Creating Dynamic Interaction Techniques by Demonstration

Brad A. Myers

Dynamic Graphics Project
Computer Systems Research Institute
University of Toronto
Toronto, Ontario, M5S 1A4, Canada

ABSTRACT

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 user interface designer 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 immediately be executed and edited. This promotes extremely rapid prototyping since it is very easy to design, implement and modify mouse-based interfaces. Peridot also supports additional input devices such as touch tablets, as well as multiple input devices operating in parallel (such as one in each hand) in a natural, easy to specify manner. This is 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.

CR Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*User Interfaces*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Synthesis*; I.3.6 [**Computer Graphics**]: Methodology and Techniques.

General Terms: Human Factors

Additional Key Words and Phrases: Interaction Techniques, Programming by Example, Visual Programming, User Interface Design, User Interface Management Systems, Direct Manipulation.

1. Introduction

Peridot is an experimental User Interface Management System (UIMS) that can create graphical, highly interactive user interfaces. A previous paper [16] presented an overview of Peridot concentrating on how the static displays (the presentation) of the user interfaces are created. This paper describes how the dynamics of the user interface can be specified by demonstration. Peridot, which stands for Programming by Example for Real-time Interface Design Obviating Typing, is implemented in Interlisp-D [30] on a Xerox DandeTiger (1109) workstation.

The central approach of Peridot is to allow the user interface designer to design and implement Direct Manipulation user interfaces [24] [12] in a Direct Manipulation manner. The designer does not need to 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, such as moving a mouse or pressing its buttons or hitting keyboard keys. 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*) the general case. Because any inferencing system will occasionally guess wrong, Peridot uses two strategies to insure correct inferences. First, Peridot always asks the designer if guesses are correct, and second, the results of the inferences can be immediately seen and executed. The interface can be easily edited 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 [16], this technique allows the presentation aspects of the interface to be created by non-programmers in a very natural manner. Peridot may even be simple enough so that end-users can use it to modify their user interfaces. This paper describes how these ideas have been extended to allow the dynamics of the interaction to be programmed by demonstration, which is a harder problem due to the dynamic and temporal nature of the interactions.

To control the dynamics, all parts of the interaction that can change at run-time are attached to *active values* which 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 paper, the term "designer" is used for the person creating user interfaces (and therefore using Peridot). The term "user" (or "end user") is reserved for the person using the interface created by the designer. A longer report providing more detail and covering other aspects of Peridot is in preparation [17].

2. Background and Related Work

It has long been realized that programming user interfaces is a difficult and expensive task, and there has been a growing effort to create tools, called User Interface Management Systems (UIMSs) [28] [20] [23], to help create them. Many early (and some current) UIMSs require the designer to specify the interfaces in a textual, formal programming-style language. This proved useful and appropriate for textual command languages [13] but difficult and clumsy for graphical, Direct Manipulation interfaces [25], and designers have been reluctant to use it [22]. Therefore, a number of UIMSs allow the designer to use more graphical styles. Examples of this include Menulay [4], Trillium [11], and GRINS [21]. These are, for the most part, still limited to using graphical techniques for specifying the *placement* of pieces of the picture and interaction techniques (e.g., where menus are located and what type of light button to place where). Some systems, such as Squeak [6], allow interaction techniques to be specified textually, but no system that I am aware of attempts to allow the dynamics of the actual input devices and the interaction techniques themselves to be programmed in a graphical, non-textual manner.

In order to try a new approach to these problems, Peridot uses techniques from Visual Programming and Programming by Example [15]. "Visual Programming" refers to systems that allow the specification of programs using graphics. "Programming by Example" systems attempt to infer programs from examples of the data that the program should process [1]. Some systems that allow the programmer to develop programs using specific examples do not use inferencing [10] [14] [26]. For example, SmallStar [10] allows users to write programs for the Xerox Star office workstation by simply performing the normal commands and adding control flow afterwards. Visual Programming systems, such as Rehearsal World [9], have been successful in making programs more visible and understandable and therefore easier to create by novices. Peridot does use inferencing to try to make automatic some of the difficult parts of these systems, such as specifying the control flow.

Active values in Peridot are very much like the binding of data to graphics in the Process Visualization System [8], which was influenced by "triggers "and "alerters" in database management systems [3]. They are also similar to the "Control" values in GRINS [21] except that they are programmed by example rather than textually and can be executed immediately without waiting for compilation.

Peridot was also influenced by graphical constraint systems such as Thinglab [2] and its descendents [7] [19].

