An Implementation Architecture to Support Single-Display Groupware

Brad A. Myers

May, 1999
CMU-CS-99-139
CMU-HCII- 99-101

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Abstract

Single-Display Groupware (SDG) applications use a single display shared by multiple people. This kind of interaction has proven very useful for children, who often share a computer for games and educational software, and also for co-located meetings, where multiple people are in the same room discussing, annotating and editing a design or presentation which is shown on a computer screen. We have developed a number of SDG applications that use multiple 3Com PalmPilots and Windows CE devices to emulate a PC's mouse and keyboard. All users can take turns sharing a single cursor to use existing applications like PowerPoint. We have also created other new applications where all users have their *own* independent cursors. This paper describes the architectural additions to the Amulet toolkit that make it easy for programmers to develop applications with multiple input streams from multiple users. Amulet supports shared or independent editing, and shared or independent undo streams. The implementation differs from other Computer-Supported Cooperative Work (CSCW) architectures in that others have one Model and multiple Views and Controllers (one for each user), whereas we have one Model and one View, and multiple Controllers.

This research was partially sponsored by Microsoft, and partially by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Single Display Groupware, Pebbles, Amulet, Toolkit, Computer-Supported Cooperative Work (CSCW), Personal Digital Assistants (PDAs), PalmPilot, Windows CE, Model-View-Controller (MVC).

INTRODUCTION

The Pebbles project is creating applications to connect multiple Personal Digital Assistants (PDAs) to a main computer such as a PC. We are supporting both 3Com PalmPilots and Windows CE devices. We created various PDA-side applications to allow each user to send input from their PDAs to a PC as if they were using the PC's mouse and keyboard. The *Remote Commander* application allows each person to take turns controlling the real cursor, so they can share existing, off-the-shelf applications. *Scribble* allows each person to have an independent cursor, which "floats" above the real applications and can draw directly on the screen, as if on a new layer. All of the cursors are independent of the real cursor, and do not affect any applications. This is useful for pointing and quick, ephemeral annotations. Most relevant to this paper, the *MultiCursor* application allows each person to have an independent cursor inside custom applications which can handle multiple cursors. One such application is "PebblesDraw," which is a shared whiteboard application we built that investigates how multiple people can draw and edit pictures while sharing the same PC display. A previous paper [Myers 1998] describes the overall design and user interface of the Pebbles groupware applications. Other applications are described on the Pebbles web pages: http://www.cs.cmu.edu/~pebbles. The current paper discusses the implementation architecture that makes it all possible, focusing on the issues arising from MultiCursor and PebblesDraw. Note that although we are using PDAs as the input device, the underlying multi-user architecture described in this paper would work no matter what kind of input devices are supplying the parallel streams of input.

The interesting innovation in this architecture is how the multiple streams of input on a single computer are handled independently (so that, for example, there is no interference if one user presses the mouse button and then a different user releases a mouse button). One goal of the project is to allow multiple people on the shared display to use familiar interaction techniques and widgets, even though these interaction techniques were originally designed for use by a single user. We discovered that palettes, selection handles, and menus had to be modified in interesting ways, both in their user interface and in their implementation.

This research is being performed as part of the *Pebbles* and *Amulet* projects. Pebbles stands for: **P**DAs for **E**ntry of **B**oth **B**ytes and **L**ocations from **E**xternal **S**ources. Amulet [Myers 1997] stands for **A**utomatic **M**anufacture of **U**sable and **L**earnable **E**ditors and **T**oolkits, and is a C++ toolkit into which the multi-user architecture has been integrated. The Amulet part of the multi-user architecture runs on X/11, Windows, and the Macintosh, but the part of Amulet that handles PDAs currently only works on Windows.

Amulet needed to be modified in a number of ways to support multiple users. A new slot was added to the interactive behavior objects (called "*Interactor*" objects) and widgets (such as menus and scroll bars) to control which user they belong to, or to specify that the Interactor or widget can be shared in various ways by multiple users. Many of the widgets and commands needed to be "hardened" in various ways to make them more robust for multiple users. The undo facility allows all the commands to go into a single undo history, which allows any user to either undo their own last operation, or anyone's last operation. Alternatively, each user can have a separate undo history over the same set of operations and objects.

Using the "Model-View-Controller" terminology [Krasner 1988], most previous multi-user systems have had a single model (or multiple models with some kind of synchronization mechanism) and multiple View-Controller pairs. For example, this is the design for GroupKit [Roseman 1996]. In contrast, our system has a single Model and a single View, but *multiple* Controllers sharing that one View and Model.

MOTIVATION

Most Computer-Supported Cooperative Work (CSCW) applications deal with multiple people collaborating, each with their own computer. Why would multiple people want to provide input to the *same* computer using separate input devices? The first example is kids collaborating around games and educational software. Background studies have shown that children often argue and fight about who will control a single mouse [Stewart 1998], but when using separate mice, the children exhibited enhanced collaborative behavior. Another study showed that children stay more focused on their tasks when each child has their own mouse and they simultaneously manipulate the same object together [Bricker 1998].

The second example of when multiple people might want separate input devices with a single computer is in certain kinds of meetings, including design reviews, brainstorming sessions, and organization meetings, where a PC is used to display slides or a current plan, and the people in attendance provide input. In small, informal meetings, the users might simply look at a PC's screen. For larger meetings, the PC's screen might be projected on a wall. Many conference and presentation rooms today have built-in facilities for projecting a PC onto a large screen, and various inexpensive technologies are available for rooms that do not. When a PC is used as part of the discussion, often different people will want to take turns controlling the mouse and keyboard. For example, they might want to try out the system under consideration, to investigate different options, or to add their annotations. With standard setups, they will have to awkwardly swap places with the person at the PC. Also, there are times when it will be productive for multiple people to provide input at the *same* time, such as during brainstorming [Stefik 1987][Nunamaker 1991]. Other ideas for applications of single-display groupware are as a demonstration guide, where one user can help another user through an application (just as a driving instructor might have an extra brake or even a steering wheel in a car), and in joint coding and debugging sessions, where one user might be typing in fixes while another user is searching in header files and annotating important features.

