A New Model for Handling Input

BRAD A. MYERS
Carnegie Mellon University

Although there has been important progress in models and packages for the *output* of graphics to computer screens, there has been little change in the way that *input* from the mouse, keyboard, and other input devices is handled. New graphics standards are still using a fifteen-year-old model even though it is widely accepted as inadequate, and most modern window managers simply return a stream of low-level, device-dependent input events. This paper presents a new model that handles input devices for highly interactive, direct manipulation, graphical user interfaces, which could be used in future toolkits, window managers, and graphics standards. This model encapsulates interactive behaviors into a few "Interactor" object types. Application programs can then create instances of these Interactor objects which hide the details of the underlying window manager events. In addition, Interactors allow a clean separation between the input handling, the graphics, and the application programs. This model has been extensively used as part of the Garnet system and has proven to be convenient, efficient, and easy to learn.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; I.3.6 [**Computer Graphics**]: Methodology and Techniques

General Terms: Human Factors

Additional Key Words and Phrases: Direct manipulation, input devices, interaction, interaction techniques, model-view-controller, object-oriented design, user interface management systems.

1. INTRODUCTION

The Interactors subsystem of the Garnet project [21] provides a new model for handling input from the mouse and keyboard. This model provides a high-level interface that is independent of the details of the underlying window manager's input event mechanism, but is still flexible enough to support arbitrary looks and feels. Interactors can handle all conventional mouse and keyboard-based interaction techniques such as menus, scroll bars, and buttons, as well as application-specific interactions such as selecting and moving boxes and arrows in a graph editor. Each Interactor is entirely independent of the particular

This research was sponsored by DARPA (DOD) under contract F33615-87-C-1499, ARPA order 4976, Amendment 20, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
Author's address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

graphics used for the feedback. Therefore, the graphics, the input handling, and the application program can all be defined separately and independently. Interactors also support multiple input devices operating at the same time and multithreaded dialogues.

The observation that makes Interactors feasible is that there are only a few distinct behaviors used in graphical user interfaces. For example, although the graphics can vary significantly and the specific mouse buttons may change, most menus behave in the same manner. Another example is that many different kinds of objects might want to follow the mouse around (be "dragged" with the mouse).

Garnet captures these common behaviors in a small set of Interactor objects while still providing a high degree of customizability to application programs. There are currently only six types of Interactors, and these are sufficient to handle all interaction in our user interfaces. The types are Menu-Interactor, Move-Grow-Interactor, New-Point-Interactor, Angle-Interactor, Text-Interactor, and Trace-Interactor. Parameters to these Interactors, such as what events cause them to start and stop and what graphics should be used, allow significant customization.

This paper discusses the current design of the Garnet Interactors, along with a number of important trade-offs and design decisions that were made along the way. It is a longer version of a previous paper [18]. A complete reference manual for the current implementation of Interactors is also available [22].

2. RELATED WORK

Garnet can be considered a User Interface Management System since it helps build user interface software. UIMSs have been surveyed in various articles [17, 24].

The primary influence on the Garnet project was the Peridot UIMS [16, 19]. Peridot was a construction tool that allowed toolkit items (menus, scroll bars, buttons, title line, and iconic controls for windows, etc.) to be created without programming. It successfully encapsulated mouse interactions into three different kinds of objects, and was able to handle all the mouse-based interaction techniques in the Macintosh Toolbox. Peridot did not handle any keyboard input. The Garnet Interactors are generalizations of the interaction objects in Peridot and are easier to use.

There have been many other attempts to separate input devices from application programs. The approach used by all of the graphics standards (PHIGS, GKS, CGI, CORE, etc.) identifies five or six basic input types (e.g., locator, stroke, valuator, choice, pick, and string for PHIGS [28]). This is based on a model by Foley and Wallace [7]. In all of these, the goal is to free the designer from details of the interaction and the device it is implemented on. This is achieved by hiding, for example, whether "pick" is implemented by pointing on the screen, by typing the name of an object, or by successively highlighting items and having the user press a button when the correct one is highlighted. Similarly, the programmer has no control over what kind of menu is used when a "choice" input is requested.

Unfortunately, these abstractions have proven to be inadequate and inappropriate for modern user interfaces [14], in two important ways:

(1) They are modal. For example, if the programmer requests a "choice" input, the system blocks while waiting for the user to choose an item from a menu. The user cannot, say, choose a graphic object instead. Therefore, it is not possible to allow the user either to pick a graphical item in the main editor window or to choose from a palette or one of a set of menus, as is common in virtually all Macintosh applications.
(2) They restrict the programmer too much. There is little control over the style of menus, the way that graphic objects are selected, or what feedback is shown during the operation.

The Garnet Interactors provide a similar degree of device-independence, while being nonmodal and providing flexibility of look and feel (for example, the programmer has complete control over the feedback graphics and behavior for menus).

The Garnet Interactors are similar to the Smalltalk "model-view-controller" (MVC) paradigm [12]. The idea in MVC is to separate the code into three parts: The *model* which embodies the application semantics, the *view* which handles the output graphics that show the model, and the *controller* which handles input and interaction. Unfortunately, programmers have found that the code for the controller and view are often tightly interlinked; creating a new view usually requires creating a corresponding new controller. In fact, both are often also entwined with the model, so all three need to be recoded.

In Garnet, the Interactors serve as the controller, the object-oriented graphics are the view, and conventional Lisp code is used for the model. Constraints are used to tie the parts together. The primary difference from the Smalltalk MVC is that in Garnet, the programmer does not create new types (subclasses) of Interactors, but rather supplies parameters to instances of the built-in types. This makes it clearer what information should be implemented as part of the interactors and what should go in the view and model. Also, because there are built-in interactors for the behaviors that programmers need, the programmer does not need to code event handlers, so it is less work to make objects respond to input in Garnet than in Smalltalk.

Because in the MVC paradigm it is often difficult to decide which functionality should go into the view and which into the controller, some object-oriented systems either have not used any separation, or, like the Andrew toolkit [26], have only used two parts: the view (which handles all input, output, and interaction) and the model. The problem with this is that the view must handle all input events and output drawing and must be recoded whenever either changes.

The MacApp application builder for the Macintosh uses "command" objects to encapsulate the handling of input [33]. When a menu item is selected, MacApp creates the appropriate command object and calls a standard method in it. The command object is responsible for executing the command and saving enough information to undo it. MacApp supplies standard command objects

for some operations, such as opening and closing files and moving a bounding box with the mouse, but for most operations, the programmer must code new methods.

Other object-oriented systems have also had specific objects to handle commands. For example, GWUIMS uses the same split as MVC, separating the interface into representation objects, interaction objects, and application objects [29]. IDL has an "action schema" for each action defined by the user interface and a "parameter schema" for each parameter required for the action [9]. These must be coded for each new command and parameter.

A significant difference in Garnet from the above systems is that the Interactor types are parameterized sufficiently so that the programmer does not need to create new types or subclasses. Therefore, the Interactors are much more than just a methodology for structuring the software that the programmer must write; rather, they supply high-level functionality that programmers use.

The X toolkit tries to separate some properties of the graphics from the rest of the interaction techniques with separate "geometry managers," but typically, new geometry managers are needed for every new interaction technique.

In some user interface tools, such as DialogEditor [5], the term "Interactor" is used for what this paper calls interaction techniques or widgets. These are buttons, menus, sliders, and so forth, and contain *both* graphics and behavior. These are therefore not related to the "Interactors" described here.

Some other ideas that have been used for programming the response to input events are transition networks [11], event languages [10], and multiple-process models [4]. Unfortunately, these have proven difficult to use and unpopular with user interface designers [25].

3. FEATURES OF INTERACTORS

Interactors make it much easier to program user interfaces, while still providing a high degree of flexibility. In particular,

—they are independent of the graphics and application program and can be specified separately;
—the details of the input handling are hidden from the programmer, providing window manager independence;
—they make it easier to create user interface building tools;
—they make programming user interfaces easier;
—they support rapid prototying;
—they promote code reuse;
—they support multiple behaviors attached to the same object;
—they support multiple input devices operating in parallel;
—they simulate multiple processing in a single process; and
—they can handle "semantic feedback."

These features are expanded upon in the following sections.

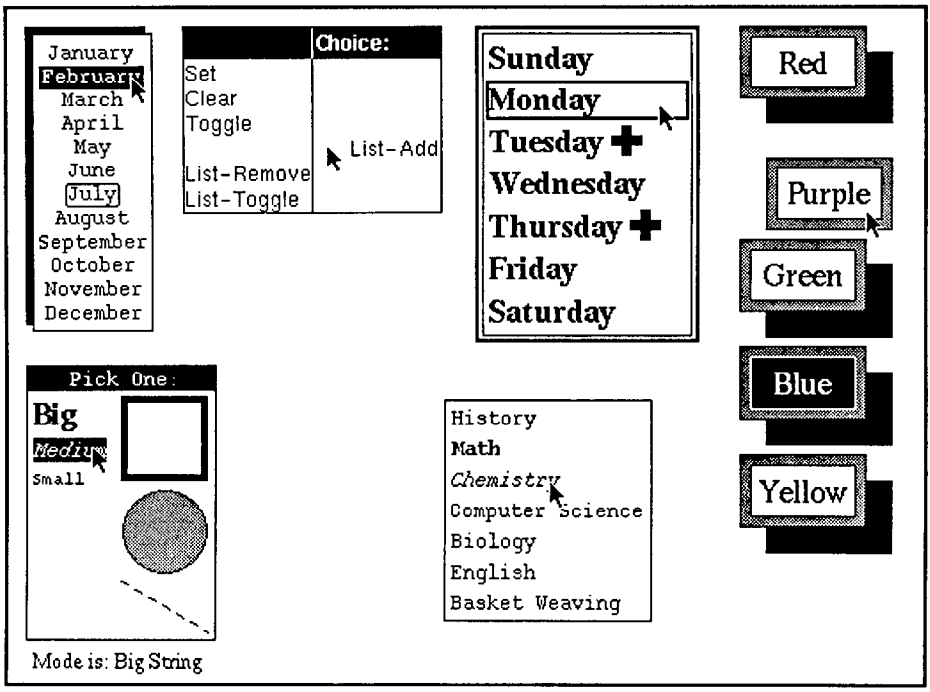


Fig. 1. The same Interactor type can be used for menus with very different feedback and background graphics. Here, the interim feedback is a reverse-video rectangle (a) and (e), an outline box (c), moving the item towards its shadow (d), or changing the item itself to be italic (f). The final feedback is an outline box (a), moving the item left or right (b), plus signs (c), a reverse-video rectangle (d), or turning the item itself bold (f).