3. Sample of Peridot in Action.

The best way to demonstrate how easy it is to create a user interface with Peridot is to work through an example. Due to space limitations, some of the details will be left out, but further explanations of the process are contained in the next sections and in [16] [17].

When creating a procedure by demonstration using Peridot, the designer first types in the parameters to the procedure and 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, which is on 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 that can be used to create a scroll bar that displays both the part of a file that is visible in a window and the percent of the file visible. First, in (a), the background graphics are created. In (b), the designer creates a grey bar the full inside height to represent when the user can see the entire file and gives a Peridot command to have this position remembered. Then, in (c), the designer modifies the height of the bar to be two pixels high, and tells Peridot, using the same command, that this is the other extreme. Peridot prompts for the active value that this should depend on ("ScrollPercent" in this case), and then asks the designer for the values that correspond to the two graphical extremes (here, 100 and 0). Peridot then automatically creates a linear interpolation that modifies the height of the bar based on the value of ScrollPercent, as shown in (d). Similarly, the designer moves the grey box to the bottom of the bar (d) and then 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.

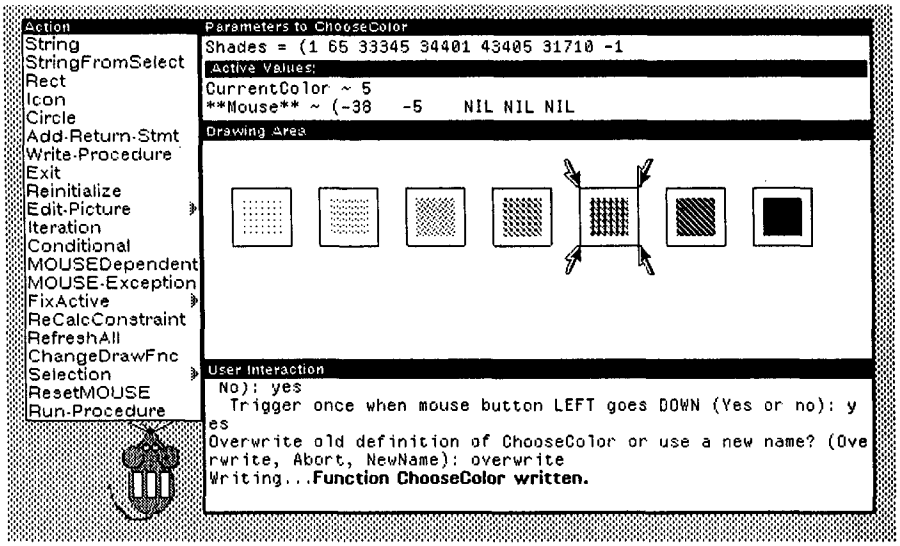

Figure 1.
The three Peridot windows (the parameter window at the top is divided into two parts) and the Peridot command menu (on the left).

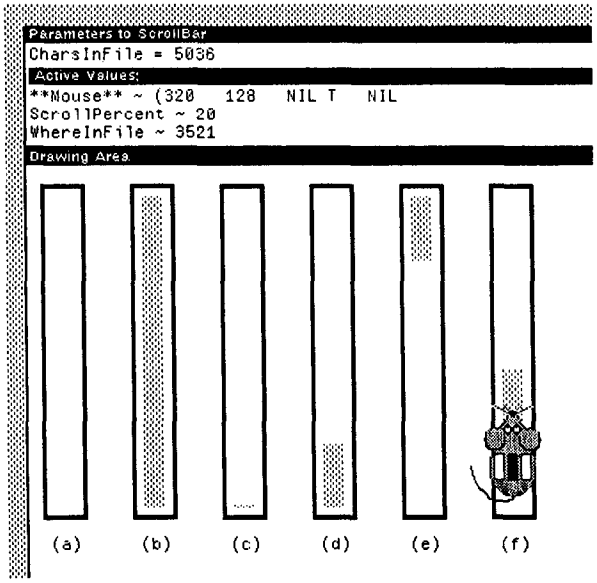

Figure 2.
Steps during the creation of a scroll bar using Peridot. In (a), the background graphics have been created. The grey bar will represent percent of file visible in the window. The two extremes of the full file (b) and none of the file (c) are demonstrated. This will depend on the active value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

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 the mouse middle button is down. Of course, for this and all other inferences, the designer is queried to insure that the guesses are correct. If it were not, Peridot would investigate 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 immediately executed, either with the real or simulated devices.

