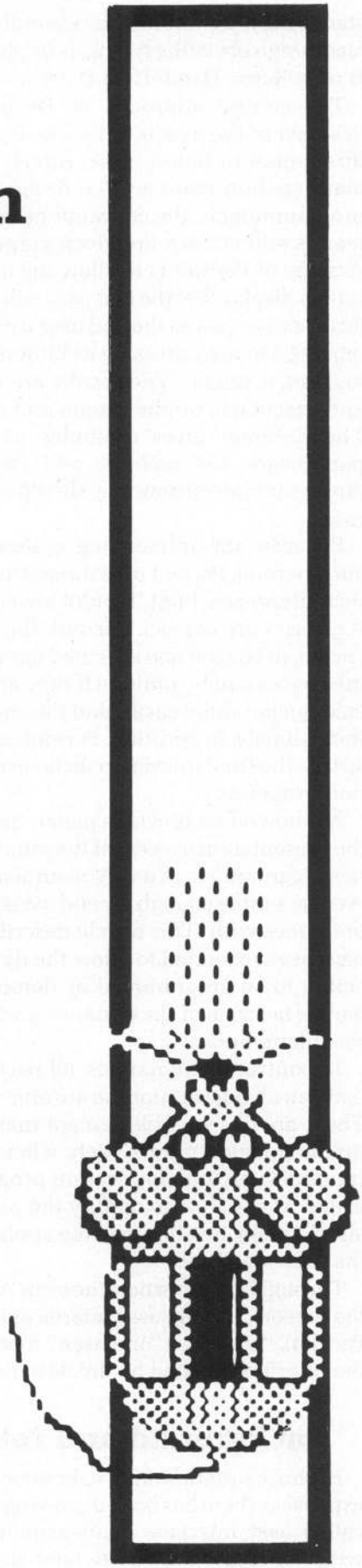
Creating Interaction Techniques by Demonstration

Brad A. Myers
University of Toronto*

When creating highly interactive, direct-manipulation interfaces, one of the most difficult design and implementation tasks is handling the mouse and other input devices. Peridot, a new user interface management system, addresses this problem by allowing the designer of the user interface to demonstrate how the input devices should be handled by giving an example of the interface in action. The designer uses sample values for parameters, and the system automatically infers the general operation and creates the code. After an interaction is specified, it can be executed immediately and edited. This promotes extremely rapid prototyping, since it is very easy to design, implement, and modify mouse-based interfaces.

Peridot also supports such additional input devices as touch tablets, as well as multiple input devices operating in parallel (for example, one in each hand) in a natural, easy-to-specify manner. All interaction techniques are implemented using *active values,* which are like variables except that the objects that depend on active values are updated immediately whenever they change. Active values are a straightforward and efficient mechanism for implementing dynamic interactions.



Peridot is an experimental user interface management system, or UIMS, that can create graphical, highly interactive user interfaces. A previous article[1] presented an overview of Peridot, concentrating on how the static displays (the presentation) of the user interfaces are created. This article describes how the dynamics of the user interface can be specified by demonstration. The full description of Peridot and the research that led to its development is also available.[2] Peridot, which

*The author is now with Carnegie Mellon University.

0272-1716/87/0900-0051$01.00 © 1987 IEEE

stands for programming by example for real-time interface design obviating typing, is implemented in Interlisp-D on a Xerox DandeTiger (1109) workstation.

The central approach of Peridot is to allow the designer of the user interface to design and implement direct-manipulation user interfaces[3,4] in a direct-manipulation manner. The designer need not do any programming in the conventional sense, since all commands and actions are given graphically. The general strategy of Peridot is to allow the designer to draw the screen display that the end user will see, and then to perform actions just as the end user would—for example, by moving a mouse, pressing its buttons, turning a knob, or toggling a switch. The results are immediately visible and executable on the screen and can be edited easily. The designer gives examples of typical values for parameters and actions, and Peridot automatically guesses (or infers) how they should be used in the general case.

Because any inferencing system will occasionally guess wrong, Peridot uses three strategies to ensure correct inferences. First, Peridot always asks the designer if guesses are correct. Second, the results of the inferences can be seen and executed immediately. Finally, the inferences can be undone if they are wrong. The interface can be edited easily, and the changes will be visible immediately. In addition, Peridot creates efficient code so that the final interface can be used in actual application programs.

As shown in a previous paper,[1] this technique allows the presentation aspects of the interface to be created by nonprogrammers in a very natural manner. Peridot may even be simple enough for end users to modify their user interfaces with. This article describes how these ideas have been extended to allow the dynamics of the interaction to be programmed by demonstration, which is harder because of the dynamic and temporal nature of the interactions.

To control the dynamics, all parts of the interaction that can change at runtime are attached to active values. These are like variables except that the associated picture is updated immediately when the value changes. Input devices and application programs can set active values at any time to modify the picture. Active values also form the link between the application program and the user interface.

Throughout this article the term "designer" is used for the person creating user interfaces (and therefore using Peridot). "User," or "end user," means the person using the interface created by the designer.

Background and related work

Because programming user interfaces is difficult and expensive, there has been a growing effort to create tools, called user interface management systems,[5-7] to help with the task. Many early (and some current) UIMSs require the designer to specify the interfaces in a textual, formal programming-style language. This procedure proved useful and appropriate for textual command languages[8] but difficult and clumsy for graphical, direct-manipulation interfaces,[9] and designers have been reluctant to use it.[10] Therefore, a number of UIMSs allow the designer to use more graphical styles. Examples include Menulay,[11] Trillium,[12] and GRINS.[13] These are, for the most part, still limited to using graphical techniques for specifying the placement of pieces of the picture and interaction techniques (for example, where menus are located and what type of light button to place where). Some systems, such as Squeak,[14] allow interaction techniques to be specified textually; but as far as we know, no previous system attempts to allow the dynamics of the actual input devices and the interaction techniques themselves to be programmed in a graphical, nontextual manner.

