MMM: A User Interface Architecture for Shared Editors on a Single Screen

Eric A. Bier† and Steve Freeman‡

†Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, bier.parc@Xerox.com
‡Cambridge University, Cambridge, England CB3 9EU, freeman@europarc.xerox.com

Abstract

There is a growing interest in software applications that allow several users to simultaneously interact with computer applications either in the same room or at a distance. Much early work focused on sharing existing single-user applications across a network. The Multi-Device Multi-User Multi-Editor (MMM) project is developing a user interface and software architecture to support a new generation of editors specifically designed to be used by groups, including groups who share a single screen. Each user has his or her own modes, style settings, insertion points, and feedback. Screen space is conserved by reducing the size and number of on-screen tools. The editors use per-user data structures to respond to multi-user input.

KEYWORDS: conference-aware editors, single-screen collaboration, per-user customization, home areas

1 Introduction

We are interested in the software architecture and user interface needed to build multi-user editors that are convenient to use. While most previous multi-user editors assume that each user has his or her own networked workstation, we restrict ourselves to editors shared on a *single* workstation with multiple pointing devices. This restriction allows us to set aside the problems of coordinating multiple workstations and to focus on user interface and internal architecture problems. Also, our approach directly supports scenarios of use such as:

(1) Users may wish to collaborate on the same workstation screen. For example, two programmers working together at a workstation may both desire input devices.

(2) Several people can share a computer with a blackboard-sized display by directly writing on the display with a stylus or by controlling it from hand-held computers. In addition, people can share a hand-held computer by passing it back and forth.

(3) Even when multiple workstations are available, conference-aware editors like ours will enhance the ability of participants to contribute simultaneously.

(4) Future software applications may provide an "assistant" -- a program that acts like another user with which we collaborate on workstation tasks. A conference-aware architecture will make such assistants easy to integrate.

(5) Our architecture supports a single user with pointing devices in both hands. Two-handed interfaces have been motivated by Buxton and Myers [Buxton86].

Because we expect that large-screen and portable computers will use a stylus as their chief input device, we focus on user interfaces that do not require a keyboard.

We have built a set of toy editors that support simultaneous real-time collaboration. They allow fine-grained sharing, often allowing simultaneous access to the same text string or graphical object. A mouse can be registered with a user and will work with that user's defaults and preferences (such as insertion points, modes, selection colors and mouse parameters) until the device is registered with another user. Registration is fast enough that users can pass devices back and forth during a session. Users can alternately collaborate tightly and work separately with little interference.

We address four user interface problems:

Registration. How can an input device be quickly registered with a user?

Real Estate. How should the screen be managed so that collaboration is practical in limited space?

Per-User Feedback. How can the system direct feedback to the right user without disturbing others?

Interference. How can users engage in separate tasks without interfering with each other?

We also address three software architecture problems:

Input Handling. How should input events from multiple devices and/or multiple users be queued and combined to allow fine-grained sharing?

Replication. Which data structures of a document editor need to be replicated per user, to support per-user modes, selections, and preferences?

Screen Update. What screen update algorithms will produce an image consistent with the editing activities of all users?

Section 2 describes related work. Section 3 describes our user interface and how it addresses our four user interface problems. Section 4 describes our architecture and how it addresses our three architectural problems. Section 5 presents our early experiences and conclusions.

Our prototype editors run in a framework called the Multi-Device Multi-User Multi-Editor (MMM), which is implemented in the Cedar Mesa programming language and runs in the Portable Cedar programming environment [Swinehart86] on SUN Microsystems SPARCstations and other computers. We have augmented our workstations with one or two serial mice to control additional cursors.

2 Related Work

In recent years, many projects have built computer-based tools for supporting real-time collaboration [Kraemer88]. These tools include shared window systems, collaborative learning environments, tools to support meetings, and shared editors. Because MMM is intended to support a broad range of collaborative work situations and because it includes both shared editing and shared window system functionality, it builds on all of these system classes. MMM also builds on single-user tools such as window systems.

Shared window systems [Lantz86, Lauwers90, Crowley90, Ensor88] allow users to share applications over a network. Work on shared window systems has focused on sharing unmodified single-user applications, maintaining a networked conference, and floor control. Like shared window systems, our system supports multiple shared applications; however we focus on conference-aware applications.