4. Overview

All UIMSs are restricted in the forms of user interfaces they can generate [27]. Peridot is only aimed at graphical, Direct Manipulation interfaces. It is clear, however, that Peridot will not be able to create every possible mouse-based type of interaction, and it cannot handle text editing or other textual, command-language styles of interfaces. The claim is that Peridot does have sufficient coverage, however, to create interfaces like those of the Apple Macintosh [29] as well as some entirely new interfaces, and that it is much easier to create these interfaces using Peridot than with other existing methods.

Peridot attempts to allow the designer to specify the input device actions mostly by demonstration. The goal is to allow the designer to simply move the devices the same way the end user would, and Peridot will create the code to handle the actions. In order 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, exceptional 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. The problems are that it is usually much easier to implement the specification technique in UIMSs, and in some cases, demonstration may be harder for the designer to use. This happens when the designer knows how the system should act and believes that it would be much easier to simply *specify* the actions rather 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: once over a set value (this will cause the value to be cleared for the function **toggle**, stay set for **set**, and cleared for **clear**) and once over a cleared value (this will cause the value to be set for the function **toggle**, cleared for **set**, and stay cleared for **clear**). To *specify* which should happen only requires the designer to chose **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 proven to be the case for most aspects of the presentation of user interfaces (the static pictures) as described in [16].

In order to make Peridot as easy to use as possible, the specification method is allowed whenever there are a small number of easily delineated choices. The demonstrational method 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 they are for static pictures since issues of *when* operations should happen are involved (not just *what* should happen), and the ephemeral nature of the actions makes it harder to select the ones to which operations apply.

5. Active Values

The key to allowing the handling of the input devices to be easily specified 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 to use for the designer. 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. There will typically be an active value controlling each part of the interface that can change at run-time, as shown in Figure 2, where ScrollPercent and WhereInFile vary continuously in a specified range.

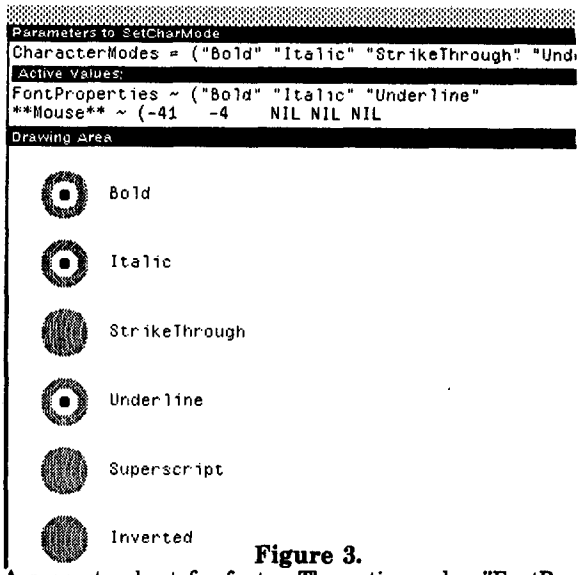

Figure 3.
A property sheet for fonts. The active value "FontProperties" controls which properties are shown as selected.

A different kind of control using active values is shown in Figure 3. Here the active value called "FontProperties" contains a list of the properties to apply to the current font (the list ("Bold" "Italic" "Underline")).

An important advantage of active values is that they allow the application to deal in their own units (0 to 100 and 1 to CharsInFile in Figure 2, and the string names of the font properties in Figure 3), and be totally independent of how these values are represented graphically or set by input devices; the graphics can be changed arbitrarily and the application code is not affected.

5.1. *Exceptional values*

An important consideration is what to do when an active value is set outside of its expected limits. This is obviously most important when the active value is set by an input device, but it can also be useful for preventing application programs from setting values incorrectly. The actions supported by Peridot are:

(1) raising an error exception,

(2) pegging the value to the nearest legal value (MIN or MAX),

(3) wrapping around the value around to the other extreme (MOD),

(4) allowing the value to go outside the range, and

(5) checking using an application-supplied procedure.

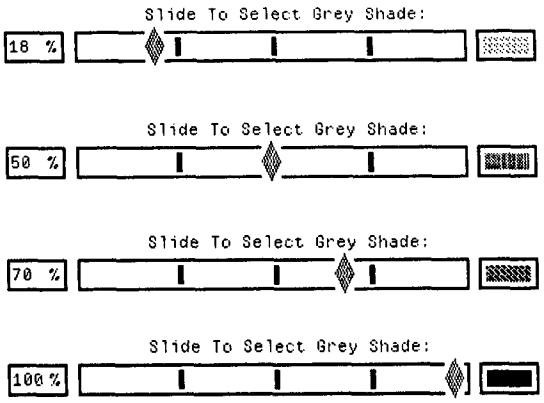

Figure 4.
Multiple views of a graphical slider. The diamond and the numerical percent (on the left) depend on the active value "SliderValue" and show its current value. The box on the right is shaded automatically based on the halftone color returned by an application procedure based on SliderValue.