In trying a new approach to these problems, Peridot uses techniques from visual programming and example-based programming with plausible inferencing.[15] "Visual programming" refers to systems that allow the specification of programs using graphics. Some of these systems, such as Rehearsal World,[16] have succeeded in making programs more visible and understandable and therefore easier for novices to create.

Example-based programming systems allow the programmer to use examples of input and output data during programming. Some of these systems use "plausible inferencing," which means that they try to guess generalizations or explanations from specific examples.[17] Systems of this type are generally called "programming by example."[15] Some systems that allow the programmer to develop programs using specific examples do not use inferencing.[18-20] For example, SmallStar[18] allows users to write programs for the Xerox Star office workstation by simply performing the normal commands and adding control flow afterward. Peridot uses inferencing to try to make some of the complex parts of interface design automatic, such as specifying the control flow.

Another important component of Peridot is *constraints*, which are relationships among objects and data that must hold even when the objects are manipulated. Peridot uses two kinds of constraints. *Graphical constraints*, the kind used in ThingLab[21] and related systems,[22,23] relate one graphic object to another. *Data constraints* ensure that a graphical object has a particular relationship to a data value; these are used in the Process Visualization System,[24] which was influenced by "triggers" and "alerters" in database management systems.[25] They are also similar to the "control" values in GRINS[13] except that they are programmed by example instead of textually and can be executed immediately without waiting for compilation.

In Peridot, data constraints are associated with active values, which have been used in artificial intelligence simulation environments.[26,27]

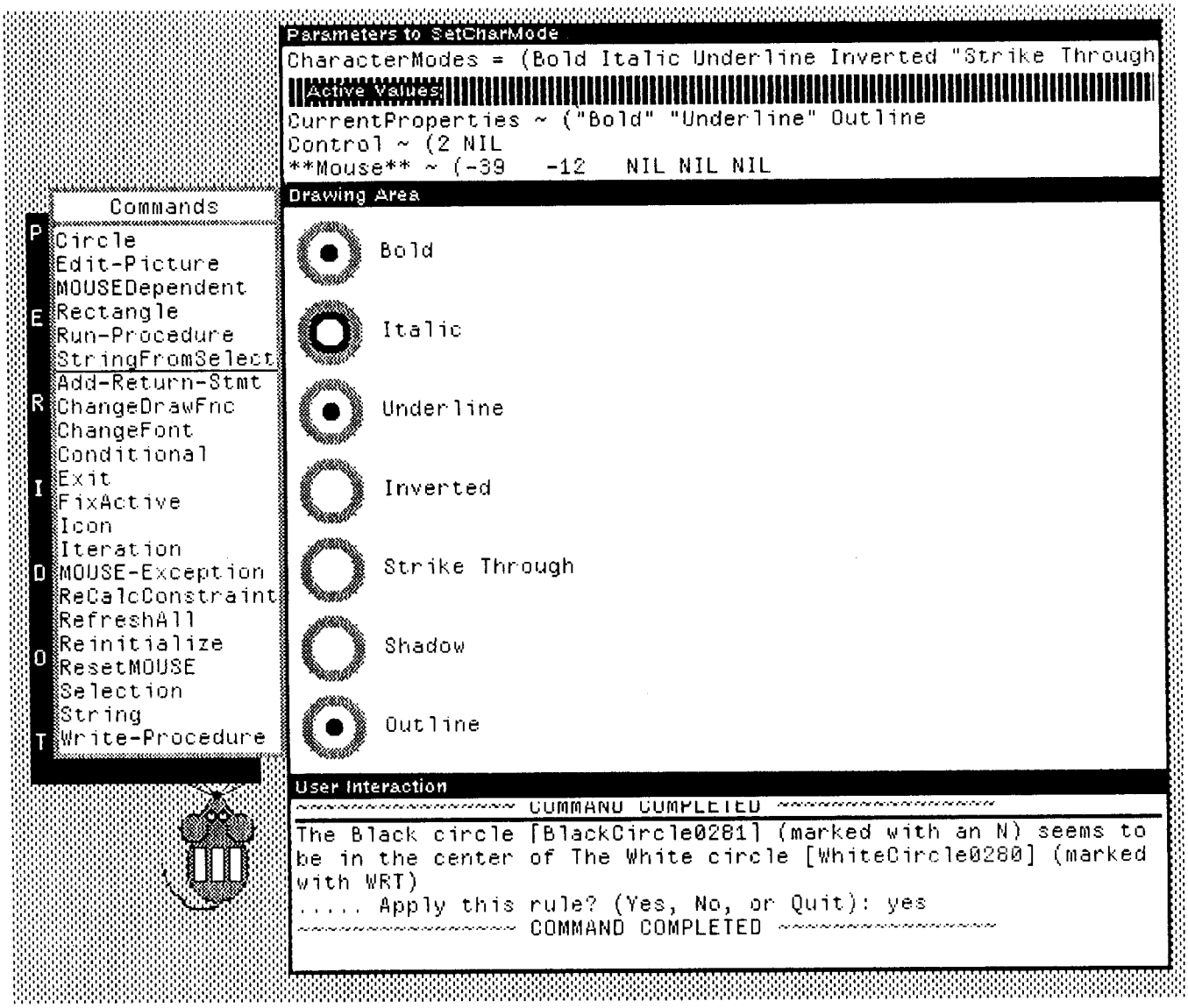


Figure 1. The three Peridot windows (the parameter window at the top is divided into two parts) and the Peridot command menu (left).

