Creating Highly-Interactive and Graphical User Interfaces by Demonstration

Brad A. Myers
and
William Buxton

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

ABSTRACT

It is very time-consuming and expensive to create the graphical, highly-interactive styles of user interfaces that are increasingly common. User Interface Management Systems (UIMSs) attempt to make the creation of user interfaces easier, but most existing UIMSs cannot create the low-level interaction techniques (pop-up, pull-down and fixed menus, on-screen "light buttons", scroll-bars, elaborate feedback mechanisms and animations, etc.) that are frequently used. This paper describes Peridot, a system that automatically creates the code for these user interfaces while the designer *demonstrates* to the system how the interface should look and work. Peridot uses rule-based inferencing so no programming by the designer is required, and Direct Manipulation techniques are used to create Direct Manipulation interfaces, which can make full use of a mouse and other input devices. This allows extremely rapid prototyping of user interfaces.

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: Programming by Example, Visual Programming, User Interface Design, User Interface Management Systems, Graphical User Interfaces, Direct Manipulation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-196-2/86/008/0249 $00.75

1. Introduction

This paper discusses Peridot, a new User Interface Management System (UIMS) currently under development, that can create graphical, highly interactive user interfaces. Peridot stands for Programming by Example for Real-time Interface Design Obviating Typing. It is implemented in Interlisp-D [Xerox 83] on a Xerox DandeTiger (1109) workstation, and allows the user interface designer to create user interfaces by *demonstrating* what the user interface should look like and how the end user will interact with it. This approach frees designers from having to do any programming in the conventional sense, and allows them to design the user interface in a very natural manner. The general strategy of Peridot is to allow the designer to *draw* the screen display that the end user will see, and to perform actions just as the end user would, such as moving a mouse, or pressing a mouse button or keyboard key. The system attempts to guess (or *infer*) the relationship of that action to existing elements of the user interface based on context, and asks the designer if the guess is correct. If so, a piece of code is generated by the system that will handle this action for the end user. If incorrect, other reasonable guesses are tried, or the designer can explicitly specify the relationship.

The guesses are encoded as simple condition-action rules, and the generated code is put into small parameterized procedures to help ensure a structured design of the resulting system. The screen displays and interactions depend on the values of the parameters to the procedures. The procedures created by Peridot can be called from application programs or used in other user interface procedures created by demonstration.

Many user interface designers now draw, typically on paper, scenarios (or "story boards") of how the user interface (UI) will look and act. Unfortunately, it is difficult to get a feeling for how a system works from the paper descriptions, and customers of the user interface are not able to investigate how the system will work. Peridot enhances the design process by supporting extremely rapid prototyping with little more effort than drawing the scenarios on paper. In addition, the user interfaces produced by Peridot are expected to be efficient enough for use in the actual end systems.

Another motivation for this style of specifying user interfaces is that it should be possible to allow non-programmers to design *and implement* the interfaces. This will allow professional UI designers (sometimes called "User Interface Architects" [Foley 84]) and possibly even end users, to design and modify user interfaces with little

training and without conventional programming. Virtually all textual UI specification methods are too complicated and program-like to be used by non-programmers [Buxton 83].

The *Direct Manipulation* style of user interfaces [Shneiderman 83][Hutchins 86], where the user typically uses a mouse to select and manipulate objects on the screen, has become very popular (and possibly even predominant) for modern computer systems. Unfortunately, there are virtually no tools available to help develop the low level interaction techniques that support these interfaces, so almost all are laboriously programmed using conventional programming languages. It is well documented in the literature how expensive this process is [Williams 83][Smith 82]. This limits the amount of prototyping possible, and therefore the quality of the interfaces. Existing tools to help build user interfaces, called User Interface Management Systems (UIMSs) [Thomas 83][Olsen 84][Pfaff 85], have not provided a powerful and flexible way to conveniently generate the interaction techniques for these styles of interfaces. In particular, few systems have allowed Direct Manipulation techniques to be used to create the interfaces [Shneiderman 86].

All UIMSs are restricted in the forms of user interfaces they can generate [Tanner 85]. Peridot is only aimed at graphical, Direct Manipulation interfaces. For example, Peridot should be able to create interfaces like those of the Apple Macintosh [Williams 84]. Peridot does not help with textual command interfaces or with the coding of the semantics of the application. The set of interfaces it will produce is rich enough, however, to be very interesting and of practical use for commercial systems.

In summary, the goals of Peridot are that:

1) interaction techniques for Direct Manipulation interfaces should be supported,
2) the system should be easy to use for the designer and require little or no training,
3) the designer should not have to write programs,
4) the interface should be visible at all times as it is developed and changes should be immediately apparent,
5) the *behavior* of the interface should also be created in a Direct Manipulation manner and it should run in real time (points 4 and 5 provide for extremely rapid prototyping), and
6) the system should create run-time code that is efficient enough for use in actual application programs.

This paper presents the design and implementation of the demonstrational aspects of Peridot. A longer report providing more detail and covering other aspects is in preparation [Myers prep]. 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.

2. Background and Related Work

Tanner and Buxton [Tanner 85] present a model of User Interface Management Systems that identifies a number of separate parts (see Figure 1). Peridot is aimed mainly at the "module builder" aspects, but it also covers the "system glue" and "run-time support" components.

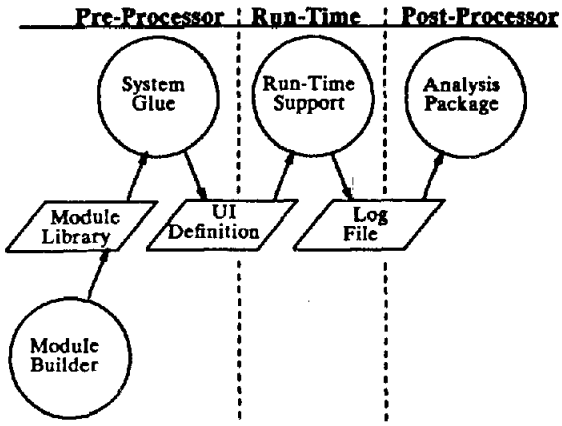

Figure 1.
Model for User Interface Management Systems (from [Tanner 85]).