Peridot allows the designer to explicitly specify what happens (the default is "allow"), and automatically infers the constraint in some cases. The application-supplied procedure (number (5)) is useful for supporting gridding and some types of semantic feedback [18] (where the application must be involved in the inner feedback loop).

5.2. *Application notification*

Another important consideration is when to notify an application program if an active value changes. This is mainly useful when the value is changed by input devices, but it can also be used to tie certain active values together to provide semantic feedback. For 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 directly tied to the active value "SliderValue", and the halftone representation of the corresponding 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.

It is important to emphasize that this allows the application program to have *fine-grain control* over the interface (and not just coarse-grain control as in most UIMSs). The application can control default values, error detection and recovery, and feedback at a low level, and this operates fast enough so it can be used in the inner loops of mouse tracking and other input device handling.

The possible choices for when an application program is notified include:

(1) whenever the value is set (including when it is set to the same value that it already is)—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 (e.g. when the mouse button is released after moving the diamond in Figure 4), and

(5) never.

These are specified explicitly. The threshold choice (number (3)) is useful for increasing efficiency (so the application is not notified too often), and it is useful for controlling animations using the system-provided active value for the clock (e.g. blinking or moving at a specific speed). Active values are also used to extend what Peridot supports. If some kind of interaction or special effect is not provided, then usually a very short procedure can be written to perform the action by querying and setting active values.

The implementation of active values is very efficient (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.

6. Input devices.

Each input device is attached to its own active value. For example, the mouse has an active value 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[1]. A button-box would be represented as a set of booleans—one for each button.

Clearly, the mechanisms described in section 5 can be used to attach the input devices' active values to active values controlling the graphics. The techniques of section 5.1 are used to restrict the values to certain limits and the application will be notified when appropriate (section 5.2).

This is not sufficient, however, to cover all of the requirements for input devices. The main problem is that interaction techniques need to be activated only under certain conditions. For example, a typical menu has an inverting 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.

[1] 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.

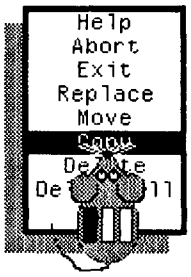

Figure 5.
The "simulated mouse" with its left button down being used to program a menu of strings by demonstration. The inverted rectangle (now over "Copy") will follow the mouse while the left button is held down.

When specifying these types of interactions, Peridot uses a postfix-style sequence (like most Direct Manipulation interfaces). First, the designer creates the graphics that should appear (the black rectangle in the case of the menu), and then specifies that it should depend on the mouse, as shown in the example of section 3. The "simulated mouse" [16] is used for this 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. Based on the position of the simulated mouse, Peridot infers whether the action should happen when the mouse is over a particular object (e.g. the diamond in Figure 4), over one of a set of objects (e.g., any of the strings in the menu—generalizing from the example given where the mouse is over a particular item: here, "Copy"), or anywhere on the screen. Since the simulated button was down, Peridot assumes that the operation should happen *continuously* while the button is pressed. If the simulated button had been pressed and released, the action would happen *once* when the button went down. It is also possible to demonstrate that the action should happen once when the button is released or continuously while the button is up.

Exception areas, where the interaction is not allowed, can be defined by demonstration. For example (Figure 6), the black rectangle will not go over any of the greyed out names. Of course, the graphic presentation of the illegal items is totally up to 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 when the mouse goes outside the object's boundaries, can be specified by the designer.

The property sheet interaction (Figure 3) is demonstrated similarly to 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 scrollbar (Figure 2). After each piece of the interaction is designed, it can be run immediately either using the actual devices (by going into "run mode"), or the simulated devices.