We observed that at most meetings and talks, attendees do not bring their laptops, probably because they are awkward and slow to set up, and there is a social stigma against typing during meetings. Today, however, many people are taking notes on their PDAs. For example, the popular PalmPilot is a small PDA from 3Com with a 3¼ inch diagonal LCD display screen which is touch sensitive, and a small input area for printing characters using a special alphabet. There are various vendors of palm-size Windows CE devices, including Casio and HP. PDAs have the advantages that they are small, they turn on instantly, the batteries last for weeks, and notes are taken by writing silently with a stylus. One of the most important reasons that PDAs are so popular is that they connect very easily to a PC (and also to a Macintosh or Unix workstation) for synchronization and downloading. Each PDA is shipped with a cradle and wire that connects to a computer's standard serial port. Software supplied with the PDA will synchronize the data with the PC. It is also easy to load new applications into the PDA. Pebbles takes advantage of this easy connection to a PC. Since people have the PDAs in their hands anyway, we developed a set of applications to explore how these PDAs could be used to allow everyone to provide mouse and keyboard input to the PC without leaving their seats. The architectural issues discussed in this paper would also be useful if multiple regular mice and keyboards were attached to a PC.

RELATED WORK

MMM [Bier 1991] (Multi-Device, Multi-User, Multi-Editor) was one of the first Single Display Groupware (SDG) environments to explore multiple mice on a single display. MMM only supported editing of text and rectangles, and only supported up to three mice. MMM was implemented with two "editors" – one for rectangles and another for editing text. Each editor had to know about multiple users and had to handle each user's state separately. Also, each editor combined the handling of the View and Controller. In Pebbles, the View and Controllers are separated, and neither keeps track of the multiple users' state since instead independent instances of the pre-defined Controller objects are used, and multiple Controllers share the same View objects.

The Xerox Liveboard [Elrod 1992] originally supported multiple cursors operating at the same time, but when produced commercially, it only supported one person with one cursor at a time. The Tivoli system [Pederson 1993] supports up to three people using pens simultaneously on the original version of the LiveBoard. However, the LiveBoard applications do not seem to have been created using a general multi-user architecture as in Amulet.

The term "Single Display Groupware" was coined by Stewart et. al. [Stewart 1998]. Stewart's KidPad [Stewart 1998] is a SDG environment for kids, where multiple mice are connected to a Unix computer. Stewart explicitly decided not to support standard widgets and interaction techniques, and instead uses a "tools" model because it seemed easier for children, and because it avoided many of the issues that needed to be addressed in Amulet.

The M-Pad system [Rekimoto 1998] supports multiple users collaborating with PDAs and a large whiteboard, which is similar to our PebblesDraw, but there does not seem to be underlying architectural support in their toolkit, and they do not deal with conventional widgets.

Most CSCW tools support *multi*-display groupware. Pebbles is most closely related to the form of multi-display groupware called tightly-coupled WYSIWIS (what you see is what I see) systems. However, these systems were generally found to be too limited [Stefik 1987], and most multi-computer systems provide different views for each user, or else use a "relaxed" WYSIWIS style where, for example, the menus and other widgets are *not* shared [Dewan 1991]. Thus, these systems avoid the issues that need to be addressed by Pebbles.

There are many CSCW toolkits for *multi*-display groupware. For example, Rendezvous [Hill 1994] provides for multiple users, each with their own display supported by a single server. The software architecture replicates the View and Controller parts, and uses constraints to keep them synchronized. Groupkit [Roseman 1996] is a multi-user toolkit in tcl/tk which supports a distributed architecture and also uses a multiple View and Controller mechanism. Groupkit is exploring techniques for presenting the pop-up menus and other interactions from users on other computers in a way that will be less disturbing [Gutwin 1998]. The GINA system [Berlage 1993] studied how to distribute command objects to support multi-user undo on multiple machines.

EXAMPLE APPLICATION

Figure 1 shows PebblesDraw, an example application that will be used to explain the single-display groupware features added to Amulet. Each user can pick a particular shape which will be used to identify that user's pointing cursor, selected objects, and text editing cursor. Unlike other systems that assign each user a color (e.g., [Bier 1991][Roseman 1996]), we assign each user a *shape* because in a drawing editor, users can create objects of any color. For example, if the blue user was creating a red circle, it would be confusing. All active users are shown along the bottom of the window, which corresponds to MMM's "home areas" [Bier 1991], but we also show each user's state in their cursor to reduce confusion and eye movements. The cursor shows the current drawing mode, line color and fill color. At the left of the window are the conventional drawing and color palettes. At the right is a button panel that contains the most common commands. The details of the design of PebblesDraw are covered elsewhere [Myers 1998].

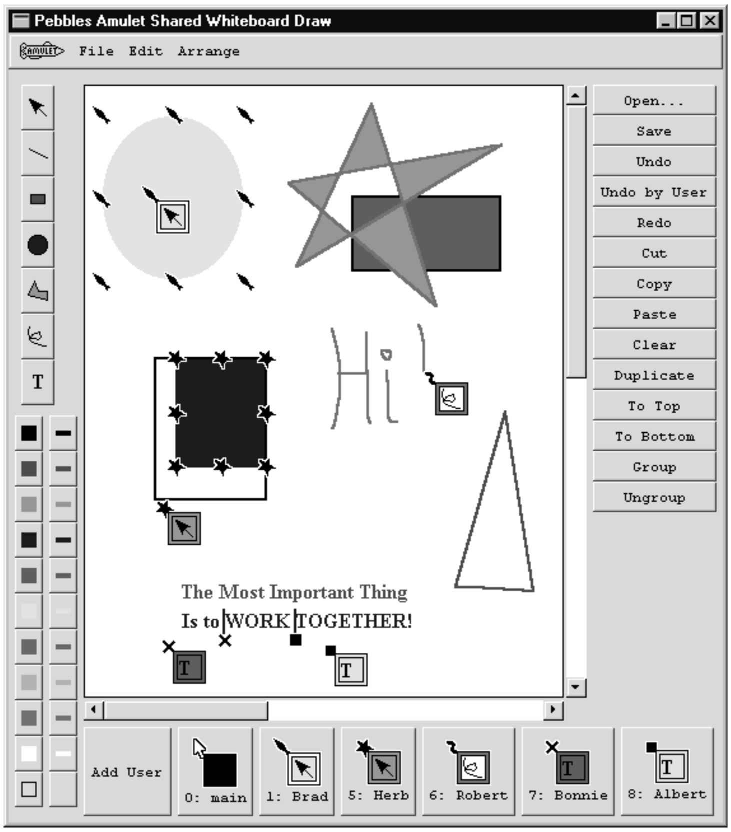

Figure 1. PebblesDraw with a number of people simultaneously editing a drawing. Brad has the yellow oval selected while Herb is growing the blue rectangle. Bonnie and Albert are both editing the text string, while Robert is drawing some freehand letters.

