Proceedings of the Third European Conference on Computer-Supported Cooperative Work
13-17 September, 1993, Milan, Italy
G. De Michelis, C. Simone and K Schmidt (Editors)

Building Shared Graphical Editors Using the Abstraction-Link-View Architecture

Tom Brinck and Ralph D. Hill
Bellcore, USA

Abstract: We have written several multi-user graphical editors in the Rendezvous™[1] system. In our approach to building these editors, the applications are first written as single-user editors. When multiple users wish to share a drawing surface, the drawing surfaces of their individual editors are connected using the Abstraction-Link-View (ALV) architecture. "Links" communicate the editing operations among the editors they connect. Links are designed to be invisible to the applications they are attached to, allowing the interface for each user to be highly customized. Links can also attach editors to the interface of a running RENDEZVOUS application, allowing the interface to be edited as the application is being used.

Introduction

We have written several multi-user drawing programs and we describe here an architecture for building such applications. Many multi-user drawing programs have been reported in the literature and several are reviewed by Brinck and Gomez (1992). The design of these drawing programs differ in significant ways, especially with regard to multi-user characteristics. Our goal has been to design an infrastructure which makes it easy for us to build multi-user drawing applications and easily make modifications to explore the implications of various design alternatives.

This paper begins by discussing our architecture as it applies to building single-user drawing programs. Our toolkit provides color palettes, tool palettes, drawing surfaces, and a set of plug-in tools for creating and manipulating object-oriented graphics. These facilities are all provided as part of the

1. Rendezvous is a trademark of Bellcore.

RENDEZVOUS system (Hill et al, 1993), a user-interface development system designed at Bellcore. It is important to note that the plug-in tools are supported at the toolkit level and not at the level of drawing programs. This means the plug-in tools are not restricted to being tools for editing drawings.

By viewing a drawing program as a graphical viewer of structured information it becomes possible to use a single-user drawing program written in the RENDEZVOUS system to edit other things, for example, the user interface of any other RENDEZVOUS application. The drawing program is attached to the interface of the other application with "links". Links communicate the results of editing operations between the drawing editor and the application. Toolkit level support for plug-in editing tools makes it possible to build tools for editing user interfaces.

This link-based architecture also allows us to connect a drawing application of one user to the drawing application of another user, resulting in a multi-user drawing application. Because links are separate from the applications, each of the drawing editors can be authored independently, and have different user interfaces. Links can provide various viewing transformations, so the interfaces for each user can be highly customized. This paper discusses our architecture for multi-user applications, explores the impact of multi-user editors on tool design, and discusses other design issues for multi-user graphic editors.

Single-user Graphical Editors

In conventional graphical interfaces, several interaction techniques are common for allowing a user to edit a drawing surface (we'll call it a "canvas"). The most common approach is for the user to select a tool from a tool palette, then click and drag on the canvas to create an object such as a box or line.

Several other interaction techniques are common. The user can push a button or select a menu item to perform a *global canvas operation*, such as clearing the canvas or saving and loading. The user can also *select* an object (possibly using the selection tool) and then operate on the selection by dragging or resizing the selection rectangle or by choosing a menu item that operates on the current selection. The user might also grab an object from a *well*. The user can click on a well to get an object to drag onto the canvas. For instance, an image well provides a stack of images from which you can pull off the top image and add it to the canvas (this could be useful for a deck of cards). A color well allows you to drag a color chip, and by dropping it on an object, the object becomes that color. All of these interaction techniques are available to various degrees in the RENDEZVOUS system.

As an example of a drawing program we've built, the Conversation Board is shown in Figure 1 (Brinck 1992, Brinck and Gomez 1992). The Conversation Board can be used as either a single-user drawing program or a multi-user