Peridot in action

A concrete example is the best way to demonstrate how easy creating a user interface is with Peridot. Space limitations require that some of the details be left out, but further explanations of the process appear in subsequent sections and in other articles.[1,2]

When creating a procedure by demonstration using Peridot, the designer first types in the parameters to the procedure, any active values needed, and an example of a typical value for each. Peridot then creates three windows and a menu and puts the parameters and active values in the upper window (see Figure 1). The menu, at the left, is used to give commands to Peridot. The window in the center shows what the user will see as a result of this procedure (the end-user interface), and the window at the bottom is used for messages and prompts.

Figure 2 shows the steps for creating a scroll bar that displays both the part of a file that is visible in a window and the percentage of the file visible. First, in (a) the background graphics are created. In (b) the designer creates a grey bar that is as tall as the surrounding rectangle. This will be used to indicate that the entire file is visible, and the designer gives a Peridot command to have the height remembered. Then in (c) the designer makes the bar two pixels high and uses the same command to tell Peridot that this height is the other extreme. Peridot prompts for the active value that the height change should depend on (*ScrollPercent* in this case) and then asks the designer for the values that correspond to the two graphical extremes

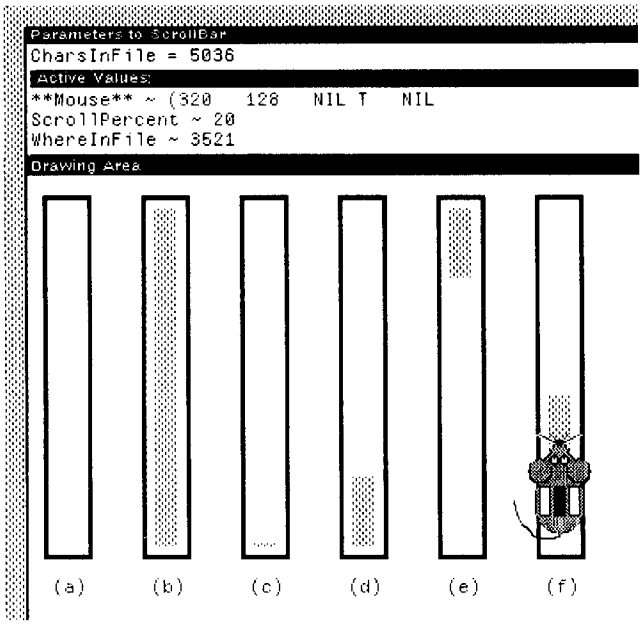


Figure 2. Steps during the creation of a scroll bar using Peridot. In (a) the background graphics have been created. The grey bar will represent percentage of file visible in the window. The two extremes of the full file (b) and none of the file (c) are demonstrated. The height of the bar will depend on the active value *ScrollPercent*, which ranges from 100 to 0. Next, two other extremes are demonstrated—the end of the file visible (d) and the beginning of the file visible (e). The active value *WhereInFile* controls this indicator. The designer then uses the simulated mouse (f) to demonstrate that the bar should follow the mouse when the middle button is down.

(here, 100 and 0). Peridot then automatically creates a linear interpolation that modifies the height of the bar on the basis of the value of *ScrollPercent*, as shown in (d). Similarly, the designer moves the grey box to the bottom of the bar (d) and then to the top (e) and specifies that this corresponds to the active value *WhereInFile* showing the position in the file. When asked, the designer specifies that *WhereInFile* varies from the value of the parameter *CharsInFile* down to 1. These two active values can then be set independently or at the same time by an application.

Next, the designer moves the simulated mouse (which represents the real mouse) over the grey box and presses the middle button (Figure 2f). Since the box has already been defined to move in *y* with an active value, Peridot infers that the mouse should control this action while its middle button is down. Of course, for this and all other inferences, the designer is queried to ensure that the guess is correct. If it is not, Peridot investigates other possibilities. When the mouse is used to update the graphics, the active values are also set and an application will be notified if appropriate. Now this piece of the interaction can be executed immediately with either the real or simulated mouse.

Overview

All UIMSs are restricted in the forms of user interface they can generate.[28] Peridot is aimed only at graphical, highly interactive interfaces that do not use the keyboard. It is clear, however, that Peridot will not be able to create every possible mouse-based type of interaction. Nevertheless, it does have sufficient coverage to create interfaces like those of the Apple Macintosh[29] as well as some entirely new interfaces, and it is much easier to create these interfaces using Peridot than with other existing methods.

Peridot tries to let the designer specify the input device actions mostly by demonstration. The goal is to let the designer simply move the devices the same way the end user would, and Peridot will create the code to handle the actions. For this to work, the system must infer how the specific actions on the example data should be generalized to handle any appropriate end-user data. In addition, exceptions and error cases must be handled.

An important consideration for any demonstrational system is how much should be done by demonstration and how much by conventional specification. It is usually much easier to implement the specification technique in UIMSs, and in some cases demonstration may actually be harder for the designer to use. This happens when the designer knows how the system should act and believes it would be much easier simply to specify the actions than laboriously demonstrate them. For example, to demonstrate by example whether an action should *toggle*, *set*, or *clear* a value, the designer must demonstrate the action twice. The first demonstration, over a set value, will cause the value to be cleared for the function *toggle*, stay set for *set*, and be cleared for *clear*. The second demonstration, over a cleared value, will cause the value to be set for the function *toggle*, be cleared for *set*, and stay cleared for *clear*. To specify which should happen, the designer need only choose *toggle*, *set*, or *clear*, which will probably be much quicker. In other cases, however, the number of possible choices is so large that it would be more difficult to use specification. This has been the case for most aspects of the presentation of user interfaces (the static pictures).[1]

To make Peridot as easy to use as possible, the specification method is allowed whenever there is a small number of easily delineated choices. Demonstration is considered the primary method, however, since it is more novel and difficult to provide, and thus more interesting in a research context. Demonstrational methods are more difficult for the dynamic interactions than for static pictures, since issues of *when* operations should happen are involved (not just *what* should hap-

pen), and the ephemeral nature of the actions makes it harder to select the ones to which operations apply.