ARCHITECTURE

The general architecture of Pebbles is shown in Figure 2. Various PDA applications run on the PDAs. The user switches among the applications on the PDA as desired. The PDA communicates with the PC using either a serial cable or some wireless technology. Currently, we only support infrared (IR) but we are hoping for a good radio frequency (RF) technology soon. Most relevant for the current paper is when the PDA is running the MultiCursor application.

On the PC end, the serial ports and wireless ports are monitored by the PebblesPC program, which is the general controller. PebblesPC can handle multiple PDAs connected at the same time. PebblesPC loads various "plug-ins" that support the Pebbles capabilities. Each PDA application requests whichever plug-in is desired. Note that there is not a one-to-one correspondence between PDA applications and the plug-ins. Multiple PDAs can share a single instance of the plug-in, although some plug-ins store PDA-specific state to support multiple users simultaneously. Plug-ins are implemented as Windows "dynamic link libraries" (DLLs), which allows new ones to be easily added without recompiling or re-linking PebblesPC or any other plug-in.

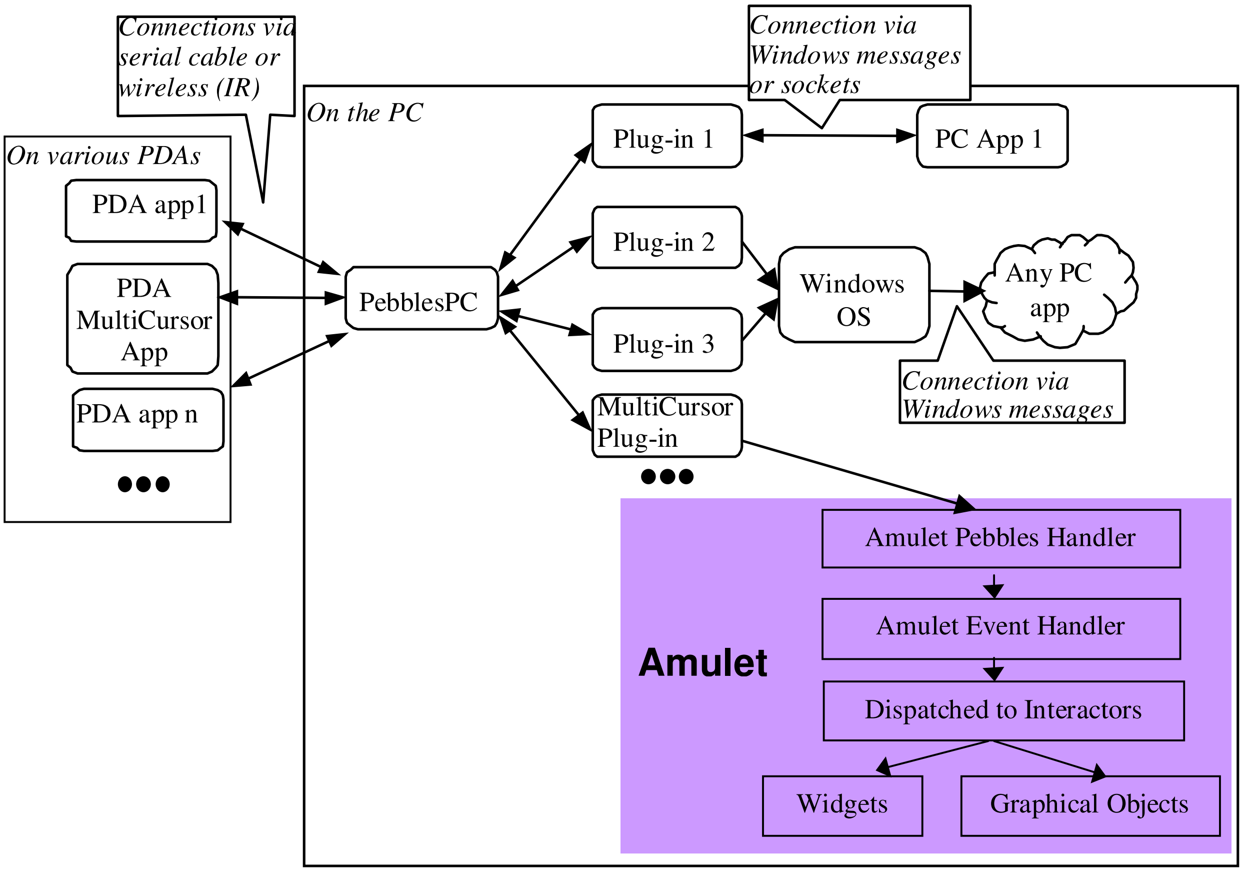

Figure 2. The general Pebbles architecture, including the Amulet connection for MultiCursor, shown in the lower-right.

PebblesPC runs multiple threads—one for each active plug-in, and one for each different user. The threads block waiting for input on the various communication ports.

The messages among the PDAs, PebblesPC and the plug-ins use a standard protocol we designed, and we have created libraries for the PalmPilot, Windows CE and Windows operating systems to support it. This makes creating new Pebbles applications relatively easy. The protocol is simply a byte stream, where each message has a command code byte, a length, and data. Messages are marked with an identifier of the source. Some of the command codes are reserved for the Pebbles protocol. These include ones to announce when a new PDA is connected to the PC, to specify which plug-in is desired, and to announce that the PDA application is exiting. Plug-ins can use any of the command codes that have not been reserved by PebblesPC. Note that different plug-ins can use the same command code for different purposes since PebblesPC first determines which plug-in to run, and then establishes a connection between the plug-in and the PDA. The Pebbles library supplies several functions to help encode and decode Pebbles messages, such as dealing with the different byte orders of 16 bit numbers on the different architectures, and pulling out the different parts of a message. Mostly, however, the specifics of the communication are up to the application.

Plug-ins perform their operation in various ways. One Pebbles application attaches to PowerPoint using OLE Automation. To develop this kind of plug-in clearly requires significant knowledge about OLE and the application being controlled. At the other extreme, the some Pebbles applications just send keys and mouse clicks by inserting the appropriate events into the standard Windows event stream. This plug-in does not need to know anything about the applications that eventually receive the input events. Another option is that the actual work is performed by an application that receives the Pebbles messages either using custom Windows events or through a socket interface. The next sections specifically discuss how the MultiCursor PDA application and PC-side plug-in work.

The MultiCursor Application

On the PDA side, the user can run one of the various Pebbles applications that we created. Figure 3 shows the 3Com PalmPilot running the MultiCursor application. Strokes on the main display area of the PDA control the PC's mouse cursor, and the user can give Graffiti character input to emulate the PC's keyboard input. A feature of MultiCursor not shown in Figure 3 is an on-screen keyboard for entering the special characters such as F1 and ESC. The full designs of the user interface for the various Pebbles applications are described elsewhere [Myers 1998].