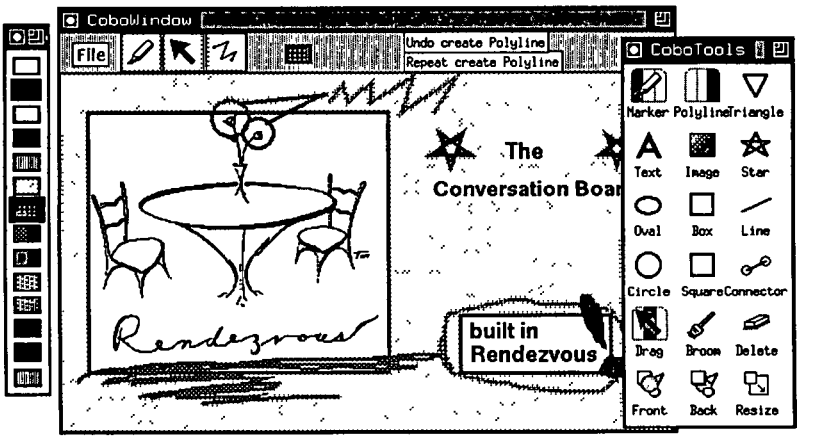


Figure 1. The Conversation Board is a multi-user structured graphics editor designed for informal conversational use.

program. It is a structured graphics editor that supports drawing with colored markers, text, and various geometric shapes, such as ovals, lines, and rectangles. It includes telepointers and supports saving and loading drawings, importing images, cut and paste, and undo. At the top-level, a drawing program like the Conversation Board consists of a canvas, a tool palette, a color palette, a line style palette, and various other possible palettes and menu options.

Plug-in Tools

When the user chooses a tool, that tool is used for interpreting what action needs to be taken when the user clicks on the canvas. The canvas acts as a dispatcher for mouse events. When the user clicks on the canvas, the canvas forwards this event to the currently selected tool, informing the tool of where the mouse event occurred and what object was clicked. The tool then performs its appropriate action.

The RENDEZVOUS language is written as an extension to the Common Lisp Object System (CLOS). CLOS provides a flexible object-oriented programming system. To encapsulate the behavior of tools, we define a class called Tool. Any subclass of Tool defines a method for handling mouse events forwarded from the canvas. Whenever a Tool handles a mouse interaction, it registers it's action with the UndoManager (an object associated globally with an application), so that the last action performed can be undone or repeated. A Tool also specifies an icon, a cursor, some explanatory text for providing the user with help information about the tool, and an optional dialog for the user to edit the default parameters of the tool. In order for plug-in tools to be possible, the graphic objects that they manipulate must be in a well-known structure and must have a standard set of properties (such as size and color) that tools can access. The RENDEZVOUS

system provides this functionality with a well-specified declarative graphics system.

By incorporating all the functionality of a tool into a single class, it becomes possible to write a new tool without modifying any other part of the application and to simply "plug" the tool into any RENDEZVOUS application that uses tools. Currently we have over 40 standard tools defined, a few of which are in the tool palette of Figure 1. This encapsulation of tools allows us to customize drawing applications quickly. Since graphic objects on the canvas do not have to incorporate any editing behavior, the canvas serves as a generic editor of any graphic elements. As a demonstration of this, the next section describes how a generic editor is used to edit user interfaces.

Graphics Editors as Views

Once a graphical editor is written using the RENDEZVOUS toolkit, it can be linked to another RENDEZVOUS application to edit the interface to that application. For instance, suppose a user is playing a Tic-Tac-Toe game written in the RENDEZVOUS system. A graphical editor can be started and attached with a set of links to the Tic-Tac-Toe game, as shown in Figure 2. In the Abstraction-Link-View (ALV) paradigm described by Hill (1992), the Tic-Tac-Toe game serves as the "abstraction" for which the graphics editor is the "view".

The links between the "abstraction" and view are objects containing constraints that exist outside of both the abstraction and the view. Constraints define relationships between objects (as described, for instance, by Myers et al, 1990), so that when a value of one object gets updated, the constraint will automatically update the value of corresponding objects. The RENDEZVOUS language provides constraints integrated into the base language. Constraints may be specified without making modifications to any of the objects that they specify relations between. Thus, no modifications had to be made to either the Tic-Tac-Toe program or the graphics editor to make this linking possible.

Once linked, any modification made by either application is propagated to the other application, provided that a link has been written for the kind of property that has changed. For instance, if the drawing editor changes the color of a circle to red, the color of the circle in the Tic-Tac-Toe application changes to red. If the Tic-Tac-Toe player places an X into a cell, then an X gets created in the drawing editor. Note that the Tic-Tac-Toe player can continue playing even while the application is being edited!