MMM shares concerns with single-user window systems. As in Trestle [Nelson91], MMM synchronizes several parallel processes to consistently interpret input and update the screen; however, MMM's locking is coarser and simpler than Trestle's so it is easier to create correctly synchronized multi-user applications. As in X Windows [Scheifler86], MMM's screen is made of a hierarchy of applications; however, MMM's applications need not be rectangular and can be repositioned in parallel.

SharedARK supports collaborative physics learning, allowing users to perform simulated physics experiments [Smith89]. Each participant controls a pointer, labeled with the participant's name, that is visible on all conference screens. Button objects can be moved or pressed by any user. Participants directly manipulate a simulated world. Like SharedARK, MMM supports per-user pointers, shared buttons, and direct manipulation.

The Colab project studied collaborative tools in the context of single-room meetings [Bobrow90, Stefik87a, Stefik87b]. Participants, working at individual workstations, control a large shared screen. The Colab work suggests that simultaneous access and user interface simplicity are important in multi-user tools. From their work on the Cognoter idea organizing tool, they felt that shared access makes brainstorming more productive because users need not wait for others to complete an entry before making their own contribution. They also note that shared user interfaces need to be simple, because there is little time for training in a group setting.

Recent shared editors, such as Grove and ShrEdit, allow users to simultaneously edit a text document [Ellis90, Olson90b]. Commune allows users to simultaneously draw on the pages of a shared electronic notebook with a stylus [Minneman91]; color is used to distinguish which cursor and which drawings belong to each user.

Olson *et. al* recently surveyed some of the design issues for group editors [Olson90a]. With respect to their taxonomy, our work relates to previous work in these ways: We ignore the problems of network delays, synchronization, turn taking, telepointing, and private views, while we focus on the problems of simultaneous use, identifying conference participants and their editing locations, and reducing the need for locking. In addition, we focus on supporting per-user tailoring, the sharing of a single screen, and multi-level editing in a hierarchy of editors.

3 MMM's User Interface

MMM's user interface consists of three visible components: *Home areas* provide iconic representations of users. *Editors* allow users to view and modify document media. *Menus* provide buttons that users can press to invoke commands. This section describes how we use these components to solve our four user interface problems.

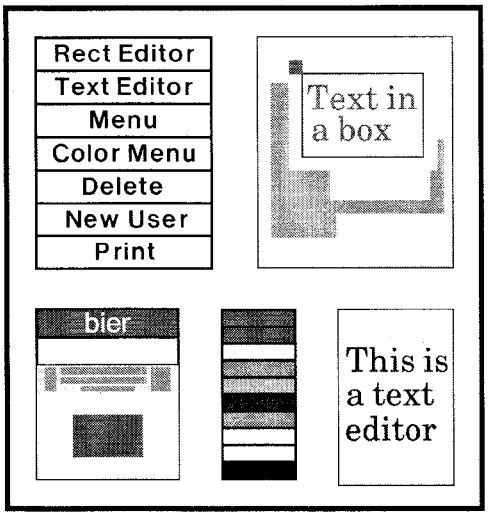

Figure 1. A typical MMM screen

Figure 1 shows a typical MMM display with (clockwise from upper left) a command menu, a rectangle editor (with nested text editor), a text editor, a color menu, and a home area.

3.1 Home Areas

MMM displays a home area for each user participating in a session. To join a session, a new user types his or her name and clicks the New User button. MMM creates a home area that displays the selected name.

To begin using MMM, a user chooses a mouse and clicks on the name bar in a home area, as shown in figure 2. As a result, MMM assigns that mouse to that user, and any actions performed with this mouse will take his or her preferences into account. The change of ownership causes the cursor color to change to that of the home area, as shown in figure 2(b). A user may have more than one home area. Extra home areas allow users to switch back and forth between different sets of preferences. Each home area is said to belong to a different *user instance*.