The "module builder" creates a library of specific interaction techniques. Some systems, such as the Macintosh ToolBox [Apple 85] and the routines that come with most modern window managers [Myers 84][Tesler 81], are essentially the library portion by itself. Using a library has the advantage that the final UI will look and act similarly to other UIs created using the same library, but clearly the styles of interaction available are limited to those provided. In addition, the libraries themselves are often expensive to create. A few UIMSs, such as Syngraph [Olsen 83] and Squeak [Cardelli 85], are designed to help with the creation of the interaction techniques that make up the library, but the indirect and abstract methods used by these programs have proved difficult to use. Peridot attempts to make this process more direct.

Many (probably most) UIMSs concentrate on combining ("gluing") the modules together after they have been created, since it is often non-trivial to write the programs that coordinate the interaction techniques. This is evidenced by the need for the MacApp system to help write programs that use the Macintosh ToolBox. Some, such as Menulay [Buxton 83] and Trillium [Henderson 86], allow the designer to see the design as it is created, but most require that the specification be in a textual language (e.g. [Hayes 85][Jacob 85]). Although a number of modern UIMSs allow the layout of the screen to be specified in a Direct Manipulation manner, virtually all still require the interaction to be specified in an abstract, indirect way, such as using state transition networks. Peridot allows Direct Manipulation to be used for both.

The power in Peridot comes from the use of a new approach to user interface design. The principles of *Programming by Example* and *Visual Programming* have been adapted to allow the designer to demonstrate the desired user interface graphically. These principles are defined, and a comprehensive taxonomy of existing systems that use them is presented, in [Myers 86]. "Visual Programming" (VP) refers to systems that allow the specification of programs using graphics. "Programming by Example" (PBE) systems attempt to infer programs from examples of the data that the program should process. This inferencing is either based on examples of input-output pairs [Shaw 75][Nix 86], or traces of program execution [Bauer 78][Biermann 76b]. Some systems that allow the programmer to develop programs using specific examples do not

use inferencing [Halbert 81 and 84][Lieberman 82][Smith 77]. For example, SmallStar [Halbert 84] 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 [Gould 84], have been successful in making programs more visible and understandable and therefore easier to create by novices.

Peridot differs from these UIMSs and programming systems in that it applies Programming by Example and Visual Programming to the specific domain of graphical user interface specification. Tinker [Lieberman 82] has similar aims, but it does not provide inferencing, and code is specified in a conventional, textual manner in LISP. Early inferencing systems were rather unsuccessful since they often guessed the wrong program and it was difficult for the programmer to check the results without thoroughly studying the code [Biermann 76a]. In limited domains, PBE has been more successful, for example, for editing in the Editing by Example system [Nix 86]. Other systems that are relevant to the design of Peridot are those, such as [Pavlidis 85], that try to "beautify" pictures by inferring relationships among the picture elements (such as parallel and perpendicular) and modifying the picture to incorporate them.

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, we will take a simple interaction: a menu of strings. The operations discussed in this example will be further explained in the following sections. First, however, we present the Peridot screen.

When using Peridot, the designer sees three windows and a menu (see Figure 2). The menu, which is on the left, is used to give commands to Peridot. The window at the top shows the name of the current procedure, the name of its arguments, and *examples* of typical values for those arguments. 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 prompting the designer and for messages. For debugging Peridot itself (and for the very few designers that will be interested), the system can be configured to display the generated code in a fourth window. Currently this code is presented in LISP, but creating a more readable form is possible in the future. The displayed procedure and the picture are always kept consistent, so if the picture is edited, the code is changed, and when the code changes, the picture is also updated. It is not necessary for the designer to view or use the code to perform any operations in Peridot.

Figure 3 shows the steps that can be used to create a procedure that handles a menu with a grey drop shadow. First, the designer types the name for the procedure, ("MyMenu"), the name for the parameters ("Items"), and an example of a typical value for each parameter (the list: ("Replace", "Move", "Copy", "Delete", "Delete All", "Help", "Abort", "Undo", "Exit")). Next, the designer draws a grey box for the shadow and then a black box for the background slightly offset from it (see Figure 3a). These commands are given using the Peridot command menu and a mouse. The system guesses that the black box should be the same size as the grey one and at an offset of 7 in X and Y. The designer confirms that this is correct. Next, (in Figure 3b)

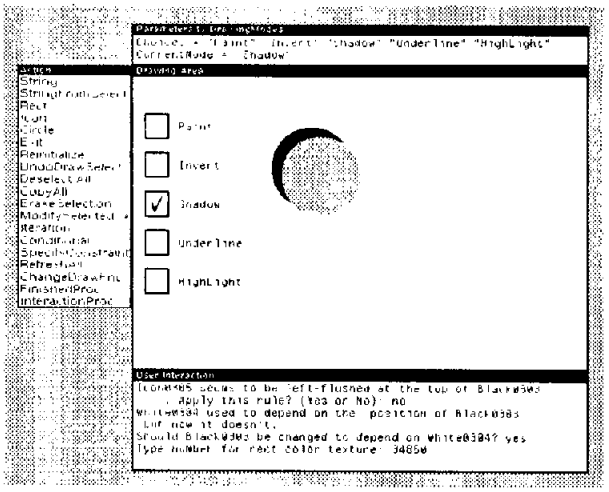

Figure 2.