There are two major types of links involved here. The appendix provides some examples of the code used to implement these links. The most common type of link simply links properties, such as color or position, of an object in the

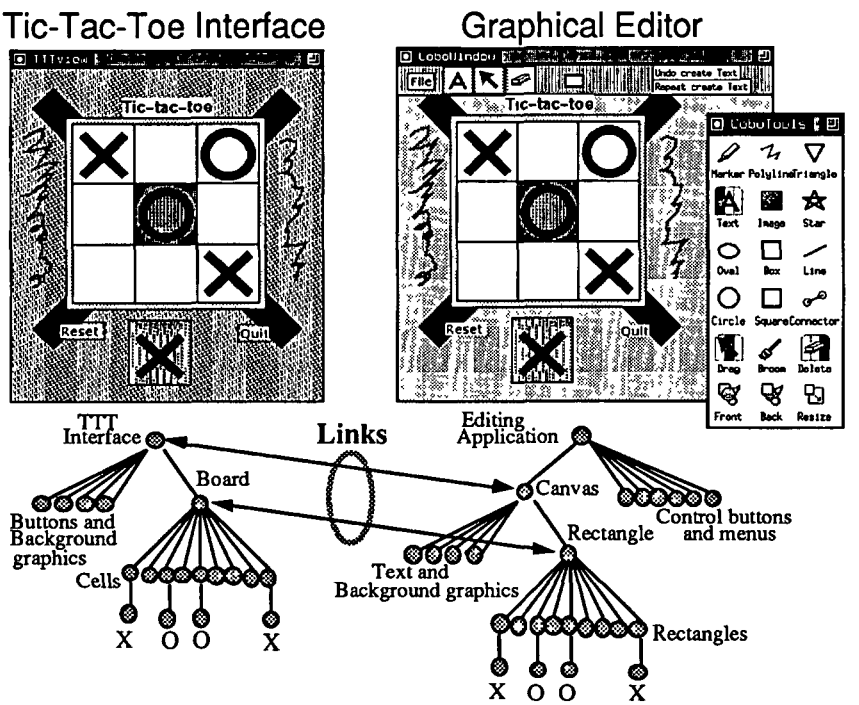


Figure 2. A Tic-Tac-Toe program on the left is linked to the canvas portion of a drawing editor on the right. Using the editor, the user has colored the central cell and added some decoration on the sides and the title text "Tic-tac-toe". The diagram shows the simplified structure of each view, and the top-level objects that get linked.

view with an object in the abstraction. This is implemented with a straightforward equality constraint.

A more sophisticated type of link is the TreeMaintenance link. This maintains the structure of a tree between a view and the abstraction. In the RENDEZVOUS system, all user views are modeled as trees of graphical objects, as in the diagram of Figure 2. The window is the object at the root of the tree. In a drawing program, the canvas is the parent of the objects that are being drawn. In Tic-Tac-Toe, the board is a parent of the cells in which the X and O get drawn. A TreeMaintenance link is attached to a tree node in the view and a tree node in the abstraction. It maintains consistency between the lists of children for the two tree nodes by creating and deleting objects in the lists of children. For the board object in Tic-Tac-Toe, the TreeMaintenance link creates a rectangle object in the graphics editor. It also creates a new link between the board and the rectangle that maintains their position, size, and color. This link must also be a TreeMaintenance link in order to recursively maintain the list of children of the board.

A TreeMaintenance link must work in two directions — when an object gets created or deleted in either the view or abstraction, the corresponding object

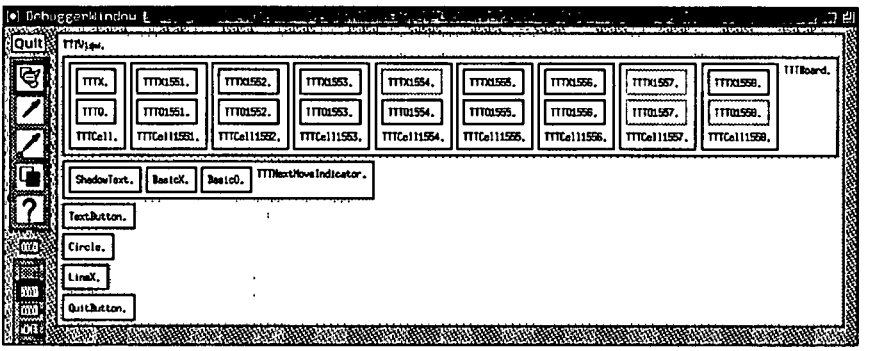


Figure 3. This is a debugger that shows a hierarchical view of the objects of any application it is linked to, using embedded boxes. This one is linked to the Tic-Tac-Toe program.