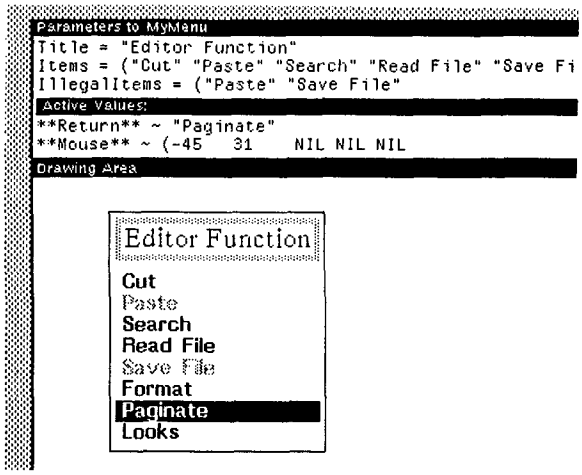

Figure 6.
A menu in which some of the items are illegal. The grey items cannot be selected using the mouse.

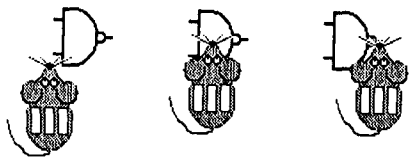

Figure 7.
Demonstrating that an object might be attached to the mouse in various places for dragging: bottom-left, center, and center of right side.

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 based on where the mouse was placed (Figure 7). Peridot checks to see if the designer placed the mouse in the center, corner, or center of one side, and asks the designer for confirmation of the inferred position.

Combining the timer (section 5.2) and the above operations allows the designer to demonstrate that something should happen after a certain period 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[2].

Multiple mouse button clicks (e.g. double-click, triple click, etc.) and other input devices can also be programmed by demonstration. If the designer presses the simulated mouse button multiple times, Peridot infers that multiple clicking is desired. To program a touch tablet or slider [5], the designer simply attaches the desired object properties (e.g. size) to the value from the input devices, possibly after filtering the values using a special application-defined procedure.

[2] A special feature of Peridot allows the amount of time to wait to be demonstrated by pressing on the mouse buttons, rather than specified numerically, to provide a demonstrational interface to time.

An important side effect of using active values for creating interactions is that multiple input devices operating in parallel [5] can be easily handled, whereas this is 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 and have both of these operate concurrently. In addition, it is no extra effort to have multiple interactions that use the same device available to the end user at the same time (such as multiple mouse menus), since Peridot ensures that all activated techniques are watching for their appropriate input.

7. Editing interactions.

It is very easy to edit static pictures since pieces can be easily selected and redrawn. It is harder to select dynamic and ephemeral things such as interactions, however, since they typically do not have visual representations on the screen. Some systems have required the user to learn a textual representation for the actions in order to allow editing [10], but this is undesirable. Therefore Peridot allows interactions to be edited in a number of ways. First, an interaction can be re-demonstrated, and Peridot will inquire if 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 scrollbar, is typically constructed from a number of small interactions, each of which takes only a few seconds to define. Second, all of the interactions that affect an active value can be removed.

8. Current Status

The implementation for Peridot is almost complete and many different interactions have been created using it. Almost all of the Macintosh-style interactions can be programmed by demonstration and Peridot can create most of its own interface.

After the implementation is complete, Peridot will be tested with a number of other user interface designers to make sure that the inferencing works for other people. In addition, the range of types of interfaces that can be created using Peridot will be investigated.

9. 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 makes multi-processing easy and makes the linking to application programs straightforward, fast and natural, and 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, which may help designers discover the next generation of exciting user interfaces.

ACKNOWLEDGEMENTS

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 (NSERC) of Canada. For help and support with this paper, I would like to especially thank my advisor, Bill Buxton, and also Bernita Myers, Peter Rowley, and Ron Baecker.

REFERENCES

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

2. Alan Borning. *Thinglab--A Constraint-Oriented Simulation Laboratory*. Xerox Palo Alto Research Center Technical Report SSL-79-3. July, 1979. 100 pages.

3. O.P. Buneman and E.K. Clemons, "Efficiently Monitoring Relational Databases," *ACM Transactions on Database Systems*. Vol. 4, no. 3. Sept. 1979. pp. 368-382.

4. W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith. "Towards a Comprehensive User Interface Management System," *Computer Graphics: SIGGRAPH'83 Conference Proceedings*. Detroit, Mich. Vol. 17, no. 3. July 25-29, 1983. pp. 35-42.

5. William Buxton and Brad Myers. "A Study in Two-Handed Input," *Proceedings SIGCHI'86: Human Factors in Computing Systems*. Boston, MA. April 13-17, 1986. pp. 321-326.

6. Luca Cardelli and Rob Pike. "Squeak: A Language for Communicating with Mice," *Computer Graphics: SIGGRAPH'85 Conference Proceedings*. San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 199-204.