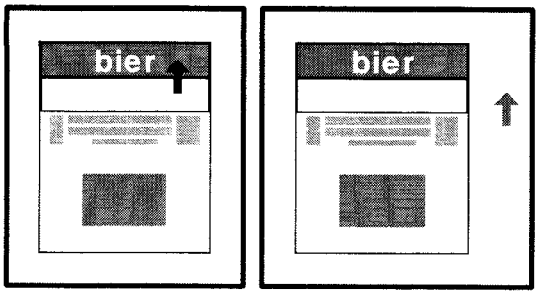

Figure 2. Clicking on a home area to register a mouse

3.2 Our Two Editors

To date, we have implemented two editors, for rectangles and text, that are both functionally very simple. The *rectangle editor* permits users to create solid-colored rectangles, select and delete groups of them, and change their size, position and color. The *text editor* allows users to place a starting point for a line of text, enter characters, choose a font and color, move a line of text, select a sequence of characters, and delete or change the color of selected characters.

3.3 Spatial Command Choice

Because we are interested in keyboardless interfaces, all editor commands, except text entry, are activated by a pointing device. For many MMM operations, command selection involves pointing to different parts of graphical objects to invoke different operations.

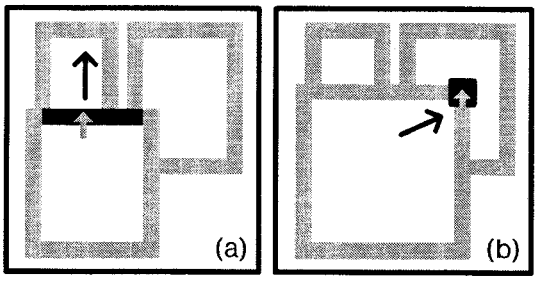

Figure 3. Spatial command choice in the rectangle editor. (a) Dragging a rectangle by its edge. (b) resizing a rectangle by its corner. (The black arrows indicate cursor motion; they do not appear on the screen.)

In particular, objects in the rectangle editor are surrounded by a wide border, a *frame*. The user drags an *edge* of a frame to move (figure 3(a)), and a *corner* to resize (figure 3(b)) the rectangle. In each case, the part of the frame being dragged is highlighted. Users create a new rectangle by clicking on the editor background.

Spatial command choice has many advantages in a multi-user context. It reduces the use of persistent modes. It is easy to see; novices can learn from experienced users by watching them, and collaborators can see what others are doing. Finally, because the cursor need not travel to an off-screen menu, the command motion is unlikely to be misinterpreted as a conversational gesture (see the description of Sketchtool in [Bobrow90]).

3.4 Per-User Commands, Modes, and Preferences

Several users can work simultaneously in an editor, performing different operations and using different modes. In the rectangle editor, for example, one user may drag one rectangle while someone else resizes another. The editor keeps track of both operations and updates the screen to show their progress. Similarly, one user may choose blue as the current color for that editor while another chooses red. The editor stores both users' modes and creates a rectangle of a color appropriate to the creating user.

3.5 Fine-Grained Editor Sharing

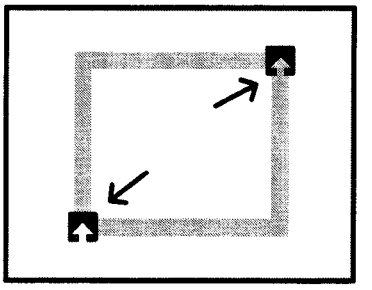

Figure 4. Two users stretching a rectangle at the same time.

Users can work simultaneously on the same object; one user can add characters to a text string while another changes the color of existing characters. Very fine-grained sharing is also possible; one user can stretch a rectangle by one corner while another stretches its opposite corner, as shown in figure 4, or one user can type a string of characters while another repositions it.

Fine-grained sharing may not often be necessary, but it allows a user to make an edit with confidence that other users will not be locked out, reducing interference among users.

3.6 Editors as Window Systems

MMM has no conventional window manager. Instead, the desktop is an instance of our rectangle editor. The rectangle editor allows other editors to be nested inside of it. Each editor presents a finite window onto an "infinite" plane that contains the objects and child editors that that editor manages, which may overlap. The shape and position of this window is managed by the editor's parent. Editors may be partly or entirely hidden. Figure 5 shows several rectangle editors and a text editor. Note that the text editor C is partly hidden by sibling editor B (and by a rectangle), and that rectangle editor D is partly outside the porthole of its parent editor B.