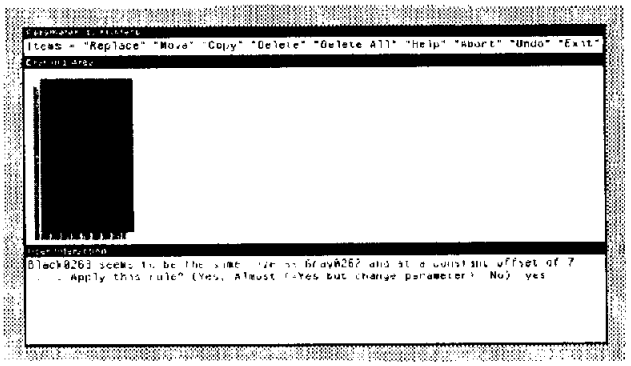

Figure 3a.

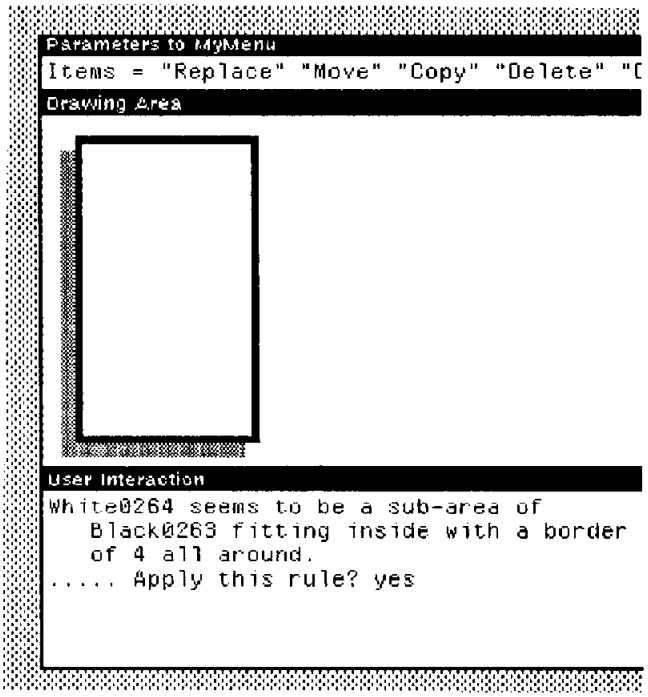

Figure 3b.

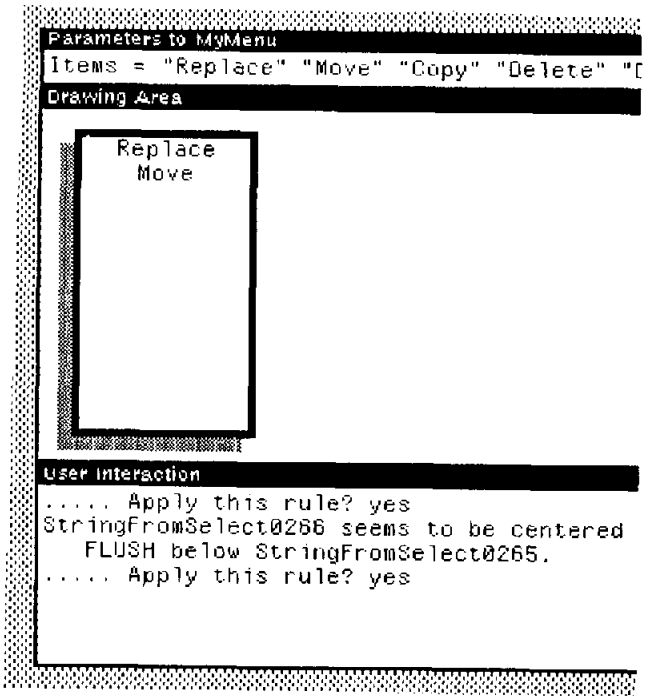

Figure 3c.

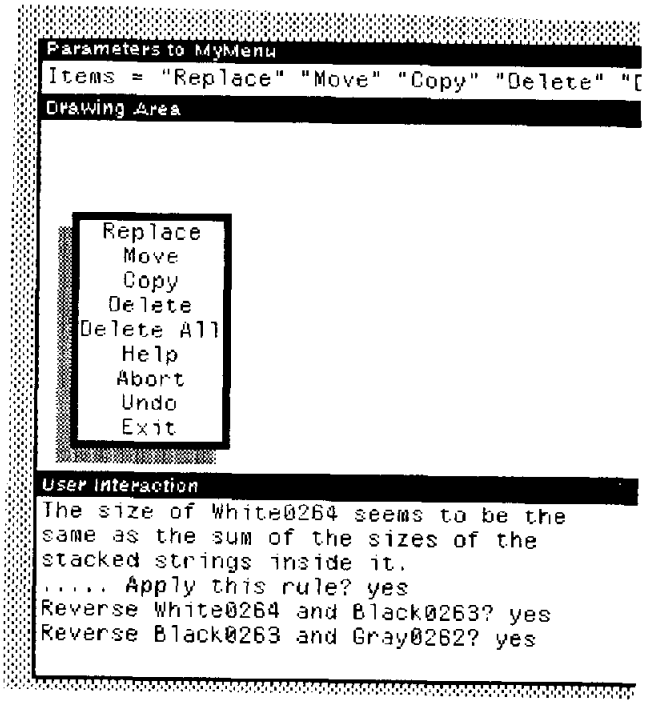

Figure 3e.