Active values

The key to easy specification of the way input devices are handled is to provide appropriate communication mechanisms between them and the graphics displays they manipulate. Peridot uses *active values* for this control, and they have proved powerful, efficient to implement, and easy for the designer to use. Active values are also used to connect the user interfaces with application programs.

Active values are like variables in that they can be accessed and set by any program or input device. They can have arbitrary values of any type. Whenever they are set, all objects that depend on them are immediately updated. The user interface designer can create as many active values as needed and give them arbitrary names. Typically, each part of the interface that can change at runtime will be controlled by an active value, as shown in Figure 2, where *ScrollPercent* and *WhereInFile* vary continuously in a specified range.

Different kinds of control using active values are shown in Figures 1 and 3. In Figure 1 the active value *CurrentProperties* contains a list of the names that are designated by a dot. In Figure 3 seven active values control a window that can scroll vertically or horizontally, move, or change size.

An important advantage of active values is that they allow the application to deal in its own units (0 to 100 and 1 to *CharsInFile* in Figure 2, and the string names of the font properties in Figure 1) and remain totally independent of how these values are represented graphically or how they are set by input devices. The graphics can be changed arbitrarily, and the application code is not affected.

Exceptional values

An important consideration is what to do when an active value is set outside its expected limits. This is obviously most important when the active value is set by an input device, but it can also have a hand in preventing application programs from setting values incorrectly. An application can supply a procedure that will support gridding and more complex types of semantic feedback (where the application must be involved in the inner feedback loop). Alternatively, one of Peridot's built-in range-checking routines can be used. The designer chooses what to do when the value is out of range:

1. Raise an error exception.
2. Peg the value to the nearest legal value (MIN or MAX).
3. Wrap the value around to the other extreme (MOD).
4. Allow the value to go outside the range.

Peridot lets the designer explicitly specify what happens

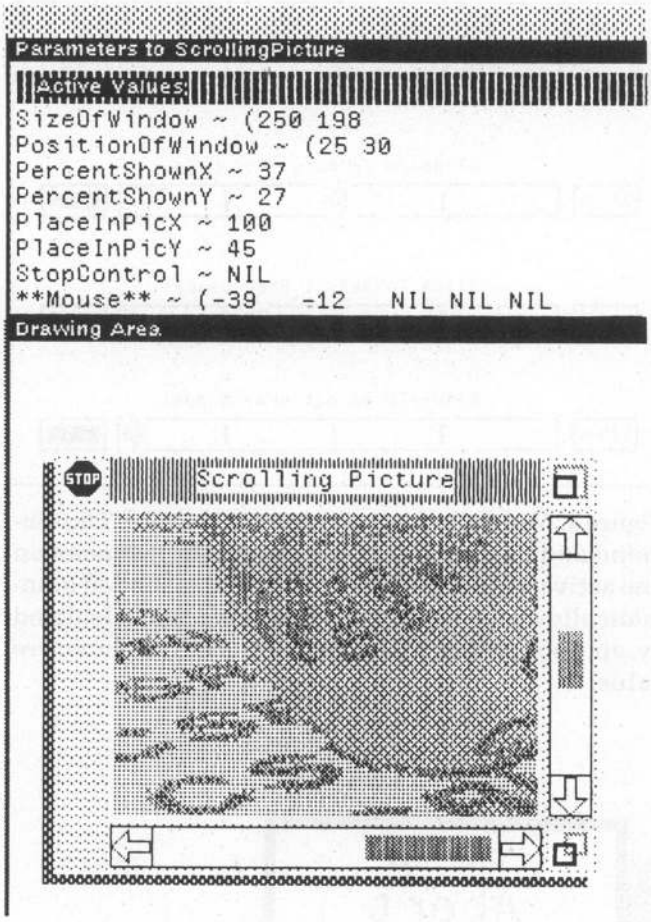


Figure 3. A complex interface created entirely by Peridot. The window can move or change size, and the picture can scroll either vertically or horizontally. This is controlled by seven active values. An application procedure is called to display the picture and calculate the percentage displayed in the window, but all other manipulations are handled by Peridot and were defined by demonstration.

(the default is "allow") and, in some cases, automatically infers the constraint.

Application notification

Another important consideration is when to notify an application program if an active value changes. This comes into play mainly when the value is changed by input devices, but an application procedure can also be used to tie certain active values together to provide semantic feedback. As an example, Figure 4 shows a graphical potentiometer for setting grey shades. The end user can move the diamond with the mouse. The position of the diamond and the number in the left box are tied directly to the active value *SliderValue* using linear interpolation, but the halftone representation of the cor-

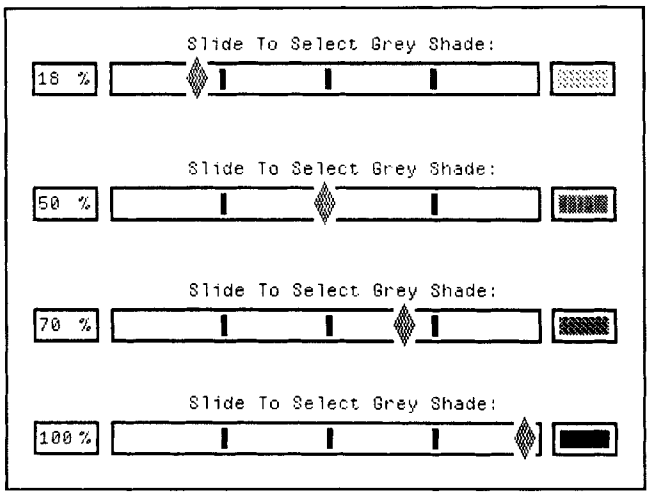


Figure 4. Multiple views of a graphical slider. The diamond and the numerical percentage (left) depend on one active value. The box on the right is shaded automatically on the basis of the halftone color calculated by an application procedure attached to the active value.