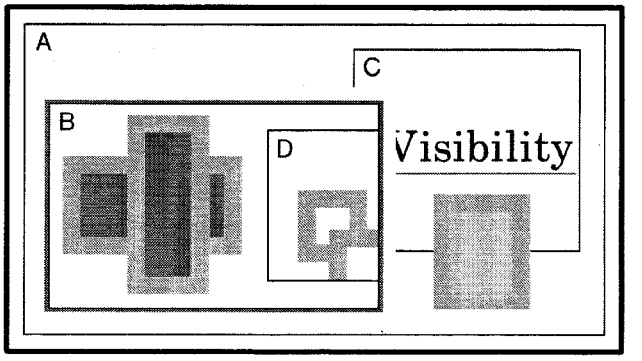

Figure 5. The visible parts of four editors. A, B, and D are rectangle editors. C is a text editor. B's highlighted border indicates that a user has selected it.

Because the window system *is* an editor, users do not need to

learn both window manager and editor operations, unlike systems where selection *of* an editor is different from selection *within* an editor. Users can also place shared widgets in the document in which they are working; widgets need not remain at the top level.

3.7 Per-User Feedback

In single-user environments, graphical feedback is used to display aspects of system state, such as the current application, mode, color, or insertion point. In our single-screen multi-user system, graphical feedback must show all of this information *and* indicate to which user the feedback applies. We use color and spatial placement to indicate this correspondence.

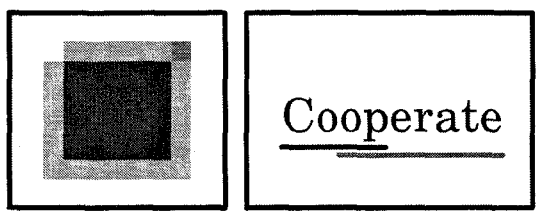

Figure 6. Selection feedback for two users. (a) A doubly-selected rectangle. (b) Doubly-selected text characters.

Each user has a color to identify his or her cursors, insertion points, and selections. In figure 6(a) two users have selected a rectangle; each user's selection is shown by highlighting a corner of the rectangle frame in that user's color. Similarly, figure 6(b) shows two selections in a text string, each marked by an underline in one user's color; the overlapping part of the selection is underlined twice. Because color feedback is less informative on black-and-white displays or with color-blind users, we plan to augment this feedback in the future.

Likewise, a narrow band in the frame around a user's current editor is set to that user's color. Where several people are using the same editor, a band is colored for each user. If too many users have selected an object (rectangle, text or editor) to identify them all in the frame, then only the most recent selections are shown.

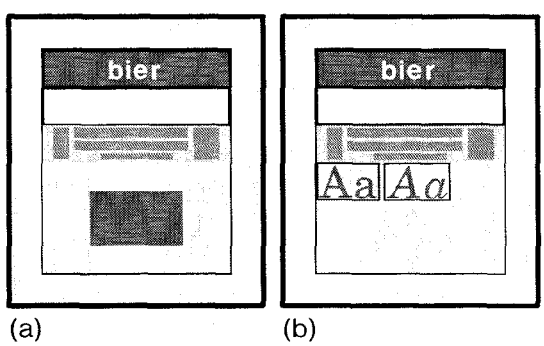

Figure 7. Mode feedback in the home area. (a) Rectangle creation color feedback. (b) Current font feedback.

A user's mode within his or her current editor is displayed in the lower half of that user's home area. When a user works in a rectangle editor, his or her home area displays that user's default color in that editor (figure 7(a)). Similarly, for a text editor, the current font and colors are shown (figure 7(b)).

3.8 Shared Menus

In many desktop environments, menus are displayed once for each application window or at a unique location on the screen (e.g., the Apple Macintosh pull-down menus are at the top of the screen). For a shared application on a single screen, however, menus displayed once per window take up much space and menus displayed at a fixed location only allow a single application to be used at once.