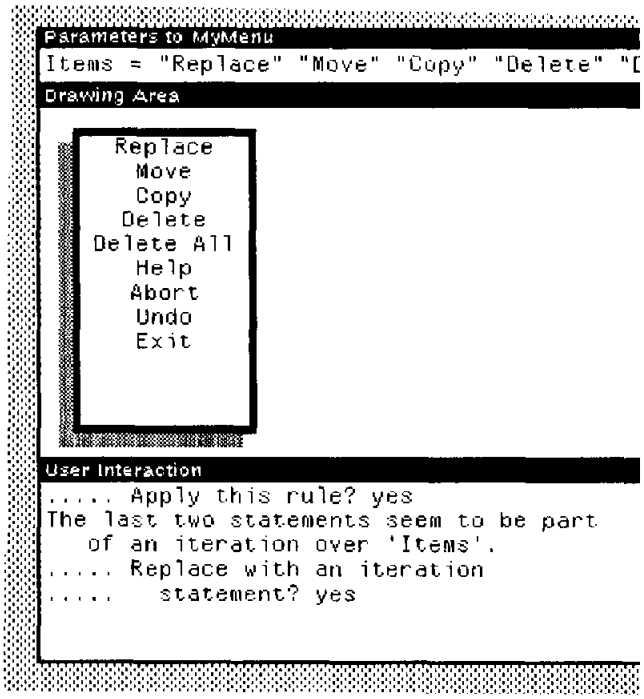

Figure 3d.

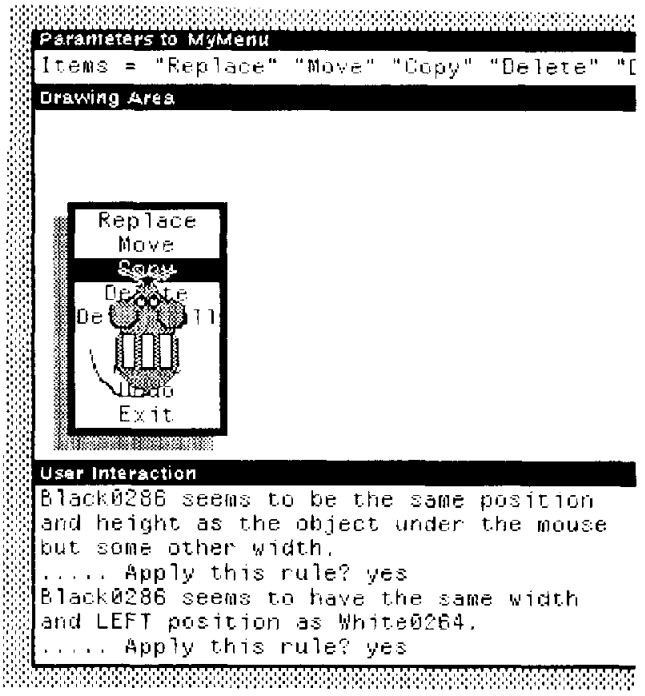

Figure 3f.

A sequence of frames during the definition of a menu interaction technique. (The pictures for 3b-3f have been expanded to be more readable.) In 3a, the shadow and background are drawn (and the system infers that they should be the same size). In 3b, a white area is nested inside the background, and in 3c the first two elements of the parameter are copied to the top of the white rectangle. Peridot notices that they are stacked vertically, and that they are part of an iteration. The rest of the iteration is executed in 3d. The size of the white rectangle is then changed to be just big enough to include all the strings and the system changes the black and grey rectangles accordingly 3e. In 3f, the interaction is being defined using the "simulated mouse."

a white box inside the black one is drawn, and the system adjusts it to be a constant 4 pixels all around, after confirmation from the designer. Next, (in Figure 3c) the first item in the argument ("Replace") is copied to the top of the white rectangle, and the system asks if it should be centered at the top. Peridot makes this assumption because the string was placed approximately centered in the box, as shown in Figure 3c. If the string had been placed left-justified in the box instead, then Peridot would have asked if the string should be left-justified. The system asks the designer to confirm every assumption because sometimes the placing is ambiguous. Next, the second string, "Move", is copied below "Replace" and the system guesses that it is also centered. Since the first two elements of a list have been placed on the screen, the system guesses that the entire list might be desired, so it asks the designer if there should be an iteration to display all elements of the list. After the designer confirms this (in Figure 3d), the system executes the rest of the iteration and changes the code to be a loop. Finally, (in Figure 3e) the designer adjusts the size of the white rectangle to be approximately the size of the strings, and the system asks if the rectangle should be adjusted to fit exactly around all the strings. The sizes of the black and grey rectangles are then automatically adjusted to be proportional to the size of the white rectangle. This completes the presentation aspects of the menu (Figure 3e). It should be remembered that the code being generated does not depend on the specific example values for the parameter; any list of strings will work correctly.

To specify the *interaction* (behavior) of the user interface for the menu, the designer uses an icon that represents the mouse. First, this "simulated mouse" is moved over one of the menu items, and then the designer draws a black rectangle over that item in INVERT drawing mode (see Figure 3f). Peridot infers that the box should be the same height and Y position as the string, and the same width and X position as the white box. The designer then moves the simulated mouse off to the side and erases the black rectangle. Peridot infers that the box should be erased when the mouse is no longer over an object. The designer can perform this action on another string, or explicitly specify an iteration, and the code that handles highlighting is completed. Now the designer "presses" one of the simulated mouse's buttons and specifies, using a Peridot command, that the object under the mouse is returned. From this, the system infers that the procedure should be exited upon button press. The MyMenu procedure is now complete.

Although the textual description of the designer's actions is clumsy, only about ten actions had to be performed to create this procedure (plus confirming Peridot's 12 guesses). Once created, the picture or interaction can be edited, and the menu can used as part of other user interfaces.

4. General Principles of Peridot

One problem with all demonstrational systems is that the user's actions are almost always ambiguous. The system cannot usually know *why* the person did a particular action. This is especially true when the system attempts to infer a general case from a particular example. For instance, when an item is selected, does the user mean that particular item, an item with a similar name, an item at that particular place on the screen, an item with the same type as the selected one, or an item with some other property? Early inferencing systems attempted to solve this problem by guessing and requiring the user to go back later and check the generated code. Non-inferencing systems, such as Halbert's system for the Xerox STAR workstation [Halbert 81 and 84], require the user to explicitly specify why objects were chosen. Peridot, on the other hand, tries to guess what the designer intends by an action, but, to avoid the problems of earlier systems, it always asks the designer if each guess is correct. It is expected that the guesses will usually be correct, which will save the designer from having to specify a great deal of extra detail and from having to know a programming language to express those details. In addition, it is easy to check for errors since the results of all actions and inferences are always immediately visible on the screen.