Figure 3. The 3Com PalmPilot running the Pebbles MultiCursor application. The input area is used to make strokes to emulate the mouse, and so is mostly empty. The Graffiti area is used to make gestures to emulate the keyboard. The round application buttons are used for the modifier keys like Shift and Control. The two up-down arrow keys in the center bottom are used as the left and right mouse buttons, analogously to the way buttons are handled on lap-tops with a touch pad.

The input is sent to the PC through the serial cable or Infrared. Each event on the PDA causes MultiCursor to send a one byte event type code to the PC, possibly followed by event data. For regular characters, the type is CMD_KEYDOWN and the data is the ASCII value of the character. For all the special characters, the type tells which special key it is, and there is no data. For press and move events, the data is the X and Y of the stylus. A header (`.h`) file is used by both the PC and PDA sides so that the codes are guaranteed to be consistent.

When the byte stream arrives at the PC end, the Pebbles software converts it back into events. The main interesting feature of this conversion is the handling of coordinates from the PDA. After experimentation, we decided that the best way to use the PDA is like a tablet, so relative movements across the surface correspond to corresponding relative movements of the cursor on the screen.

We discovered that the positions reported by the PDA digitizer are very jittery, varying by 1 or 2 pixels in all directions when the stylus is kept still, so the cursor jumped all over the PC's screen. Therefore, we added filtering of the positions. After experimenting with various algorithms and parameters, the best behavior resulted from collecting the last 7 points returned by the PDA, and returning the average as the current point. This removes most of the jitter without adding too much lag. This filtering starts over each time the stylus comes in contact with the PDA, and the array of the last 7 points is initialized with the initial point. This allows points to be provided immediately when the stylus comes in contact with the surface. We also added extra acceleration to the PDA output so one swift movement across the PDA screen would move entirely across the PC's screen. This uses an acceleration algorithm where if the delta position of the cursor movement in a time interval is bigger than a particular value, the delta is multiplied by a bigger number before adding it into the mouse's position.

Sending Events to Amulet

Amulet was designed with a single input queue for all windows. The low-level Amulet Event Handler (see Figure 2) converts the machine-specific window manager event into a machine-independent Amulet event record. The Amulet event record was augmented for Pebbles to contain a User-ID field. Input from the window manager for the regular mouse and keyboard are marked as coming from user zero.

The standard Amulet Event Handler blocks waiting for window manager input. The multiple Amulet Pebbles Handlers (one for each PDA stream) take the input events from the Pebbles Event Constructor and need to dispatch these to the single Amulet Event Handler. We do not want to insert the events into the regular PC event stream, because this would cause the real cursor to move around, and for Amulet we want instead to make sure that the real cursor is only controlled by the real mouse, and use Amulet's custom cursors for all the PDA input. To achieve this, we use the Window Manager's mechanisms to insert special events into the standard event stream. The Amulet Pebbles Handlers therefore construct these special events and insert them into the window manager's event stream. For example, under Windows we use PostMessage to send a message with a Pebbles-defined type-code, and the data pointer is the Amulet event. Each PDA event is marked with the user-id of the serial port number (which can never be 0). The single Amulet Event Handler then accepts these special events along with all the regular Window Manager events, including regular mouse and keyboard input events, and dispatches them in the regular way to the Amulet Interactors.

Identifying the Correct Window

An interesting complication is identifying the window to which the event should be directed. Window managers automatically send the input from the mouse and keyboard to the active window, but the cursor can still move anywhere on the screen. A complication with multiple users sharing a single display is that different users might be working in different (non-modal) windows at the same time. Only one of these windows will be considered the "active" window by the window manager. The same problem occurs with the pop-up windows used to implement menubars and other pop-up and drop-down menus. These windows are not marked as the "active" window by the window manager, but input should still be directed to them.

To solve this problem, Amulet checks mouse events to see which window they should be directed to. The coordinates of the input device are mapped to the screen, and then mapped from the screen to see which window-manager window the coordinates are in. If it is another window for the same application, then the event is marked as coming from that window instead of the active window. This allows the PDAs to control pop-up menus and provide input to different windows even while the real mouse is doing other things.

Modal dialog boxes are still a problem however, since they lock up all the windows of the application. If any user does an operation that causes a modal dialog box to display (like an error message or a file dialog), then all operations in other windows must halt until someone dismisses the modal dialog window. Hopefully, multi-user applications will be designed with very few modal dialogs.

SEPARATING EVENT HANDLING

Interactors and Widgets

The low level event handling described above is completely hidden from programmers using Amulet. Instead, programmers use high-level input handler objects called "Interactors" [Myers 1997]. Each Interactor object type implements a particular kind of interactive behavior, such as moving an object with the mouse, or selecting one of a set of objects. To make a graphical object respond to input, the programmer simply attaches an instance of the appropriate type of Interactor to the graphics. The graphical object itself does not handle input events.

In the "Model-View-Controller" idea from Smalltalk [Krasner 1988], Interactors are the Controller. Most previous systems, including the original Smalltalk implementation, had the View and Controller tightly linked, so that the Controller would have to be re-implemented whenever the View was changed, and vice versa. Indeed, many later systems such as Andrew [Palay 1988] and InterViews [Linton 1989] combined the View and Controller and called both the "View." In contrast, Amulet's Interactors are independent of graphics, and can be reused in many different contexts.

Internally, each Interactor operates similarly. It waits for a particular starting event over a particular object or over any of a set of objects. For example, an Interactor to move one of a set of objects might wait for a left mouse button press over any of those objects. When that event is seen, the Interactor starts running on the particular object clicked on, processing mouse move events, while looking for a stop event such as the left button up event. Each Interactor operates independently, so that multiple Interactors can be waiting for input events at the same time.

All of the widgets in Amulet are implemented internally using Interactors. For example, the menubar at the top of Figure 1 uses a single Choice-Interactor to allow the user to select the menu items.

For Pebbles, the Interactors were augmented with a User-ID field. This field can contain the ID of a particular user or one of two special values. Widgets also have a User-ID field, and just copy the value to the Interactors inside the Widget.

If the User-ID field is a particular user's ID, then this Interactor will only accept input events coming from that user, and will ignore input from all other users. For example, in Figure 1, PebblesDraw creates a cursor icon for each user, and attaches a Move-Grow-Interactor to it. The Move-Grow-Interactor will be set with that user's User-ID to make sure that the icon only follows that user's input. Note that this means that for each graphical object (the "View" and the underlying data structure (the "Model"), there can be *multiple* Interactors ("Controllers"), one for each user.