3.1 Independent of the Graphics and Application Program

It is well known that separating the definition and coding of the graphics, interactive behavior, and application program is an important, yet very difficult goal in user interface software design [10, 27]. The Interactors in Garnet are entirely independent of the particular graphics used to present the behavior (see Figure 1). There is a standard protocol used for interactors to query and modify the graphics, so each can be defined separately. They can then be attached to application programs using constraints (which are relationships that are defined once and maintained automatically by the system) or conventional call-back procedures.

Many existing toolkits, such as the Macintosh Toolbox and Motif, provide a fixed look and feel. Garnet is specifically designed to allow explorations of new looks and feels. Predefined widgets can be selected from a library when that is desired (a library of widgets with a unified look and feel is supplied with Garnet).

Providing a look and feel independent interface is also required so that Interactors can handle the interactions of the application-specific parts of the user interface, which most other toolkits ignore. For example, Interactors support

the selection and moving of arbitrary application objects, independent of the look of the application graphics.

3.2 Hide Input Handling

Interactors also provide a level of window manager independence. The designer is freed from details of how events are queued and how exception conditions are presented. The object-oriented graphics package of Garnet and the Interactors provide a complete layer that hides the complex details of the window manager interface from application programs. This makes it possible to have Garnet implementations for different window managers (such as X/11 and Macintosh QuickDraw) and allow applications to be ported with little or no modifications. The event handling portion of the code in the Interactors package is only about one page long, so porting it to new window managers is easy, especially since all window managers use essentially the same input model.[1]

3.3 User Interface Tools

Since behaviors can be specified by simply supplying parameters to Interactors, it is easy to provide graphical, direct manipulation tools that can attach input handling to graphics. For example, the Garnet user interface builder, called Lapidary, provides dialog boxes for attaching behaviors to objects (see Section 8).

3.4 Make Programming Easier

The primary goal of Interactors is to allow the commonly used and "surprisingly intricate" [4] input device handling code to be created easily. This has been achieved by identifying the primitive behaviors used in interfaces and the appropriate parameters to customize them. The programmer then only needs to create an instance of the desired behavior, attach it to the graphics, and the user interface is ready to operate.

Often, it is not even necessary for the programmer to write any code. The Interactors can be completely created by demonstration using graphical tools, such as Lapidary.

3.5 Rapid Prototyping

Another important advantage is that is is easy to investigate different looks and feels quickly. New parameters can be supplied to the existing Interactors, either by programming or by using Lapidary, and the new interface executed immediately. Similarly, new Interactors can be attached to graphics on the fly, even while the interface is running.

3.6 Code Reuse

In most conventional object-oriented systems and object-oriented toolkits, when the programmer wants a new form of interaction technique or widget, both the messages that handle the graphics and those that handle the behavior must be reprogrammed, since usually none can be inherited without change.

[1] Porting the output graphic operations requires more work, however.

With Interactors, the programmer uses the built-in behaviors. An instance of an Interactor of the appropriate type is created, some parameters are supplied, and the new behavior is ready. If an existing Interactor supplies almost the correct behavior, the programmer can simply use that Interactor as a prototype, and override the particular parameters that need to be changed and the rest will be inherited. Therefore, less new code needs to be written.

3.7 Multiple Behaviors on the Same Objects

Another difference from conventional object-oriented approaches, where the behaviors are all inherited from super classes, is that different behaviors can be attached to the same graphics. This could be used, for example, to have one Interactor operate a menu when the left mouse button is pressed and another Interactor move the menu when the right mouse button is pressed. These kinds of behaviors can rarely be mixed in other systems without writing new methods, even when multiple-inheritance is available.

This can also be used to combine different Interactors to create a composite behavior. For example, a scroll bar can be created out of Menu Interactors for the buttons and a Move-Grow Interactor for the indicator itself.

3.8 Multiple Devices and Processes

Interactors directly support multiple input devices active at the same time. Therefore, Interactors can be used when multithreaded dialogues [31] are desired. For example, one Interactor might be handling input from the mouse while another is handling the keyboard. The application program can be completely unaware of this parallelism because it is handled internally by the Interactors. Research has shown that people can effectively and easily use two hands to provide such parallel input and thereby execute certain tasks much quicker [2]. In the future, there will be additional Interactor types to handle alternative input devices, such as joysticks, physical knobs, and touch tablets.

In addition, Interactors can simulate multiple processing in a single process. Different windows can have different applications running in them in the same Lisp process, but events will be distributed appropriately to the appropriate window. As long as the processing associated with each event is short, it will appear as if each window has its own process. When a long operation is required, and if real multiple processing is available, the operation can be passed off to another process so the interaction can continue.[2]

3.9 Semantic Feedback

A system uses "semantic feedback" when the application program must process the input events in order to decide what feedback to show the user. Interactors modify feedback graphics indirectly (see Section 6.3.1), so application programs can insert constraints to perform appropriate filtering on the data.

[2] Garnet is designed to run on any CommonLisp implementation, and there is no standard multiple-process mechanism in CommonLisp, although many versions do have one.

4. THE GARNET PROJECT

Garnet, which stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, aims to make highly interactive, direct manipulation user interfaces easier to create [21]. Garnet is implemented in CommonLisp and runs on top of the X/11 window system or Macintosh QuickDraw. Garnet is therefore portable and runs on various machines and operating systems. It does *not* use the Common-Lisp Object System (CLOS) or any Lisp or X toolkit (such as CLUE, CLIM, Xtk, or InterViews).

Garnet contains both low-level and high-level tools. The low level is the "Garnet Toolkit," and it contains a prototype-instance object system, a constraint system [30], a graphic object system featuring automatic graphic object updating, the Interactors, and a collection of widgets such as menus, gauges, buttons, scroll bars, browsers, error windows, and so forth.

The toolkit has been available for some time now and is in active use by over 25 projects at CMU and elsewhere.[3]

The high-level Garnet tools include the Lapidary interface builder [20], which allows the user interface designer to draw pictures of what the user interface should look like and then demonstrate how it should act, and the Jade Dialog Box creation system [32], which automatically creates menus and dialog boxes from a list of their contents.

Garnet is designed to handle interfaces containing a number of graphic objects (up to about 2,500) which the user can manipulate with the mouse and keyboard. Garnet is suitable for applications such as drawing programs like MacPaint and MacDraw, network editors like MacProject, iconic command interfaces like the Macintosh Finder, graphical programming language editors [15], tree and graph editing programs, board game user interfaces, simulation and process monitoring programs, form and dialog-box based interfaces, and some types of CAD systems. Garnet will not handle command line parsing or text editing (other than small strings used as labels).

5. TAXONOMY OF INTERACTION TECHNIQUES

The design for the Interactors was based on the observation that there are only a few ways in which the mouse and keyboard are used in interfaces. An earlier classification [8] listed the following interaction tasks: select, position, orient, path, quantify, and text. If only a mouse and a keyboard are available, then the quantify task can only be accomplished by typing a number (text) or moving an on-screen slider (position).

In Garnet, the five remaining tasks are each assigned to a separate Interactor type. The position task is actually divided into two Interactors, depending on whether a new object is about to be created or an existing object is being modified. Therefore, the six types of Interactors are:

—Menu-Interactor—select,
—Move-Grow-Interactor—position,

[3] The Garnet toolkit is available for free. It is currently licensed to over 80 companies and universities throughout the world. Contact the author for more information.

—New-Point-Interactor—position,
—Angle-Interactor—orient,
—Text-Interactor—text,
—Trace-Interactor—path.

These are sufficient to cover all of the kinds of interactive behavior used in mouse-based, graphical user interfaces (see Section 7).

Since Interactors are implemented in an object-oriented fashion, it is relatively easy to create new Interactor types, but this is only necessary for radically new interaction tasks, such as gestures and character recognition. Also, new input devices such as joysticks and touch tablets will probably require new types of Interactors.

6. DESIGN OF THE INTERACTORS

In a user interface, there is one Interactor object for each set of graphic objects that behave similarly. For example, there is one Interactor object to handle each menu (rather than, say, one per item). The Interactor object then handles all input events that are relevant to those graphics, and modifies the graphics as appropriate through a well-defined protocol.

Each Interactor is parameterized in various ways. For example, the designer can specify which mouse button or keyboard key causes the Interactor to start operating. The parameters for Interactors are described in Section 6.1. Then the specific Interactor types are presented (Section 6.2), followed by the protocol that the Interactors use to connect to the graphics (Section 6.3) and application programs (Section 6.4). Some extra examples are given in Section 6.5, followed by details about the internal state machine of each Interactor (Section 6.6), and a discussion of how they are used in widgets (Section 6.7). Finally, some advanced features of Interactors are discussed in Section 6.8, such as the use of formulas and priority levels, and the support for modes and multiple windows. Section 7 discusses the coverage of the Interactors in detail, and Section 8 presents some techniques for creating Interactors without programming. Section 9 discusses some tools that can be used to debug interfaces created using Interactors.

6.1 Parameters to Interactors

In designing the Interactors, there are many tradeoffs that have to be considered. The current design attempts to balance flexibility and power with ease of use. The interactors in the earlier Peridot system were much simpler than Garnet's and had fewer parameters. Therefore, multiple interactors were needed for many common operations. For example, to have an outline (feedback) box follow the mouse while a button is pressed and have the object jump to the final position when the mouse button is released (as in Figure 4b) required three Peridot interactors: one to control the visibility of the feedback object, one to have it track the mouse, and one to have the actual object jump to the final position. In Garnet, the Interactors are at a higher level so that this behavior is achieved with one Interactor. A result of this is that individual Interactors have more parameters (to control the various options for feedback), but typical operations are much easier to achieve.

The following sections discuss some of the parameters to Interactors that are shared by the different types. Other parameters that are unique to a specific type of Interactor are discussed in Section 6.2. It is important to mention that there are reasonable defaults for all values, so typically the programmer has to specify very little to create an Interactor. For example, the default start event is :leftdown (left mouse button down), and the default stop event is :leftup, and if this is acceptable, the programmer does not have to specify any events. Figure 2 summarizes all the slots of Interactors available to the programmer.[4]

6.1.1 *Events.* An Interactor will start running when its *start event* occurs and continues running until a *stop event* occurs. There may also be an *abort event* that will prematurely cause it to exit and restore the status as if it had not started.[5] All Interactors are able to restore the state if an abort event happens.