Instead, our menus can be shared among users and editors, and positioned anywhere on the screen, even in documents. For example, the menus in figure 1 can be placed in a rectangle editor regardless of its nesting level, and then applied to objects in any user's selected editor, regardless of that editor's nesting level.

Allowing people and editors to share menus reduces the screen real estate needed for control. Also, menus can be created or positioned where a user is working, avoiding user interference. Finally, users can group menus into a document to use as a customized control panel. Some of the advantages of such active documents are described elsewhere in these proceedings and in a previous paper [Bier90].

3.9 Home Area Menus

Shared menus work well for commands, like changing color or deleting, that apply to several editors. Some functions, however, are specific to a particular editor: change of font, for instance, only makes sense in text editors. Menus of these functions need only be displayed when such an editor is in use. We display the menus for a user's selected editor in that user's home area. These menus can be combined with feedback that shows current modes in the selected editor. For example, figure 7(b) displays a menu of possible font choices with the user's currently selected font highlighted.

3.10 Editing at Different Levels

Users can edit simultaneously at different levels in the editor hierarchy. This capability reduces interference between users: one user can reposition or resize a document while another edits its contents. While some multi-level edits may disturb the user editing at a lower level, others work well. For example, a user may resize a rectangle editor to uncover an object beneath it without disturbing another user who is editing child rectangles that are far from the border being moved.

4 MMM's Architecture

Our user interface requires a software architecture that is unusual in several ways. It must support a hierarchy of concurrently active editors. It must handle multiple devices and users, and make links between them. Each input event must be directed to the correct editor, even though other input events may be modifying the editor tree. Each editor must store state for all of the users who have referenced it. Finally, the editors must coordinate their screen updates to produce a consistent image. In this section, we discuss the editor hierarchy, the preparation of new input events, the routing of events to editors, how editors update documents in response to events, and the algorithms for updating the screen.

4.1 The Hierarchy of Editors

Our editors are arranged in a tree. The editor at the root can have any number of child editors, each of which can have any number of children and so on (recall figure 5). Each editor has

an *application queue* of input events and its own lightweight *application process* (a thread) to dequeue and execute events. When an editor receives an input event, it may act on the event immediately, place it on its queue for sequenced execution, or pass it to a child editor.

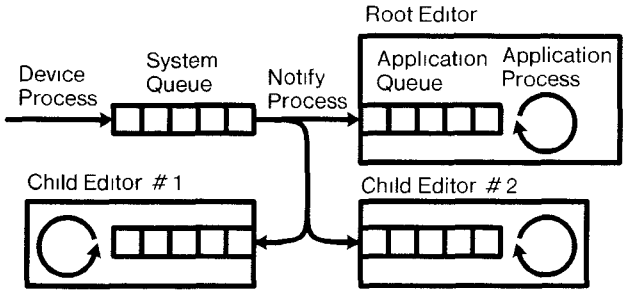

Figure 8. Input handling processes and queues.

At any moment, up to three editors are important to each user. A user's *selected editor* will receive his or her keyboard events and commands generated by clicking on a menu. If a user performs an interactive operation (dragging a rectangle, for example), the editor in which that operation began is the *mouse focus editor*. The mouse focus editor receives mouse motion and mouse button events until the interactive operation completes, so it is possible to drag objects out of view. The most deeply nested visible editor that contains a user's cursor is his or her *containing editor*. If no interactive operation is in progress, the containing editor receives all mouse motion and mouse button events.

4.2 Preparing Input Events

Figure 8 summarizes the processes and queues used in input handling. When the user manipulates an input device, MMM's *device process* builds an *event record* (see figure 9) to represent this event and fills in the time when the event occurred, the device, and the event type (mouse movement or key press, for example). The record is then placed on the *system queue*. The *device process* runs at a high priority so that the time-stamps will be close to real-time.

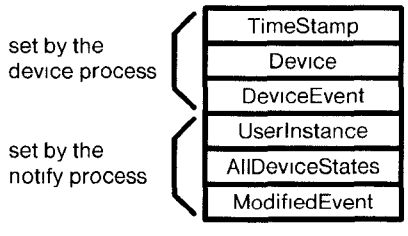

Figure 9. The *event record* data structure.