7. Robert A. Duisberg. "Animated Graphical Interfaces," *Proceedings SIGCHI'86: Human Factors in Computing Systems*. Boston, MA. April 13-17, 1986. pp. 131-136.

8. James D. Foley and Charles F. McMath. "Dynamic Process Visualization," *IEEE Computer Graphics and Applications*. Vol. 6, no. 2. March, 1986. pp. 16-25.

9. Laura Gould and William Finzer. *Programming by Rehearsal*. Xerox Palo Alto Research Center Technical Report SCL-84-1. May, 1984. 133 pages. A short version appears in *Byte*. Vol. 9, no. 6. June, 1984.

10. Daniel C. Halbert. *Programming by Example.* PhD Thesis. Computer Science Division, Dept. of EE&CS, University of California, Berkeley. 1984. Also: Xerox Office Systems Division, Systems Development Department, TR OSD-T8402, December, 1984. 83 pages.

11. D. Austin Henderson, Jr. "The Trillium User Interface Design Environment," *Proceedings SIGCHI'86: Human Factors in Computing Systems.* Boston, MA. April 13-17, 1986. pp. 221-227.

12. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. "Direct Manipulation Interfaces," *User Centered System Design,* Donald A. Norman and Stephen W. Draper, eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1986. pp. 87-124.

13. Robert J.K. Jacob. "A State Transition Diagram Language for Visual Programming," *IEEE Computer.* Vol. 18, no. 8. Aug. 1985. pp. 51-59.

14. Henry Lieberman. "Constructing Graphical User Interfaces by Example," *Graphics Interface, '82,* Toronto, Ontario, March 17-21, 1982. pp. 295-302.

15. Brad A. Myers. "Visual Programming, Programming by Example, and Program Visualization; A Taxonomy," *Proceedings SIGCHI'86: Human Factors in Computing Systems.* Boston, MA. April 13-17, 1986. pp. 59-66.

16. Brad A. Myers and William Buxton. "Creating Highly Interactive and Graphical User Interfaces by Demonstration," *Computer Graphics: SIGGRAPH '86 Conference Proceedings.* Vol. 20, no. 4, August 18-22, 1986. Dallas, Texas. pp. 249-258.

17. Brad A. Myers. *Applying Visual Programming with Programming by Example and Constraints to User Interface Management Systems.* (working title) PhD Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada. In preparation.

18. Brad A. Myers. "The Issue of Semantic Feedback." In preparation.

19. Greg Nelson. "Juno, a Constraint-Based Graphics System," *Computer Graphics: SIGGRAPH'85 Conference Proceedings.* San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 235-243.

20. Dan R. Olsen, Jr., William Buxton, Roger Ehrich, David J. Kasik, James R. Rhyne, and John Sibert. "A Context for User Interface Management," *IEEE Computer Graphics and Applications.* Vol. 4, no. 2. Dec. 1984. pp. 33-42.

21. Dan R. Olsen, Jr., Elisabeth P. Dempsey, and Roy Rogge. "Input-Output Linkage in a User Interface Management System," *Computer Graphics: SIGGRAPH'85 Conference Proceedings.* San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 225-234.

22. Dan R. Olsen, Jr. "Larger Issues in User Interface Management," *Proceedings ACM SIGGRAPH Workshop on Software Tools for User Interface Development.* to appear in *Computer Graphics,* 1987.

23. Gunther R. Pfaff, ed. *User Interface Management Systems.* Berlin: Springer-Verlag, 1985. 224 pages.

24. Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer.* Vol. 16, no. 8. Aug. 1983. pp. 57-69.

25. Ben Shneiderman. "Seven Plus or Minus Two Central Issues in Human-Computer Interfaces," *Proceedings SIGCHI'86: Human Factors in Computing Systems.* (closing plenary address) Boston, MA. April 13-17, 1986. pp. 343-349.

26. David Canfield Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought.* Basel, Stuttgart: Birkhauser, 1977. 187 pages.

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

28. James J. Thomas and Griffith Hamlin, eds. "Graphical Input Interaction Technique (GIIT) Workshop Summary." ACM/SIGGRAPH, Seattle, WA. June 2-4, 1982. in *Computer Graphics.* Vol. 17, no. 1. Jan. 1983. pp. 5-30.

29. Gregg Williams. "The Apple Macintosh Computer," *Byte Magazine.* February 1984. pp. 30-54.

30. Xerox Corporation. *Interlisp Reference Manual.* Pasadena, CA. October, 1983.