The "event" is usually a transition of a mouse button or keyboard key. The value provided to an Interactor can be a single event (e.g., :left button, #\control-G), one of a set of events (e.g., (:leftbutton or :rightbutton, :anykeyboard), or one of these with exceptions (e.g., :anybutton :except :leftbutton). These can also be qualified using the standard modifier keys: shift, control, or meta.

6.1.2 *Feedback.* The most unique parameters to Interactors control which graphic objects are modified by the operation. There is a standard protocol through which the objects are queried and modified (see Section 6.3). This holds for both feedback objects, if any, and the actual objects modified.

There are two types of feedback that might be associated with an interactor. *Interim* feedback shows what is happening while the interaction is in progress. *Final* feedback might be used to show the user's final selection (see Figure 3). Parameters to the Interactor specify the objects to use as feedback or whether there is a feedback object at all. For example, (see Figure 4) the Interactor that moves objects can have either the object itself following the mouse, or special interim feedback graphics might follow the mouse (an outline box is used in MacDraw), and the object jumps to the final location when the Interactor completes (e.g., when the button is released). Garnet will automatically replicate the feedback object as necessary, if multiple selections are allowed.

Since all graphic objects are handled alike, the Interactors are entirely independent of the specific graphical representations on the screen. For example, the feedback graphics to show which item in a menu is selected can be an outline box, a reverse-video rectangle, a collection of objects that form a plus sign, or whatever the user interface designer can imagine, and the same Interactor is used without modification (see Figure 1). In fact, the feedback can be a change to the menu items themselves, rather than a separate object that is XORed over the items. For example, Figure 1f shows a menu where the items change to be bold or italic, and in Figures 1b and 1d, the items move when selected. This is explained more fully in Section 6.3.

[4] "Slot" is our name for an instance variable, and the slot names in Garnet always start with a colon. The "parameters" to interactors are specified by providing values for the various slots, as explained in Section 6.5. This paper therefore uses "slot" interchangeably with "parameter."

[5] Section 6.6 discusses the state machine that controls Interactors in more detail.

6.1.3 *Which Objects.* A parameter to the Interactor determines where the mouse needs to be when the start event happens for the Interactor to start operating. For example, in a menu, this would be "over one of the items in the menu." Similarly, for a graphics editor, the list of objects might be any of the graphic objects that can be selected. This is encoded into the :start-where slot of the interactor.

Another parameter determines which graphic objects the Interactor operates on. Usually, this is the same as the objects the Interactor is defined to start over. For example, if the programmer specifies to start over an item in a menu, the Interactor will operate on the items of the menu. A case when these are not the same is a scroll bar that allows the user to press anywhere in the "elevator" (background) to start moving the indicator (rather than requiring a press on the indicator as in the Macintosh). Here, the :start-where is over the background, but the object to change is the indicator. Therefore, in this case, the indicator object would need to be provided as the :obj-to-change parameter.

The :start-where slot takes a list that specifies whether to look at a single object, any element of an aggregate object (optionally, only those elements of a certain type), or any element of a list of objects.

If the programmer wants some items to be *illegal* and not pickable (such as the gray items in Macintosh menus), then this information can be passed to the interactor in a different parameter. Of course, the particular graphics used to show that the item is illegal is arbitrary and independent of the Interactor.

In general, there will be an instance of one of the Interactor types for each *entire* menu, each set of buttons, and each set of objects to be moved. For example, for a scroll bar with two arrow icons (see Figure 5), there would be two Interactors: one Menu-Interactor to handle both icons, and one Move-Grow-Interactor to allow the indicator to be moved. Having the Interactors work on a *set* of objects (rather than just on one individual object) significantly decreases the number of Interactors that are needed.

6.1.4 *Outside.* Many interactions should be suspended if the mouse goes outside of an active region. Therefore, a parameter to Interactors specifies what the active area is, and whether going outside signifies to use the last legal value or to go back to the original value. For example, moving outside of a menu might cause there to be no selection, whereas moving outside of a slider might use the last legal, inside value. The programmer can also specify that the operation simply continue no matter where the mouse moves. This is all encoded into the :running-where parameter to the interactors.

6.1.5 *Continuous or One-shot.* A parameter determines whether the Interactor operates once when the start event happens or continuously from the start event to the stop event. For example, one menu might select the item under the mouse when the left button is pressed and ignore any subsequent motions (one-shot), whereas another menu might have interim feedback that follows the mouse while the button is held down and stops when the mouse button is released (continuous).

6.2 Types of Interactors

As described in Section 5, there are six types of Interactors. The following sections describe how they operate.

Slot	Section	Parameter type	Default value	Comment
:start-event	6.1.1	keyboard key, mouse button or combination	:leftdown	Starts the interactor
:stop-event	6.1.1	keyboard key, mouse button or combination	:leftup	Stops the interactor
:abort-event	6.1.1	keyboard key, mouse button or combination	^G	Aborts the interactor
:interim-feedback	6.1.2	an object or collection of objects	NIL	Object to be used as interim feedback. Its :obj-over slot is set
:final-feedback	6.1.2	an object or collection of objects	NIL	Object to be used as the final feedback. Its :obj-over slot is set
:start-where	6.1.3	one of :in, :element-of, :list-element-of followed by an object	*must be supplied*	Where the mouse must be when the start event happens for the interactor to begin
:obj-to-change	6.1.3	an object or collection of objects	NIL	If supplied, then interactor modifies this object rather than return from :start-where
:illegal	6.1.3	list of objects	NIL	Which items are now illegal
:running-where	6.1.4	list of objects	*based on :start-where*	Where the mouse must be for the interactor to be active while running
:continuous	6.1.5	T or NIL	T	Whether the interactor is continuous or one-shot
:move-to-next-p	6.2.1	T or NIL	T	Does interim-feedback move to other objects when mouse moves?
:how-set	6.2.1	one of: :set :clear :toggle :list-add :list-remove :list-toggle	:set	Object added, removed, or toggled in selection set. Also controls how may objects are selected
:repeat-rate	6.2.1	NIL or a number	NIL	Speed menu-interactor repeats if mouse button held down
:attach-point	6.2.2	one of :nw :n :ne :e :se :s :sw :w :center :where-hit 1 2	:where-hit	Where the object should move or grow from with the mouse. 1 and 2 are the end points of a line

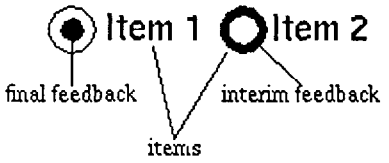
Fig. 2. All of the Interactor slots (parameters) that can be customized by programs. The "section" column shows which section of this paper it is discussed in.

Slot	Section	Parameter type	Default value	Comment
:line-p	6.2.2	T or NIL	NIL	Whether the object is defined by end-points or a bounding box
:grow-p	6.2.2	T or NIL	NIL	Whether to grow or move object
:min-width	6.2.2	number	0	Minimum width
:min-height	6.2.2	number	0	Minimum height
:min-length	6.2.2	number	0	Minimum length for a line
:abort-if-too-small	6.2.3	T or NIL	T	If smaller than minimum, abort or just use the minimum value
:flip-if-change-side	6.2.3	T or NIL	T	Flip over if move to upper left
:center-of-rotation	6.2.4	coordinate	center of the object	Where to measure rotations around for an angle interactor
:key-translation-table	6.2.5	a table	default editing table	Translation table for text interactor editing operations
:final-function	6.4	a function	NIL	Application function to call when interactor complete
:active	6.8.2	T or NIL	T	Whether interactor can run or not
:waiting-priority	6.8.3	a priority level	standard-priority	Level while waiting to start
:running-priority	6.8.3	a priority level	high-priority	Priority level while running
:window	6.8.4	a Garnet window	*must be supplied*	Window or windows that the interactor runs in
:start-action	6.8.5	a function	*based on type*	Procedure executed when start
:running-action	6.8.5	a function	*based on type*	Procedure executed for each mouse movement when running
:stop-action	6.8.5	a function	*based on type*	Procedure executed when stop
:abort-action	6.8.5	a function	*based on type*	Procedure executed when abort
:outside-action	6.8.5	a function	*based on type*	Procedure when go outside of running where
:back-inside-action	6.8.5	a function	*based on type*	Procedure when return inside

Fig. 2—*Continued*

6.2.1 *Menu Interactor.* Menu Interactors are used, not surprisingly, mostly for menus. In general, they can be used to choose one or more item from any set of items. A final feedback object (e.g., a reverse video rectangle) is often shown over the selected item. Typically, an interim feedback object becomes visible when

Fig. 3. Two Macintosh-like radio buttons, with the parts labeled as to the roles they play in the Menu Interactor. The interim feedback appears when the mouse button is pressed, and it goes away and the final feedback appears when the mouse button is released.



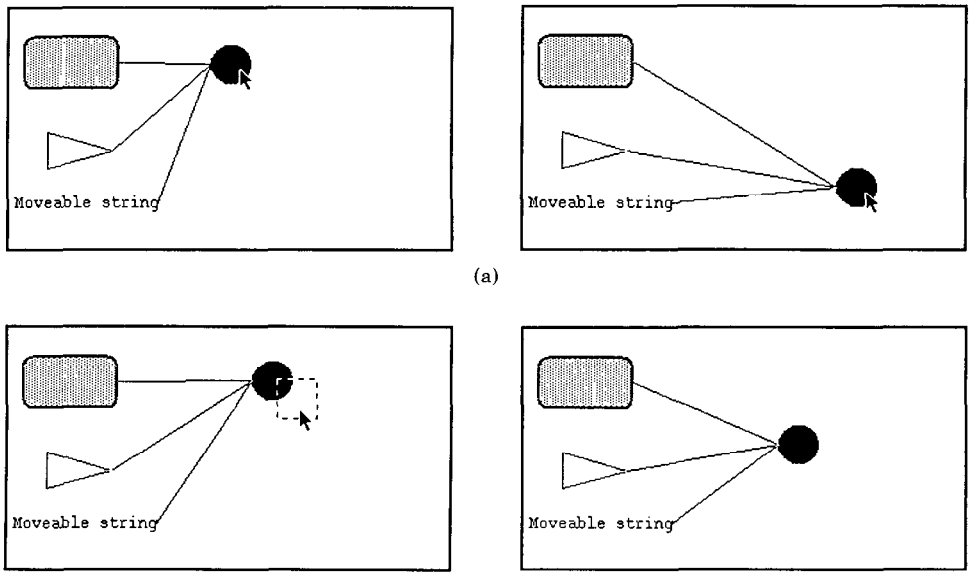


Fig. 4. Two options for the feedback with Interactors. On the top row (a), the object follows the mouse directly while the button is pressed. On the bottom row (b), the user presses a button over the circle, and an outline box follows the mouse until the button is released, when the circle jumps to the final position.

the start event happens, and it moves from item to item as the mouse moves. If the :Move-to-Next-p parameter is NIL, however, then the feedback will not move from one item to another. In this case, the user is allowed to point to any of the set, but moving away from that item causes it to be deselected; a different item will not be selected. The user must release the mouse button and press again to move the feedback to a different item. This is how radio buttons and check boxes work on the Macintosh. When the stop event happens, the final feedback (if any) is displayed. Figure 1 shows various menus.