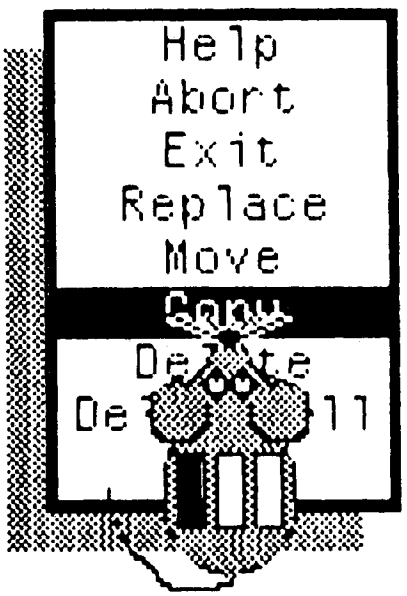


Figure 5. The simulated mouse with its left button down is being used to program a menu of strings by demonstration. The black rectangle (now over *Copy*) will follow the mouse while the left button is held down.

responding grey shade is calculated using an application-provided procedure. The conversion function is called whenever the *SliderValue* value changes, so the color in the box on the right will always be correct.

Note that this allows the application program to have fine-grain control over the interface. Most other UIMSs provide only coarse-grain control, so they cannot handle this type of semantic feedback. The application can control feedback, default values, and error detection and recovery at a low level, and active values are efficient enough to allow application procedures to be called for every increment during mouse tracking or other input device handling.

Peridot provides several options for when an application is notified:

1. Whenever the value is set (including when it is set to its existing value); this is useful as a trigger.
2. Whenever the value changes.
3. Whenever the value changes by more than some threshold.
4. When an interaction is complete (for example, when the mouse button is released after moving the diamond in Figure 4).
5. Never.

These choices are specified explicitly. The threshold choice (number 3) is useful for increasing efficiency (so that the application is not notified too often), and it is useful for controlling animations using the system-provided active value for the clock (for example, blinking or moving at a specific speed).

Application procedures attached to active values are also used to extend the operations that Peridot supports. If some kind of interaction or special effect is not provided, then a very short procedure can usually be written to perform the action by querying and setting active values.

The implementation of active values is very efficient[2] (the affected objects are computed at design time) and can be optimized for whatever operating system is in use. They do not require any complex constraint satisfaction techniques or much more computation than would be needed if the various actions were coded by hand.

Input devices

Each input device is attached to its own active value. For example, the mouse has an active value called *Mouse*, which is a list of five items: the x position of the mouse, the y position, and a Boolean for each of the three buttons.* A button box would be represented as a set of Booleans—one for each button.

*Of course, some systems may provide more or fewer items for the mouse. The connection between the hardware devices and their active values is written in conventional Lisp code.