A special value for the User-ID field is *Am_ANYONE_MIXED_TOGETHER*, which means that everyone can use this widget, even simultaneously. In this case, any user can operate the Interactor and the input events from all users sent to the same Interactor. This might be useful for situations where the designer wants the inputs from all users to be mixed together, possibly for cooperatively controlled objects [Bricker 1998].

The default value for the User-ID field for all Interactors is the special value *Am_ONE_AT_A_TIME.* This means that any user can start the Interactor, but once that Interactor is running, only that same user can provide input to it until the interaction is complete. The standard widgets in Figure 1 are all marked *Am_ONE_AT_A_TIME*, including the menubar, palettes, and scroll bars. For example, if Bonnie starts dragging the indicator of a scroll bar in Figure 1, the Interactor in the scroll bar will be marked for Bonnie, and input from all other users will be ignored. When Bonnie provides the stop event for the Interactor (which is left mouse button up), then the Interactor reverts to waiting for input from any user. This solves the problem reported by other systems where widgets would get confused if one user pressed down, and then a different user pressed down or released the mouse button before the first user was finished.

For example, Figure 4 shows the internal architecture for two parts of Figure 1: the two Text-Edit-Interactors editing the string, and the Interactor in the color palette. Note that since the palette can only be used by one user at a time, it has a single Interactor. Since two users are editing the string at the same time, there are two Interactors affecting the one view. The other users also have Text_Edit_Interactors, but they are currently idle and not affecting any graphical objects.

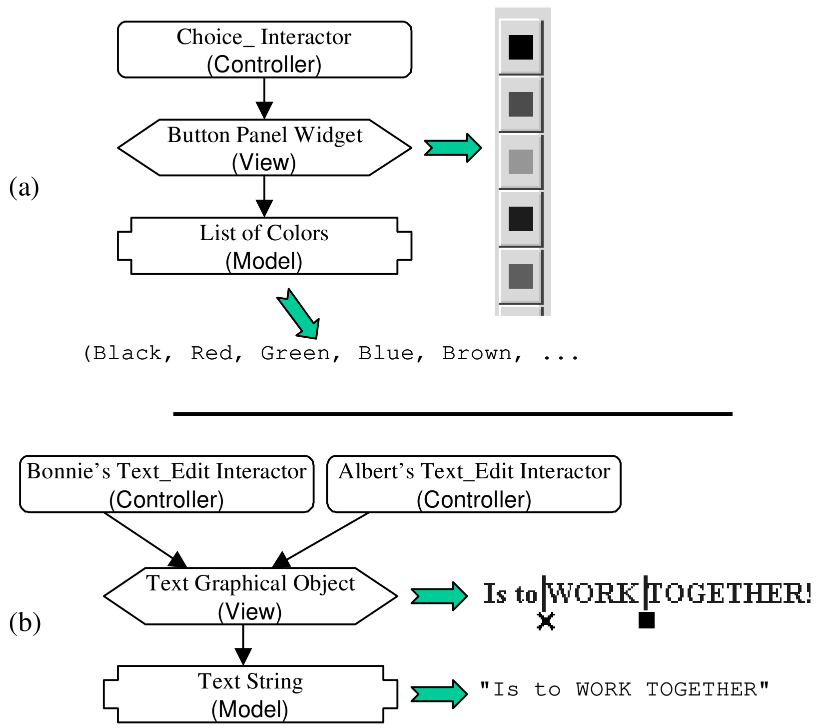

Figure 4. (a) The color panel can only be used by one user at a time, so it has one Text_Edit_Interactor (one controller). (b) Multiple users can edit the text so each user has a separate Text_Edit_Interactor (multiple controllers).

Because all the Interactors operate independently, the various Interactors in an application can be in different states. For example, in Figure 1, Herb has a Move-Grow-Interactor running to change the size of the rectangle, which is waiting for Herb's mouse-button up event to signal completion. This Interactor will ignore the input from all other users. Meanwhile, Albert is using a Text-Editing-Interactor which is processing keys from Albert and waiting for either a

RETURN character or a mouse button down outside the string to stop. Note that all input is treated the same—both from the mouse and keyboard.

Internally, Amulet uses a single process which handles each event sequentially. Each Interactor keeps track of its own state using variables in the Interactor object itself. In effect, each Interactor runs its own independent state machine. Therefore, one Interactor can be running (like Herb's Move-Grow-Interactor) and processing mouse movement events and waiting for a mouse button up event, while another Interactor is waiting for its start event.

When an input event occurs, Amulet checks each of the Interactors in turn to see which one wants the event.[1] If the Interactor wants the event, it processes the event, possibly updating its internal state and various graphical objects, and then returns control to the main loop. If the Interactor does not want the event, then the main loop checks the other Interactors. If none wants the event, it is discarded. Thus, when an input event comes in marked with a particular User-ID, that event will be given only to those Interactors with an appropriate value in their User-ID field.

In designing an application, the programmer can decide what level of cooperation and parallelism is desired. If a widget or object should be operated by only a single user at a time, then it can have a single Interactor using *Am_ONE_AT_A_TIME*. On the other hand, if multiple users should be able manipulate objects at the same time, then each user might have their own separate Interactors marked with that user's ID. To enable maximal parallelism, the PebblesDraw application allocates a set of Interactors for each user so each user can create and edit graphical objects at the same time.

Text Editing

The default behavior for text editing would be for only a single user to be able to edit a text string at a time. However, as shown in Figure 4, we wanted to explore multiple users editing the same string at the same time. This raises similar issues to multi-screen multi-user text editors, such as SASSE [Baecker 1993].

The original Amulet single-user text object had a built-in ability to show a cursor. All of the text editing operations, such as inserting a character and deleting the previous word, operate with respect to this cursor. Some of these operations are fairly complex because, for example, they handle various encodings of Japanese multi-byte character embedded in a single-byte string. To extend this to the multi-user case, we added a set of cursor positions, indexed by User-ID. Before each incremental text inserting or editing operation, the Text-Interactor sets the internal "main" cursor with the appropriate user's cursor position, performs the insert or

---

[1] There are many options and optimizations that make this mechanism much more flexible and efficient, including multiple priorities for Interactors, separating the handling of independent windows, etc. [Myers 1997].

edit, reads out the new cursor position, stores the new position with the user's ID, and then sets the internal cursor to be off. This enables the system to use all the original editing functions without change, while still supporting multiple users.