A parameter determines how many items should be selected, and whether the new item should be added or removed from the selection set. For example, if the left mouse button should cause items to become selected, but the right button should cause them to be *de*selected, then the programmer can use two Interactors with different start events. One Interactor would add the item to the selected set, and the other would remove it. A more common case is that a single Interactor

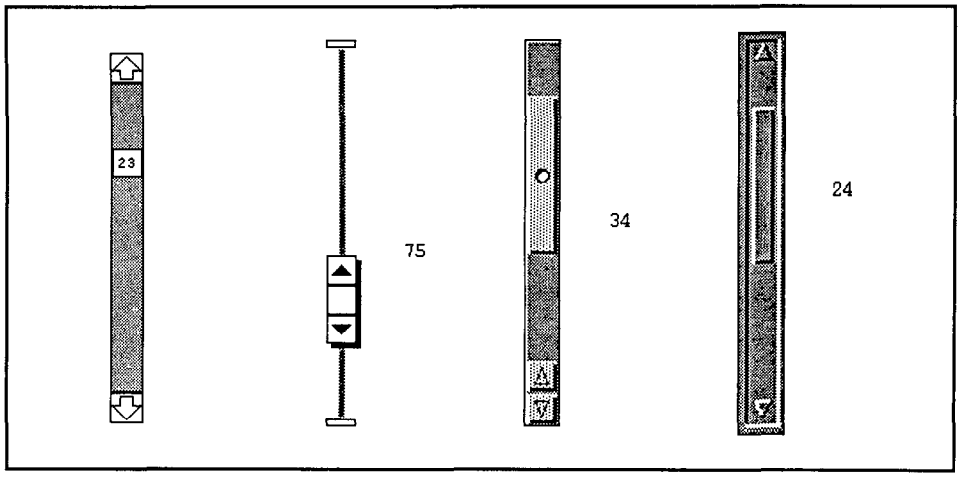


Fig. 5. The same type of interactor can handle different graphic looks. Here, a Move-Grow Interactor for each indicator and a Menu Interactor for the arrow buttons are handling scroll bars that look like those in the Macintosh, OpenLook, NeXT, and OSF/Motif.

will toggle whether the item is in the selected set, and in this case the :how-set parameter would be "toggle."

Menu Interactors can also be used for on-screen buttons, such as radio buttons and check boxes (see Figure 6), and for stand-alone buttons, such as command buttons or the arrows in a scroll bar. Some buttons, such as the arrows on Macintosh scroll bars, repeat their operation while the mouse button is held down. The :repeat rate parameter to the menu-interactor controls this behavior in Garnet.

Menu Interactors are also used to select items in a grahics editor. The programmer simply creates a list of all the items that can be selected and associates a Menu Interactor with them. The feedback indicating which item is selected can be arbitrary, of course, but typically a reverse video rectangle or a set of "handles" would be displayed (see Figure 7).

6.2.2 *Move-Grow Interactor.* The Move-Grow Interactor is used to move or change the size of an object or one of a set of objects with the mouse (see Figure 8). This is quite a flexible Interactor which handles many different behaviors including: moving the indicator in a slider, changing the size of a bar in a thermometer, changing the size of a rectangle in a graphics editor, changing the position of a text string, changing an end-point of a line, and changing the position of a line while keeping its length and slope fixed.

Parameters determine whether the object is moved or resized, an optional minimum size, and which part of the object is connected to the mouse (for example, a rectangle might be grown from any edge or corner). Another parameter determines if there is feedback to show how the object will change (e.g., a hair-line box), or if the object itself changes with the mouse (see Figure 4).

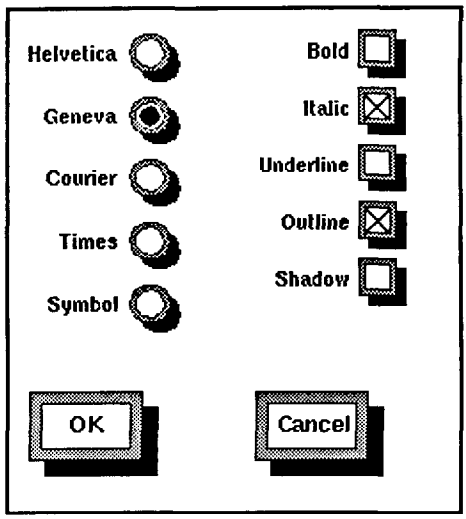


Fig. 6. A Menu Interactor can be used for radio buttons, check boxes, command buttons, and so forth. These examples are from the Garnet widget set.

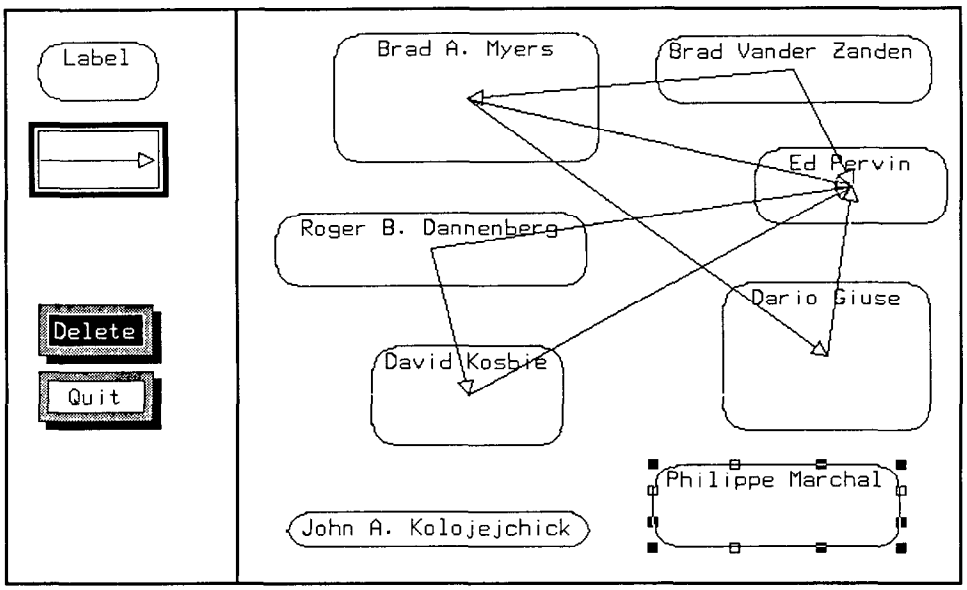


Fig. 7. A simple boxes-and-arrows editor created using Garnet. A Menu Interactor was used to cause objects to become selected and display the selection "handles." Pressing on a handle causes a Move-Grow Interactor to start, which changes the size or position of the selected object (pressing on the white handles moves the object and pressing on the black handles changes the size). Pressing on the strings allows them to be edited, which uses a Text Interactor. An arbitrary number of new boxes and arrows can be added, and their initial position is specified using the mouse (implemented with a New-Point Interactor). Any existing box or arrow can also be selected and deleted. This entire editor was implemented in about four hours using the Garnet toolkit (without using Lapidary).

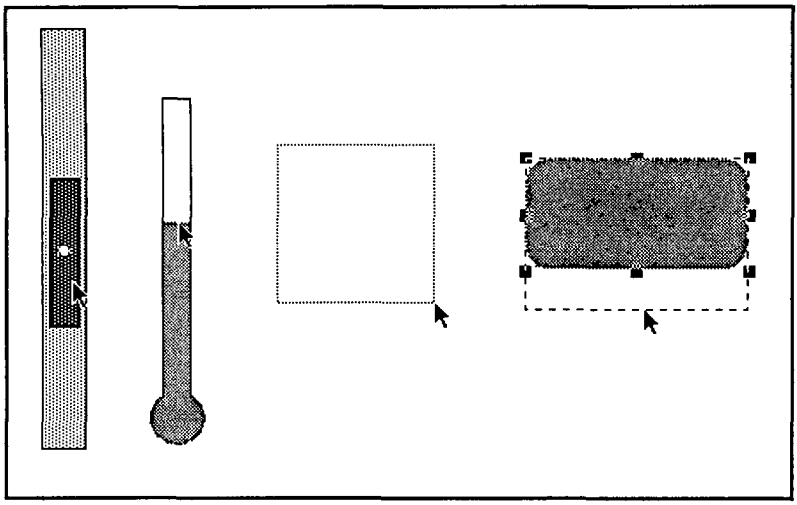


Fig. 8. The Move-Grow Interactor can be used to change the position of an indicator in a scroll bar, to change the size of a thermometer bar, to move or change an object's size in a graphics editor, and so on.

Since there are often a number of objects that might be moved or resized, the Interactor can take a set of objects as a parameter. In this case, the object that is changed is the object under the mouse when the start event happens.

6.2.3 *New-Point Interactor.* The New-Point Interactor is used when one, two, or an arbitrary number of new points are desired from the mouse. This is typically used when the user is creating a new graphic object such as a rectangle, line, polygon, and so forth. It can also be used if the programmer wants to get a region of the screen to find out which objects are inside (to cause all objects inside a bounded region to become selected, for example). A feedback object (for example, a rubberband line or rectangle) will usually be drawn based on the points specified. Parameters to the Interactor determine how many points are desired, and if the number is 2, whether the mode is bounding box or line (the computation of the minimum sizes is different). New-Point Interactor is different from the Move-Grow Interactor because there is no existing object to be changed; a new object may be created based on the results of the Interactor.