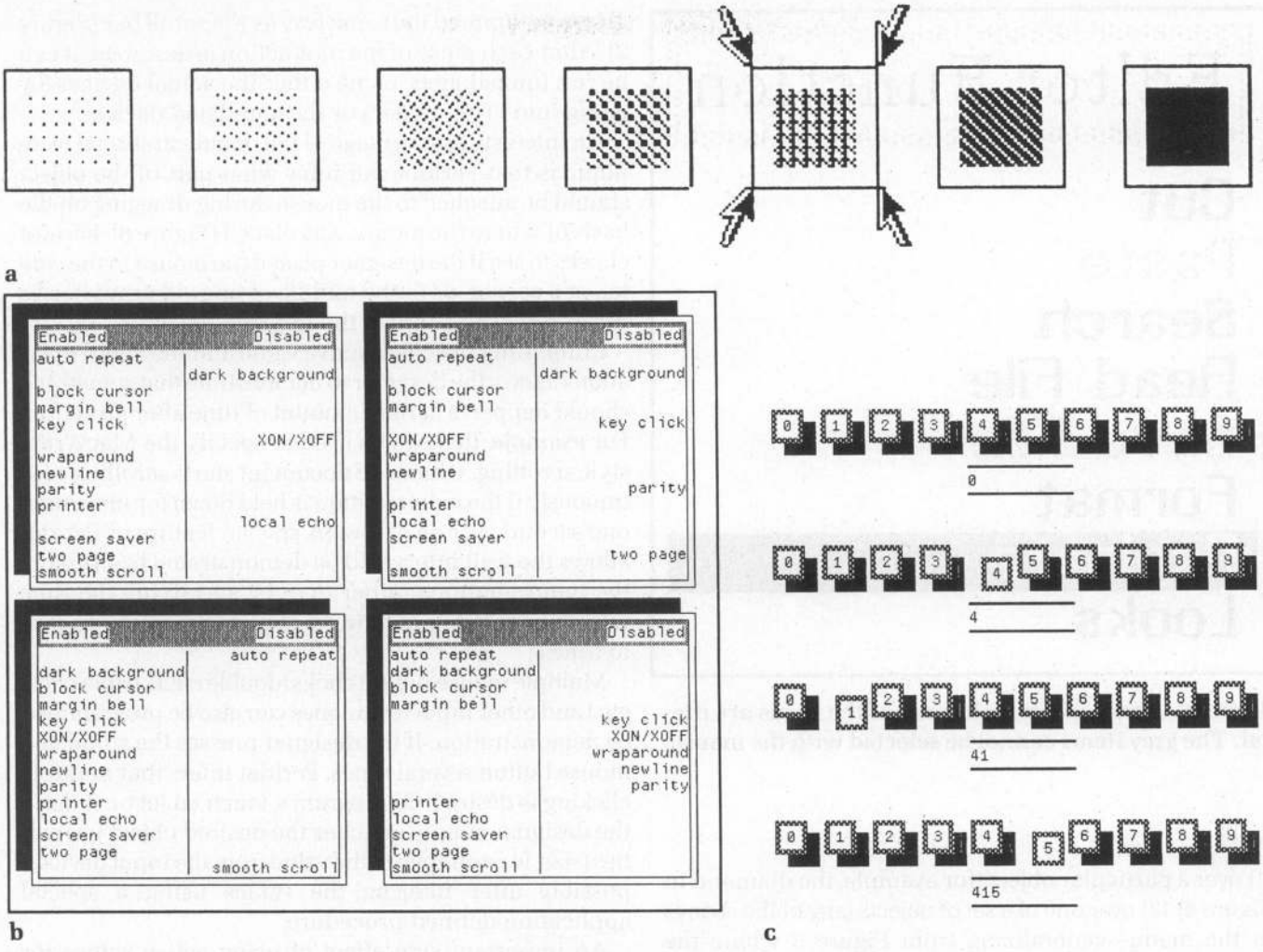


Figure 6. The response to the mouse action is limited only by the creativity of the designer. In (a) four arrows move with the mouse; in (b) text items move left and right; and in (c) number-pad buttons pretend to move in three dimensions.

Clearly, the mechanisms described in the previous section can be used to attach the input devices' active values to active values controlling the graphics. The techniques described under the subsection "exceptional values" are used to restrict the values to certain limits, and the application will be notified when appropriate.

If this were all that was provided, however, then code would have to be written for each mouse dependency to cover all the requirements. The main problem is that interaction techniques need to be activated only under certain conditions. For example, a typical menu has a black rectangle that follows the mouse (Figure 5), but only while the mouse button is held down over the menu. When the mouse button is released, the current value is returned.

When specifying interactions of this type, Peridot uses a postfix-style sequence. First, the designer creates the graphics that should appear (the black rectangle in the menu, for example) and then specifies that it should depend on the mouse. The actual graphics that respond to the mouse actions are totally under the control of the designer. For example, in Figure 6a the four arrows move with the mouse; in Figure 6b the text items move left and right when the mouse button is pressed over them; and in Figure 6c the numbers appear to move in three dimensions.

The simulated mouse[1] is used to show what should happen, since the real mouse is used for giving Peridot commands. For the menu, the designer moves the simulated mouse over the black rectangle and shows the left button down. Peridot then confirms that the action should happen on left button down. On the basis of the position of the simulated mouse, Peridot next determines whether the action should happen when the mouse is

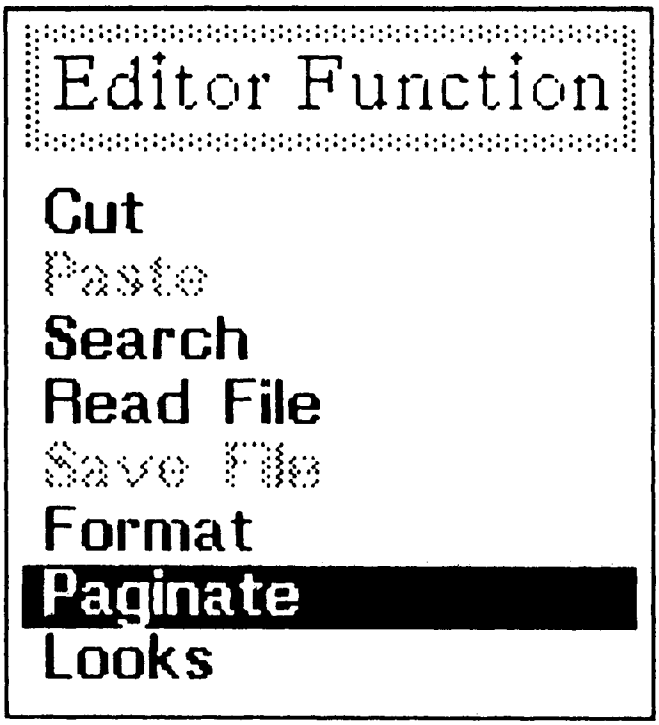


Figure 7. A menu in which some of the items are illegal. The grey items cannot be selected with the mouse.

(1) over a particular object (for example, the diamond in Figure 4), (2) over one of a set of objects (any of the strings in the menu—generalizing from Figure 5 where the mouse is over a particular item: *Copy*), or (3) anywhere on the screen. If the simulated button is down, Peridot assumes that the operation should happen continuously while the button is pressed. If the simulated button is pressed and released, the action will happen once when the button goes down. It is also possible to demonstrate that the action should happen once when the button is released, continuously while the button is up, or only after the mouse button has been pressed several times (for example, a double click).

Exception areas, where the interaction is not allowed, can be defined by demonstration. In Figure 7, for example, the black rectangle will not go over any of the names shaded in grey. Of course, the graphic presentation of the illegal items is determined totally by the designer and is independent of the exception mechanism. The value to use for the active value when the mouse is over an exception item, as well as the value used when the mouse goes outside the object's boundaries, can be specified by the designer.