One complication is that all the cursors' positions may need to be updated whenever any user performs an edit. For example, if Bonnie deletes some characters in Figure 1, Albert's cursor should still be before the "T." Therefore, an extra step is needed at the end of the loop described above to update the other cursors if necessary.

To show the different user's cursors on the screen, separate graphical objects are used for each user's text cursor. In PebblesDraw, a vertical line with the user's shape at the bottom is used. The list of cursor objects and their associated positions is associated with the text object so the various text Interactors can update them.

Another tricky issue with text editing is dealing with undo, which is discussed in the "Undo" section, below.

Selection Handles

The Amulet toolkit provides a selection handles widget to select, move, and grow graphical objects. All other toolkits require that each application re-implement this standard behavior. To support single-display groupware, the selection handles widget was augmented with a User-ID field, and the ability to show any shape as the handle, instead of just using squares.

In PebblesDraw, a separate selection handles widget is created for each user, and set with that user's User-ID.[2] This allows each user's actions to be independent, as shown in Figure 1. The cursor shapes are designed so users can always see that the object is multiply selected, although it can be difficult to tell by which users. The operations do reasonable things if two people manipulate the same object at the same time. For example, if one user deletes an object while another is growing it, then the grow will abort. If two people try to grow the same object at the same time from opposite corners, then each user will have a separate interim feedback rectangle that shows the current size as that user independently moves his or her corner, and as each user releases the button, the corner will snap to the final position.

In the future, we might want to disallow multiple people from selecting the same object at the same time if this proves too confusing. Alternatively, we might make is easier to see which users have the object selected. For example, since there happen to be eight handles around an object and PebblesDraw currently supports up to eight shapes, an obvious idea is to divide the handle positions among all the users who have this object selected. However, this might con-

---

2 Note that a single selection handles widget will allow multiple objects to be selected in the usual way, by using shift-click or dragging in background. The use of *multiple* selection handles widgets allows there to be independent sets of selected objects.

fuse users into thinking that they can only change the object's size from the handles that have their shape. Further studies of these issues are planned.

Command Objects

Rather than using a "call-back procedure" as in other toolkits, Amulet allocates a *command object* and calls its "Do" method [Myers 1996]. Amulet's commands also provide slots and methods to handle undo, selective undo and repeat, and enabling and disabling the command (graying it out). Command objects promote re-use because commands for such high-level behaviors as move-object, create-object, change-property, become-selected, cut, copy, paste, duplicate, quit, to-top and bottom, group and ungroup, undo and redo, and drag-and-drop are supplied in a library and can often be used by applications *without change.*

For Pebbles, Amulet's command objects were augmented to support multiple users. When a command is about to be executed, it is set with the User-ID of the user who invoked that command. For a single-user application, there is a single selection handles widget, which the commands such as Cut and Change-Color use to determine which objects should be affected. In the multi-user case, there might be *multiple* selection handle widgets. Therefore, the built-in command objects were augmented to accept a *list* of selection handle widgets, in which case the command object will look for the particular selection handle widget whose User-ID matches the User-ID set into the command. Then, the list of selected objects is retrieved from that selection handles widget. For example, in Figure 1, if Brad does a cut, only the yellow oval will be affected.

An interesting issue arises about the graying out of illegal items. Since only one user at a time can use the drop-down menus, it makes sense for the items in those menus to gray out as appropriate for that user. For example, since in Figure 1, Bonnie has nothing selected, if she uses the drop-down menus, the commands that require a selection, such as Cut, would be grayed out. If Brad used the drop-down menus, then Cut would not be grayed out.

However, the button panel of commands (at the right of Figure 1) is always visible. Therefore, it does *not* work for items to be grayed out in the button panel, because some commands will be valid for one user but invalid for another user. Therefore, we had to modify all the operations to make sure that they did something reasonable, like beep or display an error message, if they were invoked when they were not valid for the current user. The previous implementation of these commands in Amulet assumed that since they would be grayed out, they could never be invoked when not valid. As a further enhancement, the widget can be set to gray out the items immediately after the user pressed the mouse button in the widget. Thus, when Bonnie presses in the button panel, the items for cut, copy, etc. will gray out at that point, and return to black when Bonnie releases the mouse button. This will prevent illegal actions from being executed, as in the conventional menu design.

Palettes

The palettes in PebblesDraw (for the current drawing tool and current colors) are implemented as button panels. As such, they automatically get the *Am_ONE_AT_A_TIME* behavior. The interesting problem is that a palette cannot display the currently selected value in the palette itself as in all single user applications. If one user is drawing a red circle at the same time that another is typing blue text, how would that be shown? Most CSCW applications are for multiple machines and assume that each user can see their own private copy of the palettes on their own separate displays, so this is not an issue. The palettes in MMM [Bier 1991] did not show any state and showed each user's current modes only in the home areas. The Tivoli project [Pederson 1993] mentioned this problem with palettes, but apparently provided no feedback as to the users' modes. To solve this problem in PebblesDraw, the button panels for the palettes are marked so they do not show any final feedback as to where the user selects (although they still show interim feedback as the user is making a selection). Instead, the user's selected mode is copied into a per-user data structure and shown in the user's cursor that follows the mouse, as well as in the user's home area. Amulet's built-in Change-Property command was modified to accept a list of current values indexed by User-ID. Similarly, the Create-Object command uses the per-user data structure to get the values to use for the new object, so each user can have an independent mode.

UNDO

Amulet provides built-in support for undo. In addition to the conventional multi-level undo that can undo all the previous operations back to the beginning (like Microsoft Word version 6 and later), Amulet also supports a *selective undo* mechanism. Any previous command, including scrolling and selection operations, can be selectively undone, repeated on the *same* object, or repeated on a new selection [Myers 1996]. The Selective-Undo method has an associated method which checks to see whether the command can still be executed. For example, if an operation changes the color of an item, the Selective-Undo-Allowed method will check to make sure that the object is still visible. If not, then the Selective-Undo command in the menu will be grayed out.

Normally, all users will share a single undo history. This is the design used in PebblesDraw. The undo dialog box for Amulet was augmented to annotate each command with the shape for the user who executed it (see Figure 5). The normal Undo command undoes the last executed command no matter who executed it. Similarly for Redo.

We also added a new Undo-by-User command which undoes the last command of the user who executes this undo command. Undo-by-User searches back through the history looking for a command that was performed by the current user. If a command is found, its Selective-Undo-Allowed method is checked to make sure that the command can be undone. If so, then the command is Selectively Undone using the standard mechanism. For example, in Figure 5, if Herb performs undo-by-user, it will skip over Bonnie's commands and undo the Move of a rectangle (command # 34). If that rectangle had been deleted by a different user, then Herb's attempt to do undo-by-user would just beep, since his last command could not be undone.