The *notify process* removes the record from the system queue and fills in the identity of the user (or user instance if a user has several home areas) who generated the event by looking up the association in the *device ownership table* and fills in the current state of all devices owned by that user (e.g., whether any mouse buttons are being held down). Only the notify process is allowed to modify the device ownership table, so the table lookup produces a consistent value. Storage is also allocated to hold local information as the event is passed between editors; for instance, x-y coordinates will be modified to reflect the local coordinate system of each editor.

4.3 Delivering Events to Editors

The notify process walks the tree of editors, beginning at the root, asking each editor either to accept the event or to pass the event to a child. When an editor accepts the event, the notify process performs any actions that must be done immediately (before the next event can be handled) and then dequeues the next event from the system queue and repeats this procedure.

Each editor in the tree is represented by an *editor record*, part of which is editor-specific and accessible only to that editor's application process and the rest of which is accessible to the notify process. All editor records include a list of *user-state records*, one for each user who has referenced the editor during the current session. Each user-state record describes the child editor that is on the path to the user's selected editor and the child editor that is on the path to the user's mouse focus editor. MMM can find a user's selected editor, for example, by chaining down the former path from the root editor.

If a user has a mouse focus editor, events are passed to the child editor that lies on the path to that editor; on the way, the event's position information must be translated into that child's coordinate system. If the application process of the parent editor is busy, the child may be moving, so the notify process waits for this application process to finish its current event before translating the coordinates. Then, if the child editor is the mouse focus editor, this child editor accepts the event. Otherwise the notify process continues down the path of editors.

The notify process must not wait too long; if it becomes stuck, all editors stop responding to users. We have added timeouts to avoid this problem. If the notify process needs to wait for more than a few seconds for an application process, it gives up, throws the event away, and marks that process as "sick." Subsequent events that must pass through a sick editor are discarded until its application process is idle, whereupon it is once again marked as "healthy." While events are sometimes lost by this scheme, we believe this is an acceptable price to pay for guaranteed responsiveness; ideally, synchronous collaborations should proceed at a conversational pace.

If there is no mouse focus editor, the notify process passes mouse events to the containing editor. As the event is passed down the tree, the notify process waits for each editor's application process to be idle before translating event coordinates and passing the event to a child editor. Because keyboard events do not contain coordinates, they are passed down the tree to the selected editor without any waiting.

4.4 Editor Input Handling

When an editor receives a user event, it looks in the editor-specific fields of its user-state record for that user to discover his or her current modes and preferences. Our rectangle editor, for example, stores the user's default color for creating rectangles, the user's interactive command (dragging versus resizing a rectangle) and the last coordinates of the user's cursor that were processed during dragging. In addition, each rectangle records a list of all users who have selected it.

Editors respond to events in two phases. First, the editor may request that it become the mouse focus or selected editor for the user who originated the event. This is done by the notify process to ensure that the chains of pointers in the editor hierarchy that represent the path to the mouse focus and selected editors are updated atomically. The editor may then

place the event on its application queue.

In the second phase, if any, the application process chooses an action from its queue and executes it. To choose an action, it inspects all the queued actions. It performs the oldest mandatory action, or if there are no mandatory actions, it executes the most recent optional actions (e.g., motions during rubber-banding are optional), skipping older ones. This helps the editor handle large volumes of input.

Editors place incoming events from *all* users on the same queue, so it is necessary to decide which user's event to respond to first. Mandatory actions are handled in the order in which they are received, regardless of user. For optional actions, the editor selects the user who has been least recently served, so the editor feels equally responsive to all users.

4.5 Screen Update

Any application process can make changes that require the screen to be refreshed. Objects and editors can overlap so refreshing the screen for one editor may require the cooperation of both parent and sibling editors. We could allow each application process to walk the entire editor tree as needed to update the screen after a change, but this might allow one process to read editor data while another is modifying it.

Instead, when an editor, *E*, wishes to paint, it creates a *paint request record* which includes (in the coordinates of *E*) a rectangle that bounds the region to be repainted. The editor notifies MMM that it would like to paint and then turns off its application process and waits. The system asks all application processes to pause and waits until they all have stopped, then the *system paint process* walks the tree of editors to determine a rectangle or rectangles bounding all the regions that have changed. It then walks the tree again asking each editor in turn to repaint itself within these rectangles. The resulting paint actions are double-buffered so that they appear to the user all at once, enhancing the perception of simultaneous motion. Once all editors have finished painting, their application processes are reactivated.