Any graphical user interface is composed of two parts: the *presentation* or layout, which defines what pictures are on the screen, and the *interaction* or behavior, which determines how these pictures change with user actions. As shown in the previous example, these are specified separately in Peridot. The pictures that Peridot currently supports are: rectangles filled with various grey shades, text strings, filled circles, and static pictures drawn with other programs (e.g. icons)[1].

Peridot uses inferencing in three different ways. First, it tries to infer how various objects in the scene are related graphically. When the designer draws an object, it usually has some implied relation with other objects that have already been drawn. For example, a box might be nested inside another box, or a text string centered at the top of a box. If the picture was simply a static background that never changed, it would not be important for the system to notice these relationships. In Peridot, however, the pictures usually depend on the parameters to the procedure that generate them. For example, the size of the box around a menu might depend on the number of items in the menu and the width of the largest item. Peridot must therefore infer the meaningful relationships among objects from the drawings that the designer produces. This object-object inferencing is described in section 5.1.

The second type of inferencing used by Peridot is to try to guess when control structures are needed. For example, when the designer displays the first two elements of a list, Peridot infers that the entire list should be displayed and will generate an iteration. Conditionals are also inferred for special cases and exceptions. For example, a check-mark might be displayed to show the current value of a set of choices (as in Figure 2). Iterations and conditionals are discussed in sections 5.2 and 5.3 respectively.

The final type of inferencing used by Peridot is to try to guess when actions should happen during the execution of an interaction. For example, a highlight bar might be displayed when the left mouse button goes down. This type of inferencing is described in section 6.

[1] Straight and curved lines, and individual pixels should be easy to add in the future, if needed.

5. Specifying the Presentation of a User Interface

When specifying the presentation of a user interface, the designer is mainly interested in placing graphics on the screen. During this process, however, Peridot is constantly watching the objects to see what object-object relationships there are, and whether some objects drawn would properly be part of an iteration or conditional.

The designer may draw an object on top of another object. Depending on the drawing function in use, the second object may obscure parts of the first object. This is obvious in Figure 3e, where the black rectangle obscures some of the grey rectangle, the white rectangle obscures part of the black one, and the text obscures part of the white one. For this reason, Peridot never changes the order for drawing objects (although the designer is allowed to do this, of course). The calculation order may be changed, however, if a property of an object to be drawn later is needed. For example, in Figure 3e, the width of the strings are needed to calculate the width of the white rectangle even though the rectangle must be drawn first. Peridot insures that the calculation is done in the correct order before the drawing commences.

5.1. *Inferring Object-Object Relationships*

The object-object relationships that are inferred deal with the position and size properties of the objects. The other properties (color, value, font, etc.) are assumed to be constant unless the designer explicitly specifies that they should depend on some other object or parameter. In the example of section 3 above, the colors of the rectangles were constant, but the values for the strings were explicitly specified to depend on the parameter "Items" (by selecting "Replace" and "Move" in the parameter window and using the "StringFromSelect" menu command).

Each object-object relationship that can be inferred is represented in Peridot as a simple *condition-action rule*. Each rule has a test that determines if the relationship is appropriate (the *condition*), a message to be used to ask the designer whether the rule should be applied, and an *action* to cause the objects to conform to the rule. The Appendix lists some sample rules from Peridot. The rules are currently expressed in LISP so the designer will not be able to add new rules. It is very easy, however, for a LISP programmer to modify the rule set.

Since the rules specify very low level relationships (e.g. that a string should be centered inside a box), there appear to be a small number of rules required to handle existing interfaces. In an informal survey of a number of Direct Manipulation interfaces, about 50 rules seemed to be sufficient. In order to allow for human imprecision, however, some leeway must be given to the designer as to the placement and size of objects, so the drawings will not be exact. For example, the designer may want one box to be inside another box with a border of 3 pixels all around, but actually draw it with a border of 5 on one side and 2 on another. Therefore the tests in Peridot for whether to apply a particular rule have thresholds of applicability. Unfortunately, this means that the same drawing may pass more than one test. The *conflict resolution strategy* is simply to order the tests based on restrictiveness (the most demanding tests are first) and based on the heuristically determined likelihood of their being appropriate. This ordering is changed based on the types of the objects being tested, since, for example, it is much more likely for a text string to be centered at the top of a box than for another box to be.

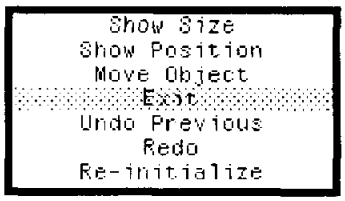

Figure 4.
The grey rectangle is the same height and Y position as the string "Exit" and the same width and X position as the white rectangle.

When the designer draws an object and a rule's test succeeds, Peridot queries the designer whether to apply the rule using the rule's message (see the lower window in Figures 3a-3f). If the system has guessed wrong, the designer answers "no" and the system will try to find a different rule that applies. If the system is correct, the designer may still want to modify parameters of the rule. For example, the system may decide that a box is inside another box with a border of 13 pixels all around, and the designer may decide to use 15 pixels instead. Of course, it may be the case that no rule is found or that the appropriate rule is skipped because the designer has been too sloppy in the original drawing and the rule's test fails. In this case, the designer will usually modify the drawing so that the test will succeed, but it is also possible to explicitly pick a rule to apply.