New-Point Interactors have built-in mechanisms for handling a number of common constraints on the values. For example, a minimum size for rectangles or minimum length for lines can be specified. If a bounding box is desired, another parameter determines whether the new box is allowed to "flip over" the initial point or not. That is, if you press and then begin dragging toward the bottom right, a rectangle is drawn. If you then move above and to the left of the original point, this parameter determines whether the rectangle simply flips over, as in Macintosh MacDraw, or whether it stays at its minimum size below and to the right of the initial point, as Macintosh windows do.

6.2.4 *Angle Interactor.* This is used to calculate the angle that the mouse moves around some point. It can be used for circular gauges or for "stirring motions" for rotating [6]. A parameter determines the center of rotation. Figure 9 shows two objects that use Angle Interactors.

6.2.5 *Text Interactor.* It is *not* one of the goals of Garnet to deal with text editors. However, it is useful to be able to input text strings on the screen. The Text Interactor can be used to input a one-line or multiline string of text, while allowing editing of the string. The intension is that this be used for string entry in text forms, for file names, object names, numbers, and labels for pictures.

The Text Interactor provides a full set of editing operations, including:

—inserting characters at the cursor position,
—moving the insert cursor around in the string (to the beginning or end, forward and backward, and up and down if it is a multiline string),
—deleting the character or word before or after the cursor, or the entire string,
—changing the cursor position with the mouse by pointing at the desired position in the string, and
—copying text to and from the window manager's cut buffer.

By default, these are bound to keys similar to the EMACS editor command set, but the programmer has complete control over the bindings. In addition, the programmer can easily add or remove editing functions.

6.2.6 *Trace Interactor.* This is used to get all of the points the mouse goes through between start and end events, as is needed for free-hand drawing. (This Interactor is the only one that has not been implemented yet.)

6.3 Protocol with Graphic Objects

This section discusses how Interactors interface to graphic objects, and Section 6.4 discusses how Interactors can be connected to application programs.

6.3.1 *Modifying the Graphics.* In order to keep the Interactors independent of the specific graphics, all modifications are done indirectly. For example, the Menu Interactor specifies which object is selected by setting a special slot in that object, called :selected. If the programmer wants the font of the selected object to change (as in Figure 1f), then a *constraint* is set up between the font slot of the string object and the special :selected slot. Constraints are implemented using *formulas.* In the Garnet object system, any slot can either contain a normal value (like a number) or a formula that calculates the value. Formulas can be arbitrary Lisp expressions, with references to slots in objects using the special form (gv ⟨*obj-name*⟩ ⟨*slot-name*⟩), where gv stands for "get value." When the slot is in the same object as the formula, a special form (gvl ⟨*slot name*⟩) can be used, which is the same as (gv :self ⟨*slot-name*⟩). Therefore, the font slot of each menu item might be as follows:

```
:font (formula (if (gvl :selected)   ; If I am selected
                   bold-font         ; then use bold font
                   regular-font))    ; else use regular font
```

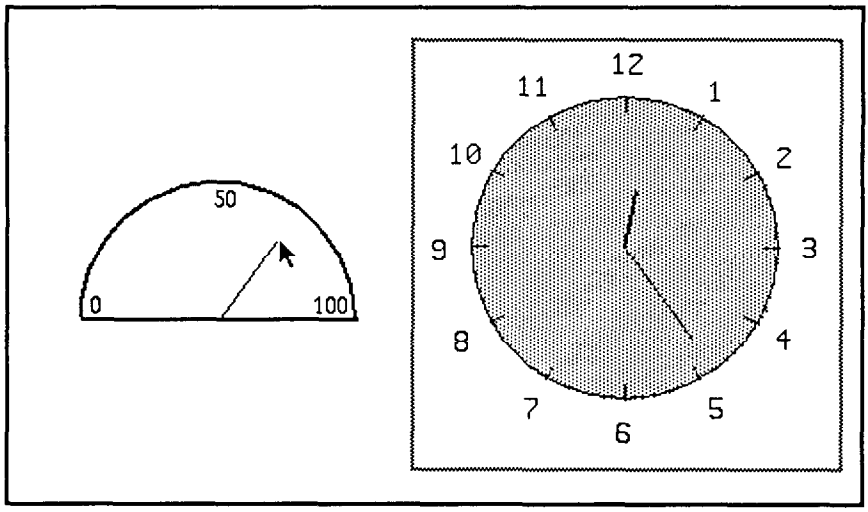


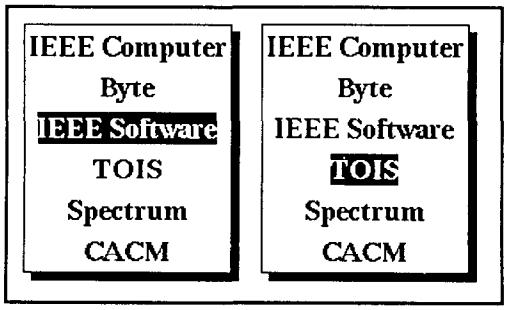
Fig. 9. The Angle Interactor can be used for circular gauges or clocks.

Similarly, if a plus sign is to appear next to the selected items (as in Figure 1c), then the :visible slot of the various objects that make up the plus sign would be tied with a constraint to the :selected slot.

6.3.1.1 *Menus.* The Menu Interactor uses a number of different mechanisms to communicate which items are selected. The programmer can pick whichever is most convenient for his particular application. First, the Interactor sets the :interim-selected slot of the graphic object the mouse is currently over while the interaction is running (for the interim feedback). Then, the Interactor sets the :selected slot as described above for the final object the mouse is released over (to control the final feedback). Also, if the item that is selected is part of an *aggregate* (or collection) object (for example, menu items are usually part of a menu aggregate), then the :selected slot of the aggregate object is set with a list of all the objects that are selected. Finally, if there is a feedback object, then a special slot, called :obj-over, in that feedback object is set with the item that is currently selected. Constraints in the feedback object can then indirectly reference through this slot to put the feedback over the selected object. This slot therefore operates like a pointer variable. For example, the reverse video rectangle object of Figure 1e uses constraints to tie its position and size to whatever object it is over. The formulas to do this are shown in Figure 10.

Of course, anywhere a single object can be used for feedback, an aggregate object containing an arbitrary collection of graphic objects can be used instead. The protocol allows the Interactor to be unaware of this.

6.3.1.2 *Moving and Growing.* The Move-Grow Interactor also modifies objects indirectly, through the use of a special :box slot. Based on the position of the mouse, the Interactor sets the appropriate values into the :box slot. The graphical object must be defined with constraints between its left, top, width, and height




```
(create-instance 'feedback-box Rectangle   ; Create a rectangle, call it feedback-box.
  (:obj-over NIL)   ; The interactor sets this slot with the object that should be highlighted.
  (:visible (formula (gvl :obj-over)))   ; I am visible if there is an object in :obj-over.
  (:left (formula (gvl :obj-over :left)))   ; The size and position is the
  (:top (formula (gvl :obj-over :top)))   ; same as that of the object
  (:width (formula (gvl :obj-over :width)))   ; stored in the :obj-over slot.
  (:height (formula (gvl :obj-over :height)))
  (:draw-function :xor))   ; xor this rectangle
```

Fig. 10. The Interactor can simply set the :obj-over slot, and the feedback box will jump to the appropriate object due to the constraints. When the interaction is complete, the Interactor simply sets :obj-over to be NIL and the box disappears.

slots and the appropriate position in the box slot. This allows the object to apply filtering on the values before they are used to determine the position. For example, a vertical scroll bar will have a formula that uses the Y value from the :box slot and ignores the x value:

```
(create-instance 'indicator-box Rectangle   ; create a rectangle to use as the feedback
                                            ; in a scrollbar.
  (:box '(10 20 40 40))   ; The initial (left top width height). This is slot is set by the
                          ; Interactor.
  (:left 10)   ; The left of the Indicator doesn't change.
  (:top (formula (second (gvl :box))))   ; The top depends on the mouse, so usewhat
                                         ; the Interactor sets into the :box slot.
  (:width 40)   ; The size doesn't change.
  (:height 40))
```

6.3.1.3 *Others.* The Text Interactor changes special slots to signify the value of the string and the current cursor position within the string. The Angle Interactor sets an :angle slot. New-Point Interactor sets either a :box or :point-list slot, depending on whether it is entering a bounding box or set of lines. The Trace Interactor sets a :point-list slot.

6.3.2 *Querying Objects.* The Interactors must determine whether the mouse point is inside of the graphic object, in order to determine which object should be selected or operated on. Since all graphics are respresented as objects, this simply requires sending the Is-This-Point-Inside-You message to the object. Again, using this object-oriented approach, the Interactor does not need to know anything about the graphics that serve as the items to be chosen.

6.4 Connection to Application Programs

Often, the desired outcome of an interactor is a change to the graphics. In this case, the protocol described in the previous sections is sufficient. If properties of additional graphic objects should be affected also, these can usually be handled with constraints.

The object system in Garnet can be used by application objects also (it is not limited to graphic objects), so often the application's own data structures can be connected to the graphic objects by constraints. For example, if a gauge is associated with a temperature value in the application's data space, then constraints can be used to insure that when an Interactor modifies the gauge display, the temperature value is also changed.

When constraints are not sufficient, however, there is an additional mechanism for notifying application programs. Each interactor can take a call-back procedure in the :Final-Function slot, which will be called when the interaction is complete. The parameters that are passed to this function are specific to the type of interactor. For example, the Move-Grow Interactor calls the Final Function with the object being changed and the final position and size of the object.

One case when the Final Function is almost always needed is for the New-Point Interactor. Here, there is no existing object to modify, so the application will usually provide a function to create a new object.