must be created in the abstraction or view, respectively. A list of maintenance entries indicates what kind of object to create and what kind of link to build. For shared drawing, the standard TreeMaintenance link maps a rectangle to a rectangle, an oval to an oval, and so on, using a standard link for graphics. The Tic-Tac-Toe board (an instance of class TTTBoard) gets mapped to a rectangle because TTTBoard is a subclass of Rectangle.

Other types of TreeMaintenance links are possible. For instance, Figure 3 shows a view used for debugging that maps any abstraction object to a rectangle labeled with the name of the object, and which lays out the children of the object inside of it, resulting in a view of the hierarchical structure of the application. Though this debugger looks nothing like a drawing program, we were able to reuse many of the tools built for manipulating graphics to allow us to query and set properties of the objects viewed in the debugger. This capability of multiple different views onto a single application is similar in spirit to that of Avrahami et al (1989). Their FormsVBT system provides a text and a graphics view of a dialog editor, as well as an operating view of the resulting dialog. The Abstraction-Link-View architecture is not restricted to dialogs, but also allows editors to be attached to highly dynamic and interactive applications.

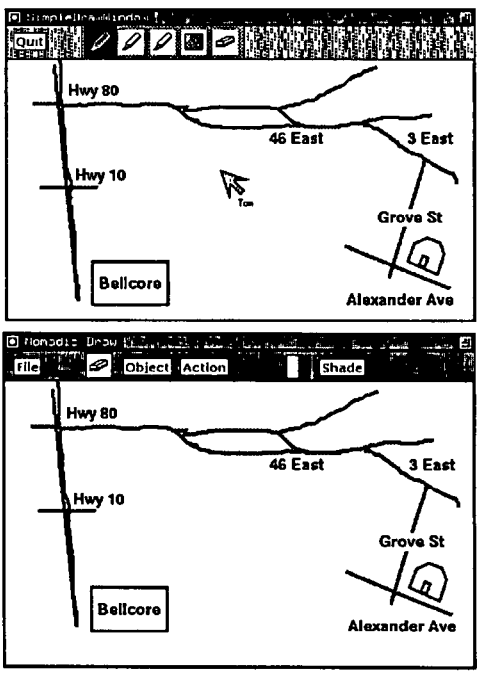
Multiple Users

To implement two-user drawing, two canvasses are linked with the standard graphical TreeMaintenance link. Links were originally designed for connecting abstraction processes to view processes, but they work equally well for peer-to-peer linking. When two canvasses are linked, any modification one user makes is immediately reflected in the view of the other user.

In order to have more than two users, more graphics editors can be linked to views in the existing session, and the links will maintain consistency among them

all. For convenience, we generally designate one of the editors as the central shared abstraction, and link the view of each user directly to the central abstraction.

The ability to connect independently-written applications using links stems from the fact that links are a mechanism built on top of a constraint maintenance system. All the applications that get connected are written in the RENDEZVOUS system, and they therefore run in a common environment in which constraints can be maintained. Similar effects could potentially be achieved in a toolkit without a constraint maintenance system if a standard graphics library is provided and callbacks are attached to all the standard properties of graphic objects. In such a system, "links" would be processes which registered with the callbacks of a view process and an abstraction process and sent updates back and forth between them. A mechanism to prevent cycles would be necessary, either in the links or as part of the callbacks in the standard graphics library. While this approach lacks some of the generality and elegance of a constraint maintenance system, it does allow the interoperability described here, and in a fashion which remains invisible to the application programmer using the standard graphics library.