To test screen update, we built an application, called a windmill, that constantly sends paint requests to draw a rotating animation. Figure 10 shows two windmills. When both windmills are painting at once, MMM will update the screen all at once, using the rectangle that bounds both of these rectangles (shown in thick black).

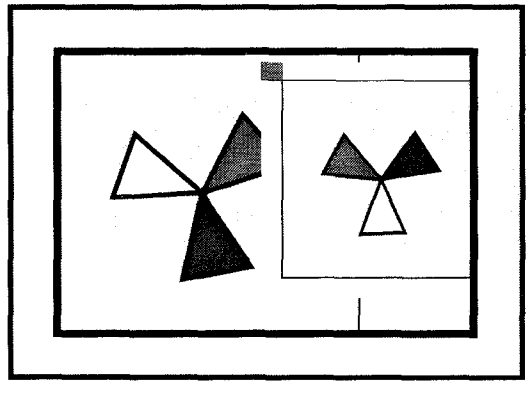

Figure 10. Two rotating windmills and the rectangle that bounds their refresh region.

5 Early Experiences and Conclusions

While still in its early stages, MMM has already proved valuable for testing ideas about the architecture and user interface of multi-user editors and for demonstrating that fine-grained editing with per-user customization can be achieved with reasonable performance and only a modest increase in data structure complexity.

Because it is a toy system, MMM does not have a large community of users. However, we believe that many of the ideas embodied in MMM will be of value in other projects. From our experiences, we draw these conclusions about the user interface of shared editors:

(1) The home area is effective as a quick way to register a pointing device with a user and as a place in which to display per-user feedback.

(2) Selecting operations by pointing at specific object positions works well because it is easy to learn and hard to misinterpret as a gesture.

(3) When editors are used as window systems, users can work at different document levels at once and can copy control objects, such as menus, into editors for convenient access.

(4) Using color to associate feedback with particular users allows the use of traditional shapes and sizes of feedback, such as underlines in the text editor and control point highlighting in the rectangle editor. However, this technique is less effective on black and white displays or with color-blind users.

(5) Allowing menus to be shared among users and among editors simplifies the interface and conserves screen space.

In order to support this user interface, we provide these four elements of system architecture:

(1) When an input event arrives, we identify which user it comes from using a table of user-device associations.

(2) Events with coordinates, when destined for a nested editor, wait until that editor has a well-defined position relative to its parent before they are delivered to the nested editor.

(3) Many editor data structures, including those representing the current modes, operations, insertion point, and selected objects, are replicated once for each active user.

(4) All editors stop manipulating their document data structures during screen refresh to produce a consistent view of the screen and a smooth view of simultaneous motions.

While we built MMM in the Cedar programming language, using lightweight processes that run in a single virtual address space, a modified version of the architecture could be implemented on other platforms. If multiple address spaces were used, application data would be manipulated solely by an application process; the notify process and system paint process would negotiate with the application processes to process input events and screen refresh requests.

MMM could also be extended to support use from multiple workstations, using a centralized server architecture. A single instance of the MMM system could collect input from multiple workstations and distribute screen update actions to these

workstations using protocols like those used in the X Window System.

In the future, we plan to extend the MMM user interface and architecture. We will experiment with forms of user feedback that function well on black-and-white displays. Using the MMM framework, we plan to build a set of practical applications that support cooperative work, giving us a chance to see how well these ideas perform when editor complexity increases and actual user demands are placed on the system. Finally, we plan to use the MMM architecture to experiment with user interfaces that allow a single user to edit by pointing with both hands at once, using either touch-sensitive displays, electronic pens, or a combination.