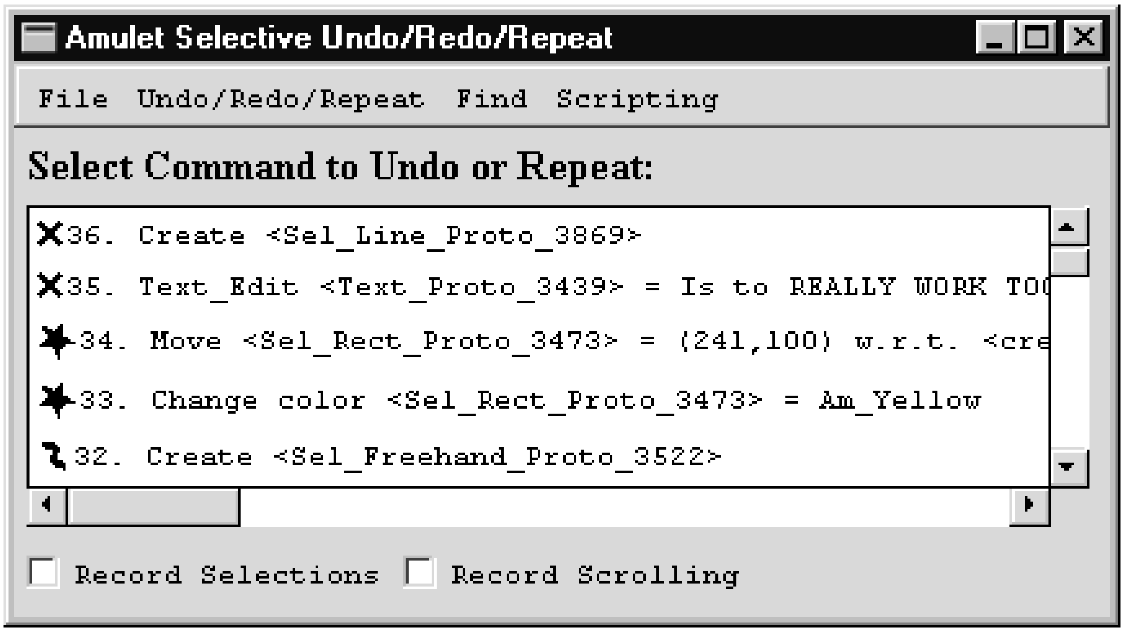

Figure 5. Undo dialog box [Myers 1996] for PebblesDraw where each command is marked with the shape for the user who performed it.

Unlike regular undo which pops items off the undo history, Selective Undo always *adds* the inverse of the command to the top of the history [Myers 1996]. For example, if Herb performs Undo-by-User, it will add a new command to the top of the history (as number 37) that will be labeled "Undo Move." The action of this command will be to move the object back where it was before command 34 was executed. An interesting consequence of this design is that if Herb does another Undo-by-User, it will add an *additional* command to the top of the stack that will undo the previous undo, and therefore move the object back where it was. Thus, Undo-by-User keeps toggling the effect of one command rather than undoing a series of command like regular undo. If this proves to be a problem, we could have an undo mode, as in the Emacs editor, where each subsequent Undo-by-User would move back in the history and undo a previous command, until the user signalled the end of undo-mode. This design for Undo is similar to that used in GINA [Berlage 1993].

Independent Undo Histories

A different design gives each user an independent undo history. Since the command objects are each marked with the appropriate user, it is easy to find which undo history to attach each command to. The complication of having independent histories is that for the single-user case, the undo methods could assume that whenever Undo was executed, the state of the system was always the same as just after the operation was performed. However, in a multi-user situation, if each user has their own undo history, then one user might modify an object such that a different user's undo will no longer be valid. For example, one user might change the color of an object, then a different user might delete the object, and the first user then could try to undo the change color.

Even though the Undo method is different from the Selective Undo method for performing the operation [Myers 1996], it turns out that the regular undo method can be performed whenever the selective undo method can be performed. Therefore, the Selective-Undo-*Allowed* method that checks to make sure the Selective Undo is possible, can be reused to check whether the regular Undo is possible in the current state. We just add a check in the top-level undo menu item to see if the selective-undo-allowed method returns true before executing the regular Undo method.

Note that the undo operation will often work even if other users have manipulated an object, because all commands save their old values in the command itself. For example, if a rectangle is white and one user changes it to be red, and then a second user changes it to be yellow, the first user's undo will still be valid, and will change the rectangle back to being white (the color before the first user performed the operation).

Undoing Text Edits

The text strings in PebblesDraw are short labels. Like other drawing packages, editing of the text label starts with clicking in the label, and ends with clicking somewhere else. The unit for undoing is therefore the complete edit from the start to the finish. Thus, if a string starts out as "one" and is edited to be "two" the undo will restore it to be "one" no matter what the operations performed to edit it. This is in contrast to text editors like Microsoft Word where there is no obvious start and end to an edit session and the editor uses heuristics to decide what is the unit for undoing.

Supporting multiple users adds a significant complication to this simple undo model. If a string starts out as "one" and the first user edits it to be "one two" but the second user then starts editing and makes it be "one two three", what should the string be if the first user then call for undo? Currently, since the unit for undo is the full label, the system undoes the string back to its state before the user started editing it. Therefore, in this case the string would become "one", thereby losing the second user's edits.

We have a design for a more sophisticated multi-user undo facility for text, but it requires much more mechanism which is probably not necessary for short labels. The new mechanism keeps multiple *marks* in the text, showing the location of each edit in the history. Then, each independent text edit operation in the history would refer to the specific marks in the string that are associated with the edit. If the marks are still available in the text, then the undo can be performed. If the marks are no longer valid, which might happen if that section of the text was deleted, then the undo is no longer available. This mechanism would allow the edits for different users on the same string be independent.

STATUS AND FUTURE WORK

The implementation of multi-user support in Amulet is complete, as described above. We have implemented PebblesDraw and a few other test applications using it. We are now planning user tests on the various options for the user interface to see what is most effective for users.

We believe in distributing the results of our research, to help collect useful feedback and aid in technology transfer. You can download Pebbles, along with PebblesDraw, MultiCursor and many other applications from the Pebbles web site: http://www.cs.cmu.edu/~pebbles. As a whole, the Pebbles applications have been downloaded about 12,000 times, although we have no way to tell how many people tried MultiCursor or PebblesDraw. The Amulet toolkit has been downloaded over 10,000 times in the past year, and is available from http://www.cs.cmu.edu/~amulet. However, the features to support multiple users described in this paper are not yet included in the released version of Amulet.