Customized views

Since the drawing application of each user is only linked at the level of the canvas, it is possible to design very different interfaces for each user, while they each edit the same drawing. In Figure 4 we show an example of 3 different editors being used to simultaneously edit the same drawing. Each of these views was developed as an independent

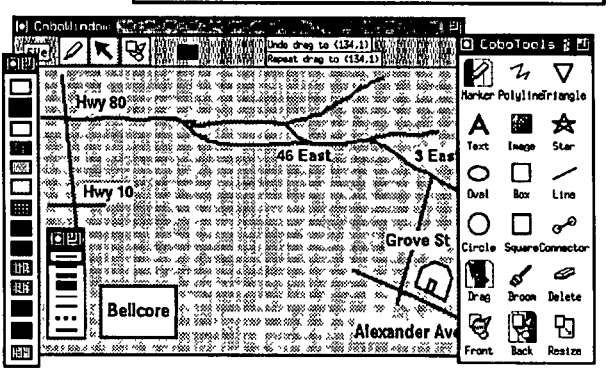


Figure 4. These 3 different graphics editors — SimpleDraw (top), NomadicDraw (middle), and the Conversation Board (bottom) — have their canvasses linked to the same abstraction.

drawing program. The Conversation Board was described earlier. SimpleDraw is an application written to provide minimal drawing functionality so that it would be as simple as possible. NomadicDraw is a drawing program designed to be used on existing portable computers using wireless networks. These editors differ in the set of tools they offer, the interface for accessing tools, and various other drawing options.

Our multi-user architecture has several important advantages over others such as GroupDraw (Greenberg et al, 1992), WScrawl (Wilson, 1992), and Ensemble (Newman-Wolfe et al, 1992). For instance, DistEdit (Knister and Prakash, 1990) is a system for building shared text editors that also allows the editors to be authored independently. However, DistEdit does not allow users to edit the text simultaneously and requires that each editor be modified to make explicit calls to the DistEdit routines. In our approach, customized interfaces which share a drawing surface can easily be authored independently, the users can interact with the application simultaneously and without any sort of locking mechanism, and the programmer can author the editing application independent of the communication, which we embed in the links.

Impact of multiple users on the design of tools

Now that we can build a multi-user graphical editor, it is important to go back and look at what impact this has on the development of our architecture for single-user editing. For almost all the functionality defined, multiple users do not affect the way in which the single-user application is implemented since links act as the sharing mechanism, and the links are not embedded in the applications. However, some of the tools have to be written carefully. For instance, what happens when one user resizes an object while another one drags it or even deletes it? We could easily lock an object so that others can't access it while one person is making modifications. Instead, our default tools simply try to do their own actions, while verifying that those actions are still valid. Thus, a tool for resizing could verify that an object still exists before it tries to change the size of the object. Instead, the resize tool changes the size attribute of an object even after it has been "deleted". Deletion merely removes an object from our graphical tree, but the storage for this object isn't actually reclaimed until all pointers to the object are removed. Therefore, most manipulations remain valid.

Multi-user undo

The implementation of undo is also affected by multiple users. Even in implementing a mechanism for undoing *only* the last action in the view that performed the action, all the problems of multi-user undo appear. For example, suppose user A drags an object across the canvas. To undo this, the application only needs to move the object back to its original position. However, suppose

that user B then deletes the object. Now if user A tries to undo the action of dragging, either the object should be undeleted and moved back or the object should just be left deleted. In either case, undo must be able to handle changes that wouldn't have occurred in the single-user case. In fact, since user B can make any number of changes before user A chooses to undo, the implementation of undo must be robust with respect to an arbitrary number of intervening operations. Thus, as each tool implements the method for undoing its actions, it must check carefully that all state that is necessary to undo the action is still valid, and if not, then act appropriately. This may mean that the action cannot be undone. In practice, this isn't a problem for the user since it is rare that an action can no longer be undone, and in the case where an action can't be undone, the user rarely has the impression that the action should still be undoable.

Multi-user undo has also been described by Prakash and Knister (1992) as applied to shared text editors. The issues they discuss are similar to the experiences we've had. Our implementation differs from theirs in where conflicts are detected and resolved. The situation described above, where user B deleted an object that user A dragged, is one example conflict. When user A wants to undo the dragging operation, the deletion operation might also need to be undone.

In their approach, this conflict is detected globally by an algorithm that compares all succeeding operations to determine if they conflict. A conflict between dragging and deleting is potentially resolvable by providing the user the option of first undoing the deletion operation.