Most rules in Peridot relate one object to one other existing object[2]. The designer can explicitly specify two objects to apply rules to, but normally the relationships are inferred automatically when an object is created. In this case, the other (existing) object is found by searching through all the other objects in a certain order. When defining the *presentation* of the user interface, the order is: (1) the selected object (the designer can explicitly select an object to apply the rules to), (2) the previous object that was created, and (3) the objects in the vicinity of the new object. When defining the *interaction* portion of the user interface, the order for checking is: (1) the selected object, (2) the object under the simulated pointing devices (see section 6), and (3) the objects in the vicinity of the new object. The system stops searching when an object and a rule are found that completely specifies all of the positional and shape properties of the new object.

Occasionally some of an object's properties may depend on one object and other properties depend on a different object. For example, the highlight bar in a menu may have the same height and "y" value as the string, but the same width and "x" as the surrounding box (see Figure 4). To handle this case, there are rules in Peridot that only define some of the properties of objects. These rules are marked as "incomplete" so that Peridot knows to try additional rules on other objects to handle the rest of the properties (in the Appendix, rule "Rect-same-size" is incomplete).

[2]There are a small number of special rules that test a *group* of objects. This is necessary, for example, to make the size of a box depend on the sum of the sizes of all the items inside it.

Peridot will infer relationships among objects no matter how they are created. Therefore, the same rules will be applied whether an object is created from scratch, by copying some other object, or by transforming an existing object. Since Peridot generalizes from the *results* of the operations, and not *traces* of the actions like many previous Programming by Example systems, it provides much more flexibility to the designers and allows user interfaces to be easily edited. For example, if the designer makes an error when drawing an object or wants to change an existing object, he can simply correct it and Peridot will automatically apply the rules to the new version.

The relationships that Peridot infers can be thought of as *constraints* [Borning 79][Olsen 85] between the two objects. Although the relationships are inferred in one direction (e.g. object R2 depends on object R1), the reverse dependency is also remembered so the relationships can be automatically reversed, if necessary. For instance, the width of the white rectangle in the example of section 3 originally depended on the width of the black rectangle (Figure 3b). When it is later changed to depend on the width of the widest string (Figure 3e), Peridot automatically reverses the constraint with the black rectangle so *black* rectangle's width depends on the *white* rectangle, and similarly for the grey and black rectangles.

Usually, the first object tested is the correct one to apply rules to and the first rule whose test succeeds covers all of the properties of the object. Even when multiple comparisons are required, however, the rule checking occurs without any noticeable delay. If the delay were to increase in the future, this would still not be a problem since the rules are checked at design time (not when the user interface is used by end users), so some delays are acceptable. The advantage of using inferencing rather than requiring the designer to explicitly specify the relationships is that much less knowledge is required by the designer. This is because the designer does not have to know how to choose which of the 50 possible relationships apply and what the parameters to those relationships are.

5.2. *Inferring Iterations*

A recognized problem with all Direct Manipulation systems is that repetitive actions are tedious. For example, if a procedure takes a list of strings to be displayed, the designer does not want to have to individually demonstrate where to display each one. Therefore, Peridot watches the designer's actions to try to infer when two previous actions might be part of a loop. If they appear to be, it queries the designer as to whether a loop is intended. If so, the statements are replaced with a loop statement, and the rest of the loop is executed. As an example, if the designer copies the first two strings from a list of strings and displays them stacked vertically (as in Figure 3c), Peridot asks the designer if the rest of the strings should be displayed in the same manner. If the designer agrees, Peridot calculates how to display the rest of the strings in a similar manner as the first two (as in Figure 3d) and the code for the procedure is automatically changed.

Clearly, this assumes that the objects will be related in some linear fashion, and it will not handle some types of layouts. For example, it will not handle the items of the menu being spaced exponentially, or only displaying every third menu item. Our claim is that these unusual layouts are extremely rare in *real* user interfaces and Peridot will have good coverage without them.

Currently, Peridot infers iterations when the first two elements of a list are displayed[3]. Other objects may also be involved in the iteration, however. For example, in Figure 2, there are black boxes and white boxes for each string taken from the list. Peridot therefore will also include these in the iteration.

5.3. *Inferring Conditionals*

Conditionals are important in user interfaces for specifying *exceptions* and *special cases*. As an example of an exception, a procedure might display a list of strings vertically. However, if one of the strings is a list, then the first element of the list might be the string to be displayed, and the rest of the list might be a sublist to select from after this element is selected. With special cases, the designer wants something extra to happen when certain conditions are met. For example, a check mark may signal the current value from a set of choices, as in Figure 2.

For conditionals, the designer needs a way to specify what to look for to signal the condition (the "IF" part) and what action or actions to perform (the "THEN" part). Peridot supports this by having the designer specify the general case as described above, and giving the "Conditional" command to Peridot. The designer then selects the item that is an exception or special case. For an exception, Peridot tries to infer why it is different, and for a special case, it tries to infer when the graphic should occur. The conditions that are noticed are:

- one value has a different type (e.g. a list versus an atom, or a number rather than a string),
- one is an empty string, or
- numerical properties such as equal to, greater than, or less than zero.

Alternatively, the designer can specify that the value of a parameter should determine whether the conditional should apply. For example, the parameter CurrentMode in Figure 2 determines when to display the check mark.

After Peridot knows the "IF" part, it then allows the designer to demonstrate the "THEN" part, if it is not already displayed, using the same techniques as for any other picture.

Naturally, after a conditional statement is specified, Peridot re-executes the code to insure that the picture is consistent with the new procedure. This causes any additional places where the condition applies to be displayed correctly, which should help the designer spot any errors in the conditional.

6. Specifying the Interaction for a User Interface

One of Peridot's primary innovations is to allow the interaction portion of a user interface to be specified by demonstration. This operates in a similar manner to the presentation component. The major change is the addition of input devices which can determine when actions should take place and the parameters for those actions.