The property-sheet interaction (Figure 1) is demonstrated much like the menu. The example value for the controlling active value is used to determine whether multiple items are allowed (as for the property sheet) or only one is allowed (as for the menu). The slider (Figure 4) is programmed the same way as the scroll bar (Figure 2). After each piece of the interaction is designed, it can be run immediately, using either the actual devices (by going into "run mode") or the simulated devices.

An interesting advantage of the demonstrational technique is that Peridot can infer what part of the object should be attached to the mouse during dragging on the basis of where the mouse was placed (Figure 8). Peridot checks to see if the designer placed the mouse in the center, at a corner, or in the middle of one side, and it asks the designer to confirm the inferred position.

Combining the clock active value and the above operations allows the designer to demonstrate that something should happen a certain amount of time after an action. For example, this can be used to specify the MacWrite-style scrolling, where the document starts scrolling continuously if the mouse button is held down for more than one second over an arrow. (A special feature of Peridot allows the wait interval to be demonstrated by pressing the mouse buttons rather than by specifying the time numerically, thus providing a demonstrational interface to time.)

Multiple mouse-button clicks (double click, triple click, etc.) and other input techniques can also be programmed by demonstration. If the designer presses the simulated mouse button several times, Peridot infers that multiple clicking is desired. To program a touch tablet or slider,[30] the designer simply attaches the desired object properties (size, for example) to the value from the input devices, possibly after filtering the values using a special application-defined procedure.

An important side effect of using active values for creating interactions is that multiple input devices operating in parallel[30] can be handled easily, whereas they are very difficult to implement in conventional systems. For example, the designer can easily tie the position of an object to the mouse, and its size to a knob operated with the other hand—allowing both to operate concurrently. In addition, it requires no extra effort to have multiple interactions that use the same device (such as multiple mouse menus) available to the end user at the same time, since Peridot ensures that all activated techniques are watching for their appropriate input.

Editing interactions

Editing static pictures is very easy, since they can be selected and respecified easily. Selecting dynamic and ephemeral things such as interactions is harder, however, because they typically do not have visual representations on the screen. Some systems have required the user to learn a textual representation for the actions to allow editing,[18] but this is undesirable.

Peridot allows interactions to be edited several ways. First, an interaction can be re-demonstrated, and Peridot will ask whether the new interaction should replace the old one or run in parallel. Since individual interactions

are small, this should not be a large burden. A complex interaction, such as a menu or scroll bar, typically is constructed from a number of small interactions, each of which takes only a few seconds to define. The second way to edit interactions is to select an active value and request that the interactions affecting it be removed.

Evaluation

It is very difficult to quantify formally the range of user interfaces that Peridot can create, since there is no comprehensive taxonomy of interaction techniques. Informally, Peridot's range can be described by example: It can create menus of almost any form (with single or multiple items selected), property sheets, light buttons, radio buttons, scroll bars, two-dimensional scroll boxes, percent-done progress indicators, graphical potentiometers, sliders, iconic and title-line controls for windows, dynamic bar charts, and many other interfaces. Thus, Peridot can create almost all of the Apple Macintosh interface, as well as many new interfaces such as those using multiple input devices concurrently. Peridot also created its own user interface. The ideas in Peridot could be extended easily to handle the keyboard and other types of input devices.

To evaluate how easy Peridot is to use, 10 people used the system for about two hours each. Five were experienced programmers and five were nonprogrammers with some experience using a mouse. The results of this experiment were very encouraging. After about an hour and a half of guided use, the subjects were able to create a menu of their own design unassisted. This demonstrates that one basic goal of Peridot has been met: Nonprogrammers can use it to create user interfaces.

In addition, programmers will appreciate using Peridot to define graphical parts of user interfaces because it is so much faster and more natural than conventional programming. As a small, informal experiment, six expert programmers implemented a particular menu using their favorite hardware and software environments. Some wrote the menu by hand and others modified existing code. With Peridot, the time needed to create the menu ranged from 4 to 15 minutes, but programming took between 50 and 500 minutes.[2] Thus, using Peridot appears to be significantly faster.

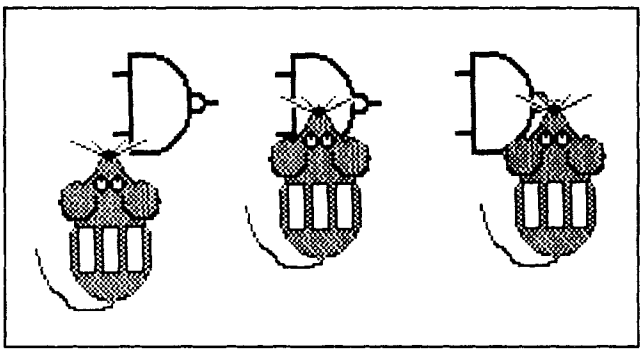


Figure 8. An object might be attached to the mouse in various places for dragging: bottom-left, center, or center of right side.

Conclusions

Peridot successfully demonstrates that it is possible to program a large variety of mouse and other input-device interactions by demonstration. The use of active values supports multiprocessing and makes the linking to application programs straightforward, fast, and natural; and it supports semantic feedback easily. Interfaces created with Peridot can be tried out immediately (with or without the application program), and the code generated is efficient enough to be used in actual end applications. This allows extremely rapid prototyping of direct-manipulation interfaces.