In our approach, the conflict is detected locally, by the Tool that originally performed the dragging operation (the DragTool). When the DragTool is told to undo an action, rather than checking the list of operations that have come after it for potential conflicts, it merely checks that the appropriate state is still valid. That is, the DragTool confirms that the object that was dragged still exists before it moves the object back into position. This approach is more efficient than comparing all succeeding operations, since only one property of the current state needs to be tested, whereas an arbitrary number of operations could have followed. Note that both approaches require the programmer to anticipate all potential conflicts between two operations.

Multi-user interface editing of multi-user applications

Since multiple views can be attached to the same abstraction, this paradigm can be extended to allow multiple users to edit a multi-user application. In Figure 5, a two-user Tic-Tac-Toe game consists of a Tic-Tac-Toe abstraction and two views. Each of these views can have multiple drawing editors attached to them, allowing multiple users to be modifying the application simultaneously, even as the two users are playing a game of Tic-Tac-Toe.

While this is a first step in designing a multi-user interface builder, some difficulties remain. Attaching a generic drawing editor to a user interface allows a

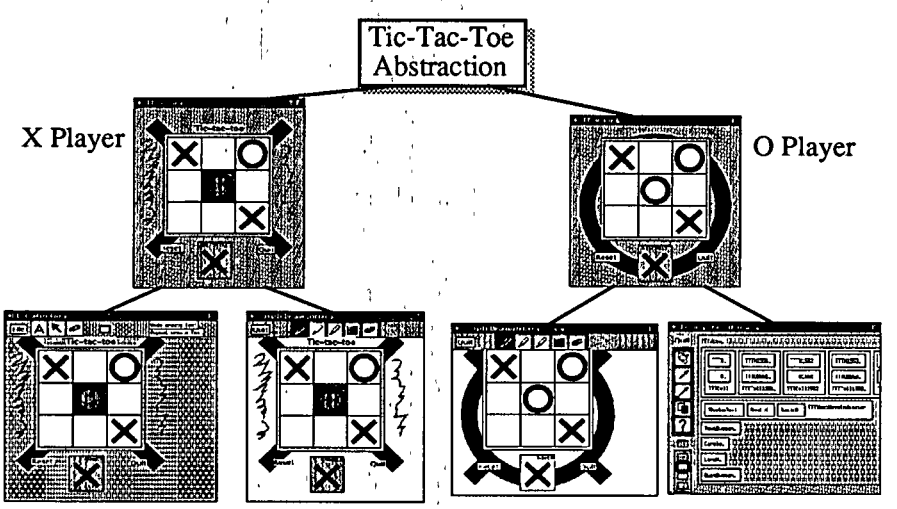


Figure 5. Multi-user editing of a multi-user Tic-Tac-Toe application. The X Player and O Player are the actual players of the game. The bottom row shows a set of editors and a debugger of these running applications.

programmer or designer to manipulate the graphic components of the interface by repositioning or resizing them or to add other graphics which have no behavior, such as static text or freehand sketches. To add components which have *behavior*, the drawing tool needs to be extended to support objects with behavior, such as buttons and menus.

Also, in a multi-user application, it is often not sufficient to be editing only a single view. In the example of Figure 5, modifications to a Tic-Tac-Toe view are not shared with the other Tic-Tac-Toe view unless the appropriate types of objects are linked through the links defined in the Tic-Tac-Toe application. If the developer of the Tic-Tac-Toe application did not anticipate a certain class of objects being created (such as menus), then an appropriate link to create menus in other views is not likely to exist. In addition, the interface developer may want to have an object appear in different ways for different views (a table of numbers in one view might be displayed as a bar chart in another view). This requires that the interface developer of a multi-user application also have a means of editing the links between the views and editing the objects that appear in the abstraction.

Other multi-user policies

SimpleDraw, NomadicDraw, and the Conversation Board all implement a shared canvas where users can all manipulate the objects simultaneously. For example, while one user is resizing an object, another can change its color, or even drag it. We have built a few other drawing editors based on this architecture that implement different multi-user policies.

Floor control

Several types of floor-control policies have been implemented, including token-passing and locks that time out. For example, a token-passing version of SimpleDraw (Figure 6) allows only one user to draw at a time. When a user has the token, drawing is possible. Other users can request the token, and acquire it when the user who possesses it decides to release it.