[3] It will be easy to also allow the designer to explicitly specify that an iteration should occur for some integer number of times, where the integer may be constant or depend on the value of some variable.

(a) (b)

Figure 5.
A simulated "mouse" pointing device with three buttons. The device can be moved by pointing at the "nose" (using a real pointing device), and the buttons can be toggled by pressing over them. In (b), the center button is pressed over the word "replace".

Ideally, the designer would simply use the various input devices in the same manner as the end user, but this has three main problems. First, all of the end user's devices may not be available to the designer (for example, in designing the user interface for a flight simulator). Second, some of the input devices are also used for giving commands to Peridot, so disambiguating actions meant for Peridot from those that the end user will perform is difficult. Third, it may be difficult to keep the input device in the correct state (e.g. with a button held down or at a certain location) for the entire time it takes to specify the actions. Therefore, Peridot uses *simulated* devices by having a small icon for each input device (see Figure 5). The designer can move these and toggle "buttons" to indicate what the end user will do with the real input devices.

In addition, it is necessary to have a mode in which the designer can demonstrate what will happen using the *actual* input devices. Although often more clumsy, this is necessary when there are time dependencies, such as with double-clicking or with animations that should happen at a particular speed[4]. In this case, there will be "start watching" and "stop watching" commands to tell Peridot when actions signify what the end user will do and when they are Peridot commands.

When specifying the interaction portion of the user interface, the designer typically moves a simulated input device or changes the status of one of its buttons, and then performs some operation, such as moving an object or drawing a new object. Peridot then creates a conditional statement that is triggered when the input device state or position changes. Of course, there will always be ambiguities (e.g. is the new position significant because it is over an object or because it is no longer over the previous object?) so the designer is always queried to confirm Peridot's guess. Iterations (e.g. perform this until a button is hit), exceptions, and special cases are all be supported for controlling the interaction.

Just as what the end user *sees* is always visible to the designer, what the end user will *do* can also be executed at any time. The designer simply enters execution mode, and the procedure so far is executed. The designer can either use the simulated or the real devices while in execution mode.

[4] It is also intended in the future to allow designers to specify timing dependencies by constraining actions to a clock as in Rehearsal World [Gould 84].

7. Current Status

The design and implementation of Peridot are not complete as of the time of this writing (May, 1986). The inferencing mechanisms in Peridot are working, and the presentation component is mostly complete: object-object inferencing is working, iterations are inferred, as shown in Figures 2 and 3, and conditionals are designed but not implemented, although they are expected to be a straightforward extension. For the interaction component, the correct inferences are being made, but the code generation is not implemented.

8. Future Work

In addition to finishing the implementation of the parts of Peridot that are described here, other aspects of Peridot will be developed. Connections with application programs will use "active values," which behave like continuously evaluated procedures. These can be updated by either the interface or the application and the other will be immediately notified so it can make the appropriate updates.

The designer can easily edit the presentation of an interface after it has been created, but it is a difficult unsolved problem how to allow editing of the interaction component. To support multiple input devices operating in parallel [Buxton 86], multiple processing for procedures and constraints will be added. In addition, multiprocessing and constraints should allow animations and complex echoing and feedback to be specified using Peridot. Peridot will also be tested with a number of different user interface designers to ensure that the same guesses about relationships apply to different people.

9. Conclusions

Although not yet completed, Peridot already is capable of producing a variety of graphical, highly interactive user interfaces. Both the presentation (layout) and interaction (behavior) of these Direct Manipulation interfaces can be created in an extremely natural, Direct Manipulation manner. For example, Peridot can now create light buttons (as in Figure 2), menus (Figure 3), and toggle switches. Automatic inferencing is used to free the designer from having to specify most of the properties of objects. Constant feedback through queries, and continuously making the results of actions visible, helps insure that all inferences are correct. When fully implemented, Peridot should be able to handle the user interfaces of state-of-the-art graphical programs, such as those on the Apple Macintosh and other Direct Manipulation systems, including Peridot's own user interface. Extremely rapid prototyping should be possible, as well as generation of the actual code used in the final user interfaces. Peridot should also be easy enough to use so that even end users will be able to modify the user interfaces of programs. In its present form, Peridot has already demonstrated that the application of rule-based inferencing and Programming by Example techniques to User Interface Management Systems has tremendous potential.

Appendix: Sample rules

This appendix shows the form of three rules used in Peridot. The rules are shown in a LISP-like form, with the arithmetic presented in the normal infix notation to make it more readable. The TEST part determines whether the rule should be applied, the MSG is used to ask the designer for confirmation, the ACTION enforces the rule, and the SPECIFIES field tells which of the graphical properties of the object are covered by the rule. The actual rules in Peridot are slightly more complicated.

```
Rect-same:
  TEST:      (AND ((abs (R1.left - R2.left)) < THRESHHOLD)
                  ((abs (R1.bottom - R2.bottom)) < THRESHHOLD)
                  ((abs (R1.width - R2.width)) < THRESHHOLD)
                  ((abs (R1.height - R2.height)) < THRESHHOLD) )
  MSG:       (CONCAT R1.name
               " seems to be the same size and position as "
              R2.name ".")
  ACTION:    (SETQ R2.left "Fetch R1.left")
               (SETQ R2.bottom "Fetch R1.bottom")
               (SETQ R2.width "Fetch R1.width")
               (SETQ R2.height "Fetch R1.height")
  SPECIFIES: ALL

Rect-same-size-with-same-offset:
  TEST:      (AND ((abs (R1.left - R2.left)) < BigTHRESHHOLD)
                  ((abs (R1.bottom - R2.bottom)) < BigTHRESHHOLD)
                  ((abs (R1.width - R2.width)) < SmallTHRESHHOLD)
                  ((abs (R1.height - R2.height)) < SmallTHRESHHOLD)
                  ((abs ( (abs (R1.left - R2.left)) -
                     (abs (R1.bottom - R2.bottom)) )) < SmallTHRESHHOLD) )
  MSG:       (CONCAT R1.name " seems to be the same size as "
              R2.name " and at a constant offset of "
              (SETQ offset (ave ((abs (R1.left - R2.left)) -
                                 (abs (R1.bottom - R2.bottom)))))
             ".")
  ACTION:    (SETQ R2.left (CONCAT "Fetch R1.left + " offset))
             (SETQ R2.bottom (CONCAT "Fetch R1.bottom + " offset))
             (SETQ R2.width "Fetch R1.width")
             (SETQ R2.height "Fetch R1.height")
  SPECIFIES: ALL

Rect-same-size:
  TEST:      (AND ((abs (R1.width - R2.width)) < THRESHHOLD)
                  ((abs (R1.height - R2.height)) < THRESHHOLD) )
  MSG:       (CONCAT R1.name " seems to be the same size as "
              R2.name " but in an unrelated place.")
  ACTION:    (SETQ R2.width "Fetch R1.width")
             (SETQ R2.height "Fetch R1.height")
  SPECIFIES: (width height)
```