For the future, we want to explore having a more sophisticated program running on the PDA. For example, instead of having the users' current modes shown in the cursor as in PebblesDraw, they might be shown on each person's PDA screen. An interesting research question is then how to augment the communication path to support the high-level *semantic* input from the PDAs. For example, we might include facilities like M-Pad that uses palettes on the PDA to set parameters of objects on the large screen [Rekimoto 1998].

Of course, we want to explore using this architecture to implement many new applications. The CSCW literature contains a number of interesting programs designed for multiple computers, such as "Electronic Brainstorming" and "Structured Idea Generation Process" from Univ. of Arizona [Nunamaker 1991] and Xerox PARC's Cognoter [Stefik 1987]. We want to see which of these will be effective if used with PDAs and a single PC display implemented using the architecture described here.

CONCLUSIONS

The Amulet toolkit was augmented with multiple user support for single display groupware. This highlighted a number of interesting research issues both in the user interface of applications and in the architecture needed to support them. New widgets and interaction techniques were needed so that multiple users can share the same set of widgets at the same time. The Interactor behavior objects and widgets were augmented with an additional parameter so they could be reserved for a single user, used by any user but one at a time, or used by multiple users simultaneously. Many of the commands in Amulet had to be “hardened” so they could be called even when they would normally be grayed out for a single user. The result is that Amulet now supports having one Model and one View with multiple Controllers, which is a different design than previous CSCW toolkits. We believe that multiple users sharing a single display can be an effective way to collaborate for a number of different applications, and having an easy-to-use architecture to explore it will make this kind of software significantly easier to build.

ACKNOWLEDGMENTS

For help with this paper, we would like to thank Rob Miller, Jason Stewart, and Lauren Bricker.

REFERENCES

[Baecker 1993] Ronald M. Baecker, Dimitrios Nastos, Ilona R. Posner and Kelly Mawby. “The User-centered Iterative Design of Collaborative Writing Software,” Human Factors in Computing Systems, Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 399-405.

[Berlage 1993] Thomas Berlage and Andreas Genau. “A Framework for Shared Applications with a Replicated Architecture,” ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93. Atlanta, GA, Nov, 1993. pp. 249-257.

[Bier 1991] Eric A. Bier and Steve Freeman. “MMM: A User Interface Architecture for Shared Editors on a Single Screen,” ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91. Hilton Head, SC, Nov, 1991. pp. 79-86.

[Bricker 1998] Lauren Bricker. Cooperatively Controlled Objects in Support of Collaboration. Seattle, WA, Department of Computer Science and Engineering, University of Washington. 1998. PhD Thesis.

[Dewan 1991] Prasun Dewan and Rajiv Choudhary. “Flexible User Interface Coupling in a Collaborative System,” Human Factors in Computing Systems, Proceedings SIGCHI'91. N.O., LA, Apr, 1991. pp. 41-48.

[Elrod 1992] Scott Elrod, et. al. “LiveBoard: A Large Interactive Display Supporting Group Meetings, Presentations and Remote Collaboration,” Human Factors in Computing Systems, Proceedings SIGCHI'92. Monterey, CA, May, 1992, 1992. pp. 599-607.

[Gutwin 1998] Carl Gutwin and Saul Greenberg. “Design for Individuals, Design for Groups: Tradeoffs between Power and Workspace Awareness,” Submitted for Publication, 1998.

[Hill 1994] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson and Wayne Wilner. “The Rendezvous Architecture and Language for Constructing Multi-user Applications,” ACM Transactions on Computer-Human Interaction. 1994. 1(2). pp. 81-125.

[Krasner 1988] Glenn E. Krasner and Stephen T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system," Journal of Object Oriented Programming. Journal of Object Oriented Programming. 1988. 1(3). pp. 26-49.

[Linton 1989] Mark A. Linton, John M. Vlissides and Paul R. Calder. "Composing user interfaces with InterViews," IEEE Computer. IEEE Computer. 1989. 22(2). pp. 8-22.

[Myers 1996] Brad A Myers and David Kosbie. "Reusable Hierarchical Command Objects," Proceedings CHI'96: Human Factors in Computing Systems, Vancouver, BC, Canada, April 14-18, 1996. pp. 260-267.

[Myers 1997] Brad A. Myers, et. al. "The Amulet Environment: New Models for Effective User Interface Software Development," IEEE Transactions on Software Engineering. 1997. 23(6). pp. 347-365. June.

[Myers 1998] Brad A. Myers, Herb Stiel and Robert Gargiulo. "Collaboration Using Multiple PDAs Connected to a PC," Proceedings CSCW'98: ACM Conference on Computer-Supported Cooperative Work, Seattle, WA, November 14-18, 1998. pp. 285-294.

[Nunamaker 1991] J. A. Nunamaker, Alan R. Dennis, Joseph S. Valacich, Douglas R. Vogel and Joey F. George. "Electronic Meeting Systems to Support Group Work," CACM. 1991. 34(7). pp. 40-61. July.

[Palay 1988] Andrew J. Palay, et. al. "The Andrew Toolkit - An Overview," Proceedings Winter Usenix Technical Conference, Dallas, Tex, Feb, 1988. pp. 9-21.

[Pederson 1993] Elin Pederson, Kim McCall, Thomas P. Moran and Frank G. Halasz. "Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings," Human Factors in Computing Systems, Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 391-398.

[Rekimoto 1998] Jun Rekimoto. "A Multiple Device Approach for Supporting Whiteboard-based Interactions," Human Factors in Computing Systems, Proceedings SIGCHI'98. Los Angeles, CA, Apr, 1998. pp. 344-351.

[Roseman 1996] M. Roseman and S. Greenberg. "Building Real Time Groupware with GroupKit, A Groupware Toolkit," ACM Transactions on Computer Human Interaction. 1996. 3(1). pp. 66-106.

[Stefik 1987] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning and Lucy Suchman. "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," Communications of the ACM. 1987. 30(1). pp. 32-47.

[Stewart 1998] Jason Stewart, Elaine M. Raybourn, Ben Bederson and Allison Druin. "When Two Hands Are Better Than One: Enhancing Collaboration Using Single Display Groupware," Human Factors in Computer Systems, Adjunct Proceedings of SIGCHI'98. Los Angeles, CA, Apr, 1998. pp. 287-288.