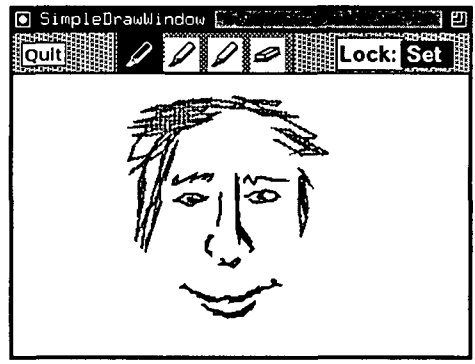


Figure 6. A token-passing version of SimpleDraw allows only one user to draw at a time. The Lock button is pushed to acquire the token. This user has the token, so the interface indicates that the Lock is Set.

The floor-control mechanism is also built using the Abstraction-Link-View architecture. The view contains buttons which allow a user to request and release a lock. A link sends these requests to the lock-abstraction. The lock-abstraction is shared by all the views, contains information about who holds the lock, and controls transfer of the lock. In the view, the lock determines whether a user's input will be ignored.

The floor-control versions of SimpleDraw cannot be used to edit the interface of any arbitrary RENDEZVOUS application. Since the lock requires a lock-abstraction, any interface being edited must also have a lock-abstraction for the lock to operate.

Other drawing editors, such as the Conversation Board, can be attached to the same abstraction as the floor-control versions of SimpleDraw, but these other editors will not be mediated by the floor-control because their interfaces have no floor-control mechanism. Thus, Conversation Board users will always be able to draw regardless of who has the floor.

CanvasTalk

CanvasTalk (Figure 7) is similar in spirit to the UNIX™[2] 'talk' program, except that instead of typing, the users draw. Any number of users can join a session and each user has a personal canvas (in white) to drawn on. Each user sees the canvasses of the other users (in grey), but cannot edit them. A user can drag a copy of a sketch on another user's canvas onto the personal canvas.

CanvasTalk requires every view to be updated when a user enters or leaves a session because there is a canvas for each user in the session. This is done by using a TreeMaintenance link (described earlier) to maintain the number of canvasses between the view and the abstraction. When a new user starts up a

2. UNIX is a registered trademark of Unix Systems Laboratories.

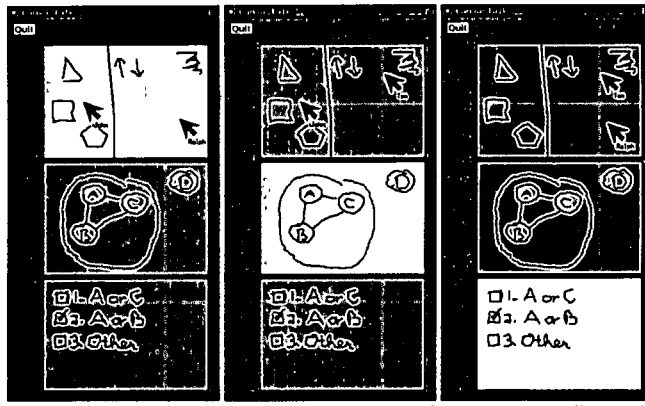


Figure 7. CanvasTalk gives each user a personal canvas to edit, and allows the user to view, but not modify, the canvasses of other users.

view, the view creates a local canvas. The TreeMaintenance link then creates a corresponding canvas in the abstraction and across to other views. When a user leaves a session, the view first deletes the local canvas before detaching from the abstraction. The TreeMaintenance link deletes the corresponding canvas from the abstraction and then from other views.

In this case, the view was actually written with explicit knowledge of how the link worked. In previous examples, it was possible to write an editor as if it were a single-user editor and ignoring communication issues. When interfaces provide an explicit representation of other users, it is not entirely possible to ignore how information about those other users is communicated.

Conclusion

We have designed an architecture for building multi-user graphics editors, which serves as the foundation for the applications SimpleDraw, NomadicDraw, the Conversation Board, a simple debugger, several floor-control variations of SimpleDraw, and CanvasTalk. These programs were made possible in the RENDEZVOUS system because of an architecture with the following characteristics:

- A large variety of plug-in tools can be made available to an application, not just to create graphics, but also to modify and manipulate structured graphics. This allows the editing of arbitrary graphic elements in the system, since the graphic elements do not have to handle user input.
- A simple and fast undo mechanism which works equally well for single-user and multi-user applications.
- The links between different views of a shared canvas are invisible to the applications they link, so the editors can be written independently of the communication between them.