ACKNOWLEDGEMENTS

First, we 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, we would like to thank the SIGGRAPH referees, and Bernita Myers, Peter Rowley, Ralph Hill, and Ron Baecker.

REFERENCES

[Apple 85] Apple Computer, Inc. *Inside Macintosh.* Addison-Wesley, 1985.

[Bauer 78] Michael Anthony Bauer. *A Basis for the Acquisition of Procedures.* PhD Thesis, Department of Computer Science, University of Toronto. 1978. 310 pages.

[Biermann 76a] 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.

[Biermann 76b] Alan W. Biermann and Ramachandran Krishnaswamy. "Constructing Programs from Example Computations," *IEEE Transactions on Software Engineering*. Vol. SE-2, no. 3. Sept. 1976. pp. 141-153.

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

[Buxton 83] 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.

[Buxton 86] 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.

[Cardelli 85] 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.

[Foley 84] James D. Foley. "Managing the Design of User-Computer Interfaces," *Proceedings of the Fifth Annual NCGA Conference and Exposition.* Anaheim, CA. Vol. II. May 13-17, 1984. pp. 436-451.

[Gould 84] 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.

[Halbert 81] Daniel C. Halbert. *An Example of Programming by Example.* Masters of Science Thesis. Computer Science Division, Dept. of EE&CS, University of California, Berkeley and Xerox Corporation Office Products Division, Palo Alto, CA. June, 1981. 55 pages.

[Halbert 84] 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.

[Hayes 85] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner. "Design Alternatives for User Interface Management Systems Based on Experience with COUSIN," *Proceedings SIGCHI'85: Human Factors in Computing Systems.* San Francisco, CA. April 14-18, 1985. pp. 169-175.

[Henderson 86] 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.

[Hutchins 86] 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.

[Jacob 85] Robert J.K. Jacob. "A State Transition Diagram Language for Visual Programming," *IEEE Computer.* Vol. 18, no. 8. Aug. 1985. pp. 51-59.

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

[Myers 84] Brad A. Myers. "The User Interface for Sapphire," *IEEE Computer Graphics and Applications.* Vol. 4, no. 12, December, 1984. pp. 13-23.

[Myers 86] 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.

[Myers prep] Brad A. Myers. *Applying Visual Programming with Programming by Example and Constraints to User Interface Management Systems.* PhD Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada. In progress.

[Nix 86] Robert P. Nix. "Editing by Example," *ACM Transactions on Programming Languages and Systems.* Vol. 7, no. 4. Oct. 1985. pp. 600-621.

[Olsen 83] Dan R. Olsen and Elizabeth P. Dempsey. "Syngraph: A Graphical User Interface Generator," *Computer Graphics: SIGGRAPH'83 Conference Proceedings.* Detroit, Mich. Vol. 17, no. 3. July 25-29, 1983. pp. 43-50.

[Olsen 84] 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.

[Olsen 85] Dan R. Olsen, Jr., Elisabeth P. Dempsey, and Roy Rogge. "Input-Output Linkage in a User Interface Management System," *Computer Graphics: SIGGRAPH'83 Conference Proceedings*. San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 225-234.

[Pavlidis 85] Theo Pavlidis and Christopher J. Van Wyk. "An Automatic Beautifier for Drawings and Illustrations," *Computer Graphics: SIGGRAPH'85 Conference Proceedings*. San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 225-234.

[Pfaff 85] Gunther R. Pfaff, ed. *User Interface Management Systems*. Berlin: Springer-Verlag, 1985. 224 pages.

[Shaw 75] David E. Shaw, William R. Swartout, and C. Cordell Green. "Inferring Lisp Programs from Examples," *Fourth International Joint Conference on Artificial Intelligence*. Tbilisi, USSR. Sept. 3-8, 1975. Vol. 1. pp. 260-267.

[Shneiderman 83] Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*. Vol. 16, no. 8. Aug. 1983. pp. 57-69.

[Shneiderman 86] 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.

[Smith 77] David Canfield Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Basel, Stuttgart: Birkhauser, 1977. 187 pages.

[Smith 82] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem. "Designing the Star User Interface," *Byte Magazine*, April 1982, pp. 242-282.

[Tanner 85] 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.

[Tesler 81] Larry Tesler. "The Smalltalk Environment," *Byte Magazine*. August 1981, pp. 90-147.

[Thomas 83] 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.

[Williams 83] Gregg Williams. "The Lisa Computer System," *Byte Magazine*, February 1983, pp. 33-50.

[Williams 84] Gregg Williams. "The Apple Macintosh Computer," *Byte Magazine*. February 1984. pp. 30-54.

[Xerox 83] Xerox Corporation. *Interlisp Reference Manual*. Pasadena, CA. October, 1983.