Acknowledgments

We thank Randy Pausch for helping us map out some of the hard problems of input handling in multi-user systems in the early stages of the project. We thank Ken Pier for writing the software that enabled multiple mice and multiple cursors in our programming environment. We thank Polle Zellweger and Wendy Mackay for suggesting changes to the text and figures that resulted in a more readable paper. We thank Dan Swinehart for comments that will motivate further research in this area. Finally, we thank Xerox and Rank Xerox for supporting the MMM project and providing the computers, software and networks that made it easy to co-author this paper across the Atlantic.

References

[Bier90] Eric A. Bier and Aaron Goodisman. Documents as user interfaces. In R. Furuta, editor. *EP90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, Cambridge University Press, 1990, pages 249-262.

[Bobrow90] Daniel G. Bobrow, Mark Stefik, Gregg Foster, Frank Halasz, Stanley Lanning, and Deborah Tatar. *The Colab Project Final Report*, Xerox PARC Technical Report SSL-90-45.

[Buxton86] William Buxton and Brad A Myers. A study in two-handed input. In *Proceedings of CHI '86 Human Factors in Computing Systems* (Boston, April), ACM, 1986, pages 321-326.

[Crowley90] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. MMConf: An infrastructure for building shared multimedia applications. In *Proceedings of the Conference on Computer-Supported Cooperative Work* (Los Angeles, October), ACM, 1990, pages 329-342.

[Ellis90] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein Design and use of a group editor. In G. Cockton, editor, *Engineering for Human-Computer Interaction*, North-Holland, Amsterdam, 1990, pages 13-25.

[Ensor88] J. Robert Ensor, S. R. Ahuja, David N. Horn, and S. E. Lucco. The Rapport multimedia conferencing system—a software overview. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, IEEE, March 1988, pages 52-58.

[Kraemer88] K.L. Kraemer, J. King. Computer-based systems for cooperative work and group decision making. In *ACM Computing Surveys*, Vol. 20, No. 2: ACM 1988, pages 115-146.

[Lantz86] K.A. Lantz. An experiment in integrated multimedia conferencing. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, ACM 1986, pages 267-275. Reprinted in I. Greif, editor, *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann Publishers, 1988, pages 533-552.

[Lauwers90] J. Chris Lauwers, Thomas A. Joseph, Keith A. Lantz and Allyn L Romanow. Replicated architectures for shared window systems: a critique. In *Proceedings of the Conference on Office Information Systems* (Cambridge, Massachusetts, April), ACM 1990, pages 249-260

[Minneman91] Scott L. Minneman and Sara A. Bly Managing à trois: a study of a multi-user drawing tool in distributed design work. In *Proceedings of CHI '91 Human Factors in Computing Systems*, ACM, 1991, pages 217-224.

[Nelson91] Greg Nelson, editor. *Systems Programming with Modula-3*, chapter 7. Prentice Hall, 1991.

[Olson90a] Judith S. Olson, Gary M. Olson, Lisbeth A. Mack, and Pierre Wellner. Concurrent editing: the group's interface. In *Proceedings of Interact '90—The IFIP TC 13 Third International Conference on Human-Computer Interaction* (Cambridge, UK, August), 1990, pages 835-840.

[Olson90b] Gary M. Olson. *Technology Support for Collaborative Workgroups*. Annual progress report for 1989-90 to the National Science Foundation, Grant #IRI-8902930, University of Michigan, June 1990.

[Scheifler86] Robert W Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986, pages 79-109.

[Smith89] Randall B. Smith, Tim O'Shea, Claire O'Malley, Eileen Scanlon, and Josie Taylor. Preliminary experiments with a distributed, multi-media, problem solving environment. In *Proceedings of the First European Conference on Computer Supported Cooperative Work* (Gatwick, UK) 1989, pages 19-34.

[Stefik87a] 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*, Vol. 30, No. 1, January 1987, pages 32-47.

[Stefik87b] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS Revised: Early experiences with multi-user interfaces *ACM Transactions on Office Information Systems*, Vol. 5, No. 2, April 1987, pages 147-167.

[Swinehart86] Daniel C. Swinehart, Polle T Zellweger, Richard J. Beach, and Robert B Hagmann. A structural view of the Cedar programming environment, *ACM Transactions on Programming Languages and Systems*, Vol. 8, No 4, 1986, pages 419-490. Also available as Xerox PARC Technical Report CSL-86-1.