6.5 Some Examples

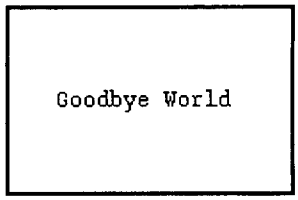
The syntax for creating and modifying Interactors is the same as for all other objects in Garnet, since the Interactors are objects themselves. When an Interactor is created, values for slots can be specified. Any unspecified slots inherit the default values, which are often sufficient. For example, to create an interactor to move a rectangle, the following can be used:

```
(create-instance 'mymover move-grow-interactor
  (:start-where '(:in moving-rectangle))
  (:start-event :middledown)   ; the middle mouse button
  (:window mywindow))
```

This Interactor starts operating when the middle mouse button is pressed over the object named moving-rectangle, and will move that rectangle until the middle mouse button is released (the default stop event is calculated from the start event). If the programmer wanted to continue moving until the middle button was pressed a second time, the code would be changed to the following:

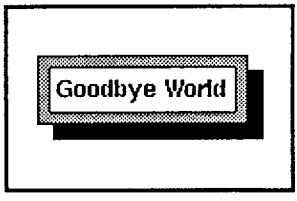
```
(create-instance 'mymover move-grow-interactor
  (:start-where '(:in moving-rectangle))
  (:start-event :middledown)   ; the middle mouse button
  (:stop-event :middledown)    ; stop on middle down also
  (:window mywindow))
```

As another example, Figure 11 is a complete, minimal "Goodbye World" program, that creates a window with an on-screen button that causes the window to go away when pressed with the left button.




```
;;; First create a window and a text object; see the manual [22] for a complete explanation.
(create-instance 'mywindow interactor-window (:left 100)(:top 10)
  (:width 150)(:height 100)(:title "My Window")
  (:aggregate (create-instance 'myagg aggregate)))
(create-instance 'mytext text (:String "Goodbye World")
  (:left 25)(:top 40))
(add-component myagg mytext)
(update mywindow)
;;; Now add the Interactor
(create-instance 'killer Menu-Interactor
  (:window mywindow)
  (:start-where '(:in mytext))   ; Operate on the mytext object.
  (:continuous NIL): Happen immediately on the downpress (see Section 6.1.5).
  (:final-function #'(lambda (inter final-obj-over)(destroy mywindow))))   ; Kill the window.
```

Fig. 11. The complete "Goodby World" program created from scratch. Of course, the Garnet object system automatically handles all intialization and overhead, including refreshing the string if the window is covered and uncovered.




```
;;; Create a window and a button object.
(create-instance 'mywindow interactor-window (:left 100)(:top 10)
  (:width 150)(:height 100)(:title "My Window")
  (:aggregate (create-instance 'myagg aggregate)))
(create-instance 'mybutton labeled-button (:String "Goodbye World")
  (:left 15)(:top 25)
  (:selection-function
  #'(lambda (gadget-object value))(destroy mywindow))))   ; Kill window
(add-component myagg mybutton)
(update mywindow)
```

Fig. 12. The complete "Goodbye World" program using a predefined Garnet labeled button. The Interactor is built into the button widget.

Typically, a programmer would not create this from scratch as done here, but would use a button widget from the Garnet widget set instead (see Figure 12). This widget itself is implemented using a Menu Interactor. In fact, using the Lapidary user interface construction tool (described in Section 8), either of these interfaces could be created without writing any code by hand.

The 13-line program of Figure 11 might be compared to the 40-line program needed using straight CLX in CommonLisp, or the 60-line program needed to handle this using straight Xlib in C.

The 8-line program of Figure 12 should be compared with the 30-line program needed using the X toolkits in C (Xtk, Motif, etc.) [13].

6.6 The Internal State Machine

All Interactors run the same simple state machine (see Figure 13) that handles starting, stopping, aborting, and suspending while outside the active region. The parameters to the Interactor determine what events cause the transitions. Unlike transition network UIMSs, such as [11], the designer does not explicitly deal with this state machine.

The particular "-action" routines shown in Figure 13 (such as start-action, stop-action, etc.), are specific to the particular type of interactor. It is also possible for the programmer to supply custom action routines, as described in Section 6.8.5, but this is not usually necessary (see Section 7).

Different Interactors can be running their individual state machines in parallel, which is how multiple threads can be handled.

As an example, consider the way the Macintosh Finder highlights icons while an object is being dragged around if that icon can accept the object being dragged. Two Interactors would be used to implement this in Garnet. The first would be a Move-Grow Interactor to drag around the object. The feedback graphics (that follow the mouse) would be calculated using a constraint based on the outline of the object that is being moved. The second Interactor would be a Menu Interactor that would highlight the object that the mouse was over (Menu Interactors can take an arbitrary list of objects as the items to choose over, and the highlight moves from one item to another as the mouse moves). The application would be responsible for giving this Interactor a list of the appropriate objects that could accept the object being moved, or this could be calculated using a constraint which was based on properties of the icons and the object being moved. Both Interactors would operate *in parallel* and therefore receive the same mouse events. This is also a good example of how Garnet can handle *semantic feedback.*

6.7 Use of Interactors in Widgets

In the Garnet object system, any collection of objects can be used as a *prototype* for other instances. This prototype can contain graphic objects, Interactor objects, and application objects. The create-instance call creates an instance of each of the objects in the prototype. For example, in Figure 12, the create-instance of labeled-button creates three rectangles, a string, and a Menu Interactor.

When creating instances, it is also possible to override any slots or objects. In Figure 12, the :string, :left, :top, and :selection-function slots of the button are overridden. In the same way, the programmer can override some slots of the Interactor, for example, to change which mouse button causes the button to operate. Interactors (and graphic objects) can even be added or deleted from the particular instance (so the drop shadow could be removed, for example). If the programmer makes a number of changes in an instance, then *that* instance could be used as a prototype for further instances, since there is no distinction between prototypes and instances in Garnet.

6.8 Advanced Features

The techniques described above are sufficient for most applications of interactors. However, some user interface styles require additional functionality. This section discusses some of the advanced features that are provided.

6.8.1 *Formulas for Interactor Parameters.* Because any slot in any Garnet object can contain a formula (constraint) instead of a regular value, slots in Interactors can contain formulas also. In order to make this convenient, there are a number of special slots set by the Interactor that can be referenced by the programmer's formulas. These include the object that is being operated on and the event that started the interactor.

As an example, the following Interactor decides whether to grow or move an object based on which mouse button is used to start it. The Interactor modifies whichever object in the collection called all-objs-aggregate the mouse is first pressed over.

```
(Create-instance 'move-or-grower Move-Grow-Interactor
  (:start-event '(:leftdown :rightdown))  ; Either left or right mouse button.
  (:start-where '(:element-of all-objs-aggregate))  ; Start if press over any element.
  (:grow-p (formula (eq :rightdown (gvl :start-char))))  ; Grow if the start-char is
                                                         ; right mouse button, else
                                                         ; move.
  (:window mywindow))
```

6.8.2 *Modes.* Many user interfaces have different modes, and different Interactors should be available depending on the current mode. For example, in Apple Macintosh MacDraw, when the mouse button is pressed in the drawing area, an existing object will be selected or a new object will be created, depending on the mode selected in the drawing palette.

In Garnet, modes can be supported in various ways. Of course, appropriate Interactors can simply be created and destroyed depending on the mode, but this is very inefficient. Alternatively, there is a special slot of Interactors called :active, and this can be set based on the mode to enable and disable the Interactor (if :active is NIL, the Interactor will not start even if the start event happens). The value in this slot can be a formula that depends on whatever value determines the mode.

6.8.3 *Priority Levels.* When an Interactor is running (between the start and stop events), it is usually given a *higher priority* than other Interactors. This means that it will be sent events before other Interactors. If it accepts the events, then they are not sent to lower priority Interactors. Therefore, for example, if the stop event for a running interactor is the same as the start event for another Interactor, the running one will get the event by default.

This mechanism is entirely under the control of the programmer, however, Any number of Interactors can all get the same event, and they can all operate on it in different ways. For example, pressing the left mouse button while the cursor is over a text object may start both a Menu Interactor which shows the object as selected, and a Text Interactor which allows it to be edited. This can be achieved by simply putting the two Interactors at the same priority level. Another example was given in Section 6.6.

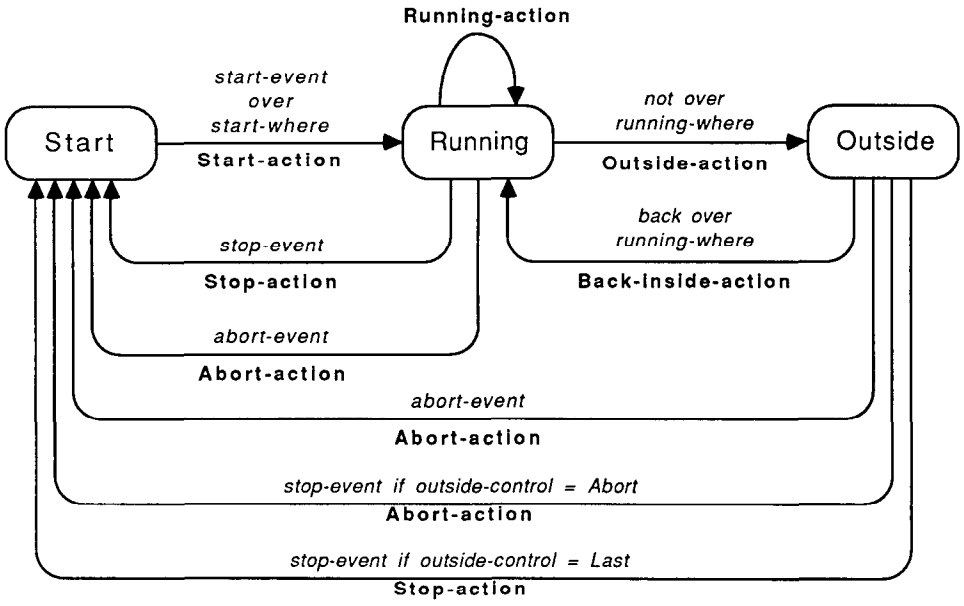


Fig. 13. Each Interactor runs the same state machine to control its operation. The start-event, stop-event, and abort-event can be specified (see Section 6.1.1), as can the various -action procedures (Section 6.8.5). Where the mouse should be for the Interactor to start (start-where), and where it should run (running-where) can also be supplied as parameters (Sections 6.1.3 and 6.1.4). The outside-control parameter determines where the interaction is aborted when the user moves outside, or whether the last legal value is used (Section 6.1.4). There are default values for all parameters, so the programmer does not have to specify them. In addition to the transitions shown, Interactors can be aborted by the application at any time.

Priority levels can also be used to implement modes. All of the Interactors in a particular mode can be put into the same priority level, and then the priority level as a whole can be enabled or disabled.

6.8.4 *Multiple Windows.* Interactors can operate over multiple windows. This allows an object to be dragged from one window to another, or to have various menu items in different windows. It is worth noting that the widgets and objects in Garnet are not usually windows, as in Xtk and some other toolkits, but Garnet allows an application to have as many top level or subwindows as desired.

