **Rapidly Prototyping Single Display Groupware through the SDGToolkit**, *Proceedings of the Fifth Australasian User Interface Conference*, *In Conference in Research and Practice in Information Technology (CRPIT)*, Dunedin, New Zealand, pp. 101-110.

43. Tse, E., Histon, J., Scott, S. and Greenberg, S. (2004), **Avoiding Interference: How People Use Spatial Separation and Partitioning in SDG Workspaces** *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '04)*, Chicago, to appear.

44. Vogt, F., Wong, J., Fels, S. and Cavens, D. (2003), **Tracking Multiple Laser Pointers for Large Screen Interaction**, *Extended Abstracts of the ACM Conference on User Interface and Software Technologies (UIST '03)*, Vancouver, pp. 95-96.

45. Wong, N., Carpendale, S. and Greenberg, S. (2003), **EdgeLens: An Interactive Method for Managing Edge Congestion in Graphs**, *Proceedings of the ACM Conference on Information Visualization (InfoVis '03)*, Seattle, pp. 51-58.

46. Zanella, A. and Greenberg, S. (2001), **Reducing Interference in Single Display Groupware through Transparency**. *Proceedings of the ACM Conference on Computer Supported Cooperative Work and Groupware (ECSCW'*01), Bonn, Germany.

Appendix A. Co-Author Permission

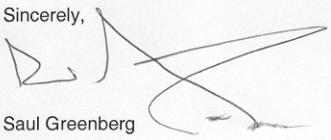
UNIVERSITY OF CALGARY

September 22, 2004

University of Calgary
2500 University Dr. N.W.
Calgary, Alberta
T2N 1N4

I, Saul Greenberg, give Edward Tse permission to use co-authored work from our papers, "SDGToolkit: A Toolkit for Rapidly Prototyping Single Display Groupware" and "Rapidly Prototyping Single Display Groupware through the SDGToolkit" for Chapters 3,4, and 5 of his thesis and to have this work microfilmed.

Sincerely,



Saul Greenberg