By providing the ability to use explicit specification and demonstrational methods, Peridot allows the designer to use the most appropriate techniques for creating the user interfaces. The novel use of demonstrational (programming-by-example) methods makes a large class of previously hard-to-create interaction techniques easy to design, implement, and modify. In addition, Peridot makes it easy to investigate many new techniques that have never been used before and in this way may help designers discover the next generation of exciting user interfaces. ■

Acknowledgments

First, I want to thank Xerox Canada, Inc., for the donation of the Xerox workstations and Interlisp environment. This research was also partially funded by the National Science and Engineering Research Council of Canada. For help and support with this article, I would especially like to thank my advisor, Bill Buxton, and also Bernita Myers, Peter Rowley, and Ron Baecker.

References

1. B.A. Myers and W. Buxton, "Creating Highly Interactive and Graphical User Interfaces by Demonstration," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 249-258.
2. B.A. Myers, *Creating User Interfaces by Demonstration*, doctoral dissertation, Dept. of Computer Science, Univ. of Toronto. Available as Tech. Report CSRI-196, Computer Systems Research Inst. Technical Reports, Univ. of Toronto, Ontario, Canada, M5S 1A1, May 1987.

3. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, Aug. 1983, pp. 57-69.

4. E.L. Hutchins, J.D. Hollan, and D.A. Norman, "Direct Manipulation Interfaces," in *User Centered System Design*, D.A. Norman and S.W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, N.J., 1986, pp. 87-124.

5. "Graphical Input Interaction Technique (GIIT) Workshop Summary," in *Computer Graphics*, J.J. Thomas and G. Hamlin, eds., ACM SIGGRAPH, Jan. 1983, pp. 5-30.

6. D.R. Olsen, Jr., et al., "A Context for User Interface Management," *CG&A*, Dec. 1984, pp. 33-42.

7. *User Interface Management Systems*, G.R. Pfaff, ed., Springer-Verlag, Berlin, 1985.

8. R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming," *Computer*, Aug. 1985, pp. 51-59.

9. B. Shneiderman, "Seven Plus or Minus Two Central Issues in Human-Computer Interfaces," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 343-349.

10. D.R. Olsen, Jr., "Larger Issues in User Interface Management," *Proc. ACM SIGGRAPH Workshop on Software Tools for User Interface Development*, reprinted in *Computer Graphics*, Apr. 1987, pp. 134-137.

11. W. Buxton et al., "Towards a Comprehensive User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 35-42.

12. D.A. Henderson, Jr., "The Trillium User Interface Design Environment," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 221-227.

13. D.R. Olsen, Jr., E.P. Dempsey, and R. Rogge, "Input-Output Linkage in a User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 225-234.

14. L. Cardelli and R. Pike, "Squeak: A Language for Communicating with Mice," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 199-204.

15. B.A. Myers, "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 59-66.

16. L. Gould and W. Finzer, "Programming by Rehearsal," Tech. Report SCL-84-1, Xerox Palo Alto Research Center, May 1984. A short version appears in *Byte*, June 1984.

17. A.W. Biermann, "Approaches to Automatic Programming," in *Advances in Computers*, Vol. 15, M. Rubinoff and M.C. Yovitz, eds., Academic Press, New York, 1976, pp. 1-63.

18. D.C. Halbert, *Programming by Example*, doctoral dissertation, Computer Science Division, Dept. of EE & CS, Univ. of California, Berkeley, 1984. Also available as Tech. Report TR OSD-T8402, Xerox Office Systems Division, Systems Development Dept., Dec. 1984.

19. H. Lieberman, "Constructing Graphical User Interfaces by Example," *Graphics Interface 82*, Canadian Information Processing Soc., Toronto, Ontario, 1982, pp. 295-302.

20. D.C. Smith, *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*, Birkhauser, Basel, Switzerland, 1977.

21. A. Borning, "Thinglab—A Constraint-Oriented Simulation Laboratory," Tech. Report SSL-79-3, Xerox Palo Alto Research Center, 1979.

22. R.A. Duisberg, "Animated Graphical Interfaces," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 131-136.

23. G. Nelson, "Juno, a Constraint-Based Graphics System," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 235-243.

24. J.D. Foley and C.F. McMath, "Dynamic Process Visualization," *CG&A*, Mar. 1986, pp. 16-25.

25. O.P. Buneman and E.K. Clemons, "Efficiently Monitoring Relational Databases," *ACM Trans. Database Systems*, Sept. 1979, pp. 368-382.

26. C.V. Ramamoorthy, S. Shekhar, and V. Garg, "Software Development Support for AI Programs," *Computer*, Jan. 1987, pp. 30-40.

27. M. Stefik, D.G. Bobrow, and K.M. Kahn, "Integrating Access-Oriented Programming into a Multi-Paradigm Environment," *IEEE Software*, Jan. 1986, pp. 10-18.

28. P.P. Tanner and W.A.S. Buxton, "Some Issues in Future User Interface Management System (UIMS) Development," in *User Interface Management Systems*, G.R. Pfaff, ed., Springer-Verlag, Berlin, 1985, pp. 67-79.

29. G. Williams, "The Apple Macintosh Computer," *Byte*, Feb. 1984, pp. 30-54.

30. W. Buxton and B. Myers, "A Study in Two-Handed Input," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 321-326.

Brad Myers is a research computer scientist at Carnegie Mellon University. From 1980 until 1983 he worked at PERQ Systems Corporation, where he designed and implemented the Sapphire window manager and numerous PERQ demonstrations for the SIGGRAPH equipment exhibition. His research interests include user interface management systems, user interfaces, programming by example, visual programming, interaction techniques, window management, programming environments, debugging, and graphics.

Myers recently completed a PhD in computer science at the University of Toronto. He received the MS and BS degrees from the Massachusetts Institute of Technology while he was a research intern at Xerox PARC. He is a member of SIGGRAPH, SIGCHI, ACM, and the Computer Society of the IEEE.

Myers' address is Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3890.