To create a multiwindow Interactor, it is only necessary to set the :window slot with a list of windows, or the special value T which means to use all Garnet windows.

6.8.5 *Custom Action Routines.* Even with all of the flexibility described above, there are still a few cases in which the programmer needs more control. Therefore, the specific action routines that define the standard behaviors of the Interactors can be overridden in particular Interactor instances. Figure 13 shows the action routines that all Interactors support. The standard routines for these handle such things as turning on and off the feedback graphics, having the feedback follow the mouse appropriately, and updating the actual object when the interaction is complete.

The Interactors reference manual describes how to supplement or replace each of these routines for each type of interactor. This has proven to be necessary in only a few cases, such as when the programmer wants a special action, such as a beep, when the mouse goes outside or if the Interactor aborts (see Section 7). Even when custom -action procedures are needed, using Interactors is still much easier than handling the raw window manager events because the Interactors handle the control of sequencing and provide a layer of device and window manager independence.

7. COVERAGE OF INTERACTORS

It is important to emphasize that Interactors handle *all* input for the user interface; both for the toolkit components (menus, buttons, scroll bars, etc.) and for the application-specific objects (icons, rectangles, arrows, etc.).

As shown in Section 5, the choice of the six types is based on a taxonomy of interaction styles, but clearly, new styles can be invented. Therefore, an important question is whether the existing Interactors can handle *all* possible interaction techniques.

In fact, the Interactors can be configured to provide the full information about all raw input events that occur. For example, the following interactor will call the application function for every mouse and keyboard event and every incremental mouse movement event. Notice that no calls to the underlying window manager are necessary:

```
(Create-instance 'everthing Move-Grow-Interactor
  (:start-event T)     ; Any event.
  (:stop-event NIL)    ; Never stop
  (:start-where T)     ; Everywhere
  (:window T)          ; All windows.
  (:start-action #'ApplicationFunc)
  (:running-action #'ApplicationFunc))
```

The input events are available to the application function. Therefore, Interactors can clearly support any technique that can be implemented with a mouse and a keyboard, so the question is not whether Interactors *can* support some technique, but rather whether it is *convenient* to implement with the existing Interactors.

It would be nice to be able to refer to a comprehensive taxonomy of interaction techniques, and classify how difficult implementing each would be with Garnet. Unfortunately, attempts to create such a taxonomy have not been successful [23].[6] Therefore, the range of Interactors must be presented informally.

In the experience of the many users of Garnet, Interactors successfully allow simple interaction tasks to be created very easily, and more complex tasks to be handled with a corresponding increase in programming. The following list gives some examples of how much effort is required for various interaction techniques with the current design of Interactors:

—*Techniques implemented by simply supplying values for slots of existing interactor types*: Simple menus and palettes, "floating" menus (where the items pretend to move in 3-D when pressed), radio buttons, check-boxes, selecting

[6] Note that taxonomies of input devices [1, 3] are not relevant here; we need a taxonomy of ways of using devices to interact with graphical objects.

application objects, moving application objects, growing an object from the nearest side or corner to where the mouse is pressed, moving a scroll bar or slider indicator, incrementing a scroll bar when an arrow button is pressed, moving the indicator of a circular gauge, giving commands by hitting single keyboard keys (e.g., command-x, control-p), editing the text label if the user clicks on it with a mouse, start editing the selected text when a keyboard character is hit, executing the default menu command if the user hits the "return" key, highlighting valid objects as the mouse moves over them.

—*Techniques best handled with formulas in slots of Interactors*: deciding whether to move or grow an object depending on the mouse button pressed, deciding whether to more or grow depending on which selection handle is hit, moving one object when the mouse is pressed over a different object (e.g., pressing on a sub-part or a selection handle), having one menu determine which other interactors are active or not (to control different modes), calculating the appropriate minimum height depending on the selected object.

—*Techniques requiring custom action procedures*: slide-out and pull-down submenus, gridding, gravity, providing an initial value for the string when text editing.

—*Techniques best handled with new Interactor types*: gesture recognition, character recognition, new hardware input devices. There is every indication that the Interactor paradigm presented here expands gracefully to cover these techniques.

—*Techniques outside the range of Garnet*: Sophisticated text editing.

8. NONPROGRAMMING INTERFACE

The Garnet project includes a high-level graphical tool for creating user interfaces, called Lapidary [20]. This kind of tool is sometimes called an "Interface Builder" or User Interface Management System (UIMS). Lapidary allows widgets and application-specific graphic objects to be created from scratch or copied from a library. To create an object from scratch, the designer draws pictures (using primitives such as rectangles, lines, and circles) of what user interface elements should look like, and then specifies the behavior using dialog boxes.

Since Interactors allow behaviors to be attached to graphic objects by simply supplying values for parameters, Lapidary is able to attach behaviors to the graphics by providing dialog boxes in which the designer can specify the parameters (see Figures 14 and 15). In Figure 14, the designer is creating a menu from scratch (Figure 14b), and adding behavior using a dialog box (Figure 14e). This dialog box supports Menu interactors, and allows the items in the editor window to be selected and used as the list of items to choose among (here, othello menu), the interim feedback (here, not used), and the final feedback (here, the checkmark). Many of the other parameters to the interactor are available in the main dialog box or one of the subdialogs which can be brought up by hitting the Details or Extra Actions buttons. Therefore, the programmer can design the behavior *by demonstration* on example objects, which makes it much easier to achieve the desired result.

An interesting feature of Lapidary is that it automatically modifies the graphic objects so that they conform to the standard protocol for Interactors. For example,

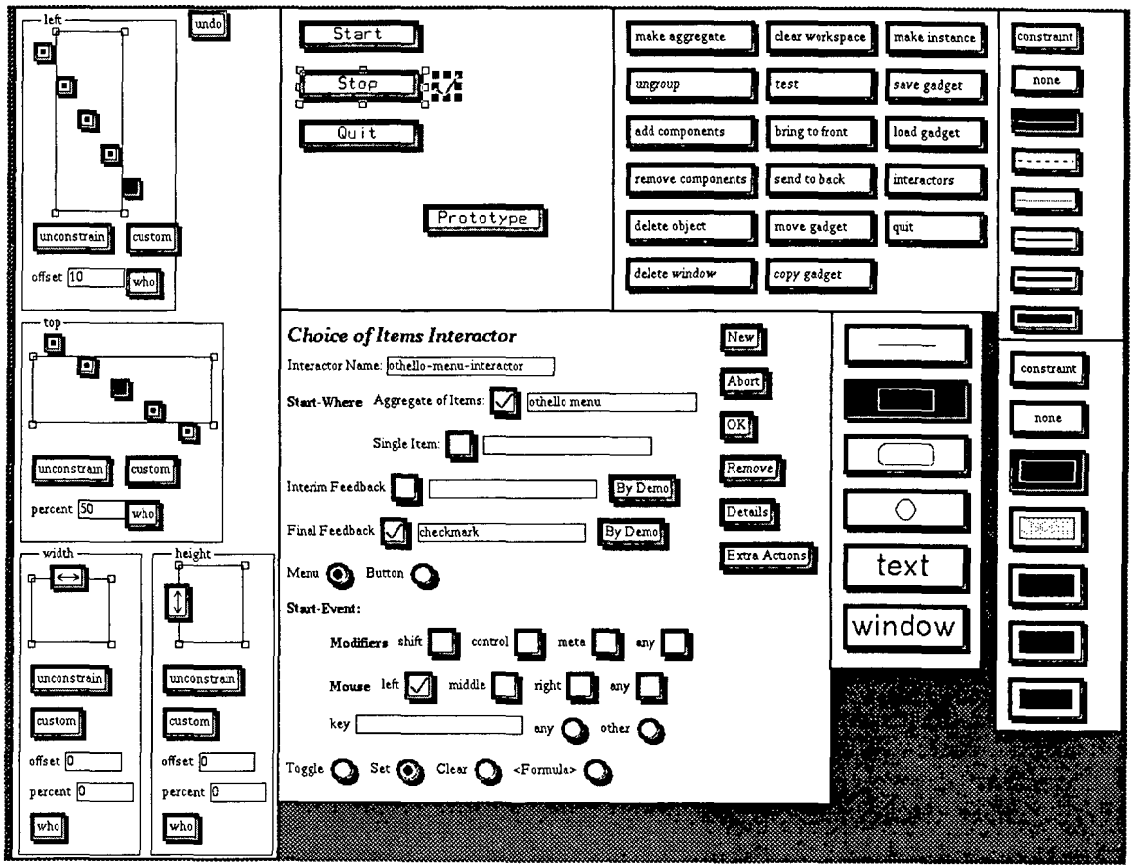


Fig. 14. An example of Lapidary in action [20]. The work window (b) contains a prototype and three buttons made from it. The iconic constraint menu (a) is being used to constrain the check mark to be at the right of the Stop button offset by 10 pixels, and centered vertically. Window (c) contains the main Lapidary commands. The menus on the right determine (f) the type of the next object to be created, (d) the line style, and (g) the filling-style. Window (e) is a dialog box for specifying the behavior of the menu being created, and it shows that the check mark icon will be used as the "final feedback" to show which item is selected when the left mouse button is pressed. Of course, this entire interface (including all buttons, etc.) was created using Garnet.