- Multi-user editors can easily have a high degree of customization for the individual users.
- Multi-user editing can be done on the interfaces to other applications, allowing those interfaces to be edited even as the applications are running and being used. This is particularly useful in interface design and debugging.

Potentially this approach or a similar one could lead to a standard for interoperability between shared graphics editors that would allow users to purchase different applications, but still be able to draw together.

Acknowledgements

Kimberly Passerella wrote NomadicDraw. John Boyd wrote variations of SimpleDraw with different floor-control policies. John Patterson developed the TreeMaintenance link. Steve Rohall and Anthony Dayao gave helpful comments on this paper.

References

Avrahami, G., Brooks, K.P., and Brown, M.H. (1989): "A two-view approach to constructing user interfaces", *Proceedings of SIGGRAPH '89*, Boston, MA, July 1989. *ACM Computer Graphics*, vol. 23, no. 3, pp. 137-146.

Brinck, T. (1992): "The Conversation Board", *Video Proceedings of CSCW '92*, Toronto, Ontario, Canada, Oct 1992.

Brinck, T. and Gomez, L.M. (1992): "A collaborative medium for the support of conversational props", *Proceedings of CSCW '92*, Toronto, Ontario, Canada, Oct 1992, pp. 171-178.

Greenberg, S., Roseman, M., Webster, D., Bohnet, R. (1992): "Human and technical factors of distributed group drawing tools", *Interacting with Computers*, special edition on CSCW, 1992.

Hill, R.D., Brinck, T., Patterson, J.F., Rohall, S.L., and Wilner, W.T. (1993): "The Rendezvous language and architecture", *Communications of the ACM*, vol. 36, no. 1, Jan. 1993, pp. 62-67.

Hill, R.D. (1992): "The Abstraction-Link-View paradigm: using constraints to connect user interfaces to applications", *Proceedings of CHI '92*, Monterey, CA, May 1992. pp. 335-342.

Knister, M.J. and Prakash, A. (1990): "DistEdit: A distributed toolkit for supporting multiple group editors", *Proceedings of CSCW'90*, Los Angeles, CA, Oct 1990, pp. 343-355.

Myers, B.A. and al. (1990): "Garnet: comprehensive support for graphical, highly-interactive user interfaces", *IEEE Computer*, vol. 23, no. 11, 1990, pp. 71-85.

Newman-Wolfe, R.E., Webb, M.L., and Montes, M. (1992): "Implicit locking in the Ensemble concurrent object-oriented graphics editor", *Proceedings of CSCW'92*, Toronto, Ontario, Canada, Oct 1992, pp. 265-272.

Prakash, A. and Knister, M.J. (1992): "Undoing actions in collaborative work", *Proceedings of CSCW'92*, Toronto, Ontario, Canada, Oct 1992, pp. 273-280.

Wilson, B. (1992): "WSCRAWL 2.0: a shared whiteboard based on X-Windows", *Apple Computer Technical Report*. August, 1992.

Appendix

This appendix provides brief examples of how links are written in the Rendezvous language. Below is the definition of a simple equality link called TypicalGraphicLink which contains constraints to maintain the color, line style, position, and size between an object in the abstraction and an object in the view. When this link is created, the programmer specifies pointers to the abstraction and view objects, and a set of constraints are automatically generated to maintain these equalities.

```
(defClass TypicalGraphicLink (Link)
     (:add-dependencies
          (link= borderColor fillColor
                 lineStyle lineWidth
                 x y deltaX deltaY)))
```

An example definition of a TreeMaintenance link is shown below. This handles creating and deleting rectangles in the view and abstraction. RectangleMaintenanceLink contains a single maintenance entry which states that rectangles map to rectangles, and when rectangles gets created, TypicalGraphicLink should be installed between the corresponding view and abstraction rectangles. Thus, when a user creates a rectangle in a view (for instance, by drawing a rectangle in a graphics editor), another rectangle will be created in the abstraction, and the two rectangles will be linked so that their color and other properties are kept consistent.

```
(defClass RectangleMaintenanceLink (TreeMaintenance Link)
     (:maintenance-list
        (list
           (make-maintenanceEntry
            :abstractionType 'Rectangle
            :viewType 'Rectangle
            :linkType 'TypicalGraphicLink)
        )))
```