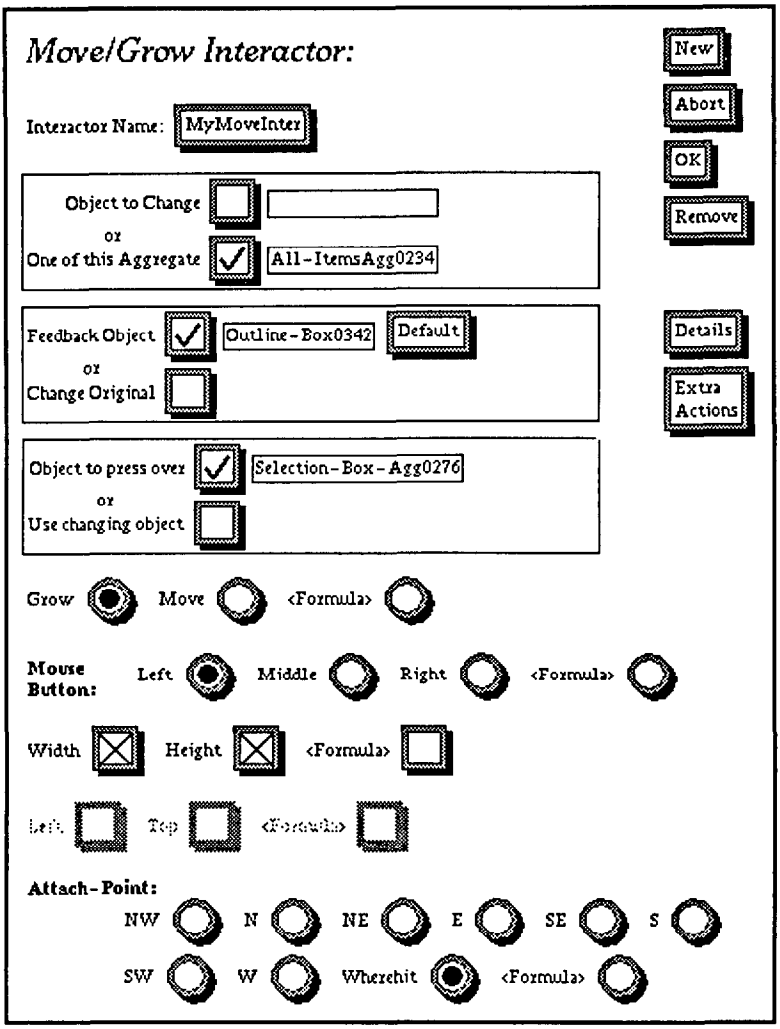


Fig. 15. The dialog box used to specify Move-Grow Interactors in Lapidary.

in Figure 14, the designer constrained the check mark to be centered at the right of the Stop button. When the check mark is specified as the final feedback object, however, Lapidary generalizes the constraints on the check mark so that they instead refer indirectly to whatever object is selected. Since the protocol for each type of Interactor is well-defined, this is easy to do automatically. Note that the designer could have drawn any kind of object or even a collection of objects, and these could have been used as the final feedback. Lapidary would still be able to generalize the constraints appropriately; there is nothing special about the check mark.

9. DEBUGGING

Since Interactors provide an unconventional interface to input, it is useful to have some appropriate debugging tools to help find errors in programs. We have

provided a number of tools in the Garnet Toolkit [22]. These include:

—a tracing mechanism that shows what is happening with some or all interactors;
—procedures to tell which interactors will affect a particular graphic object or which objects a particular interactor will affect;
—techniques for determining which interactors will start on specified events, and what events will cause an interactor to start; and
—cleanup procedures that will reinitialize Interactors if they get into erroneous states.

10. STATUS AND FUTURE WORK

Currently, all of the Interactors except Trace Interactor have been implemented. The others have been extensively used in many different kinds of interfaces. There is a complete widget set, consisting of menus, buttons, scroll bars, gauges, and so forth as well as a number of application-specific objects built using the Interactors. The Garnet toolkit has been released and is being used by approximately 50 projects all over the world at this time. Interactors have proven to be an efficient, powerful and easy-to-use mechanism for implementing graphical interaction.

In the future, there are a number of issues that we want to address:

—How well will the Interactors paradigm work for input devices other than the mouse and keyboard? It seems clear that Interactors for simple devices such as physical knobs and switches will not be hard, but what about touch tablets and 3-D devices like a data-glove? Similarly, can Interactors handle new input techniques such as gesture recognition?
—How can various sub-menu options be provided more conveniently? Some explicit coding using the running-action procedure is necessary to make the Menu Interactor handle pull-down menus as in the Macintosh. We want this to be easier to specify.
—How can Interactors be used to support an Undo/Redo facility?
—A primary focus will be on ways to allow Lapidary to specify higher-level combinations of Interactors. For example, it is currently easy to draw the selection handles that show which object is selected in a graphics editor, and then a dialog box can be used to say that the handles should appear over any object the mouse as clicked on. However, it requires programming to specify that when the user presses on a selection handle, this causes the object to which the handle is attached to grow (as opposed to, say, growing the handle itself). We believe that these composite behaviors can be specified by demonstration also.
—Finally, it would be interesting to reimplement the Interactors model in a "conventional" object system and toolkit such as C++ or Xtk. This would help demonstrate the flexibility and appropriateness of the model as a future standard.

11. CONCLUSIONS

The Interactors in Garnet allow interactive behaviors to be specified and implemented separately from the graphics and from the application programs. They provide a high-level, but look-and-feel independent, interface to input events that makes programming and rapid prototyping easier. This is achieved by using a small number of Interactor object types that take as parameters the events to start and stop the interaction, the associated graphic objects, and other appropriate information. Interactors can handle all input for both standard widgets like menus and scroll bars, and application specific objects for a wide class of user interfaces.

In addition, this research has identified the primitive, device independent behaviors that can be encapsulated to handle graphical interfaces, and the appropriate parameters to these behaviors. This can form the basis for a new model for input that could be incorporated into future graphics standards, window managers, and toolkits.

ACKNOWLEDGMENTS

For help with this paper, I want to thank the referees, Brad Vander Zanden, Roger Dannenberg, Dario Giuse, Bernita Myers, and Dave Kosbie. I also want to thank the many people who are working on various parts of Garnet, and our users who provide helpful feedback.

REFERENCES

1. BUXTON, W. Lexical and pragmatic considerations of input structures. *Comput. Gr. 17,* 1 (Jan. 1983), 31–37.
2. BUXTON, W., AND MYERS, B. A study in two-handed input. Human Factors in Computing Systems, In *Proceedings SIGCHI'86* (Boston, Mass., April 1986), pp. 321–326.
3. CARD, S. K., MACKINLAY, J. D., AND ROBERTSON, G. G. The Design Space of Input Devices. Human Factors in Computing Systems, In *Proceedings SIGCHI'90* (Seattle, Wash., April 1990), pp. 117–124.
4. CARDELLI, L., AND PIKE, R. Squeak: A language for communicating with mice. Computer Graphics, In *Proceedings SIGGRAPH'85* (San Francisco, Calif., July 1985), pp. 199–204.
5. CARDELLI, L. Building user interfaces by direct manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 1988), pp. 152–166.
6. EVANS, K. B., TANNER, P. P., AND WEIN, M. Tablet-based valuators that provide one, two, or three degrees of freedom. Computer Graphics, In *Proceedings SIGGRAPH'81* (Dallas, Tex, Aug. 1981), pp. 91–97.
7. FOLEY, J. D., AND WALLACE, V. L. The art of natural graphic man-machine conversation. In *Proceedings IEEE 62,* 4 (April 1974), pp. 462–471.
8. FOLEY, J. D., WALLACE, V. L., AND CHAN, P. The human factors of computer graphics interaction techniques. *IEEE Comput. Gr. Appl. 4,* 11 (Nov. 1984), 13–48.
9. FOLEY, J. D., GIBBS, C., KIM, W. C., AND KOVACEVIC, S. A knowledge-based user interface management system. Human Factors in Computing Systems. In *Proceedings SIGCHI'88* (Washington, D.C. May, 1988), pp. 67–72.
10. HILL, R. D. Supporting concurrency, communication and synchronization in human computer interaction—The Sassafras UIMS. *ACM Trans. Gr. 5,* 3 (July 1986), 179–210.
11. JACOB, R. J. K. A specification language for direct manipulation interfaces. *ACM Trans. Gr. 5,* 4 (Oct. 1986), 283–317.

12. KRASNER, G. E., AND POPE, S. T. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *J. Object Oriented Program. 1,* 3 (Aug. 1988), 26–49.
13. MCCORMACK, J., AND ASENTE, P. An overview of the X toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct., 1988), pp. 46–55.
14. MEADS, J. The standards factor. *SIGCHI Bull. 19,* 1 (July 1987), 34–35.
15. MYERS, B. A. Visual programming, programming by example, and program visualization: A taxonomy. Human Factors in Computing Systems. In *Proceedings SIGCHI'86* (Boston, Mass. April, 1986), pp. 59–66.
16. MYERS, B. A. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.
17. MYERS, B. A. User interface tools: Introduction and survey. *IEEE Softw. 6,* 1 (Jan. 1989), 15–23.
18. MYERS, B. A. Encapsulating interactive behaviors. Human Factors in Computing Systems. In *Proceedings SIGCHI'89* (Austin, Tex., April, 1989), pp. 319–324.
19. MYERS, B. A. Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst. 12,* 2 (April 1990), 143–177.
20. MYERS, B. A., VANDER ZANDEN, B., AND DANNENBERG, R. B. Creating graphical objects by demonstration. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Va., Nov. 1989), pp. 95–104.
21. MYERS, B. A., GIUSE, D., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D., MARCHAL, P., AND PERVIN, E. Comprehensive support for graphical, highly-interactive user interfaces: The garnet user interface development environment. *IEEE Comput. 23,* 11 (Nov. 1990), To appear.
22. MYERS, B. A., GIUSE, D., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D., MARCHAL, P., PERVIN, E., MICKISH, A., KOLOJEJCHICK, J. A. The Garnet toolkit reference manuals: Support for highly-interactive, graphical user interfaces in Lisp. Tech. Rep. CMU-CS-90-117, Computer Science Dept., Carnegie Mellon Univ. March, 1990.
23. NIELSEN, J. Classification of dialog techniques. *SIGCHI Bull. 19,* 2 (Oct. 1987), 30–35.
24. OLSEN, JR., D. R. Ed. ACM SIGGRAPH workshop on software tools for user interface management. *Comput. Gr. 21,* 2 (April 1987), 71–147.
25. OLSEN, JR., D. R. Larger issues in user interface management. *Comput. Gr. 21,* 2 (April 1987), 134–137.
26. PALAY, A. J., HANSEN, W. J., KAZAR, M. L., SHERMAN, M., WADLOW, M. G., NEUENDORFFER, T. P., STERN, Z., BADER, M., PETERS, T. The Andrew toolkit—An overview. In the *Proceedings Winter Usenix Technical Conference* (Dallas, Tex, Feb. 1988), pp. 9–21.
27. PFAFF, G. R., ED. *User Interface Management Systems.* Springer-Verlag, Berlin, 1985.
28. *Draft Proposal American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS).* American National Standards Committee X3Hc/84-40, 1984.
29. SIBERT, J. L., HURLEY, W. D., AND BLESER, T. W. *An object-oriented user interface management system.* Computer Graphics. In *Proceedings SIGGRAPH'86* (Dallas, Texas, Aug. 1986), pp. 259–268.
30. SZEKELY, P. A., AND MYERS, B. A. *A user interface toolkit based on graphical objects and constraints. Sigplan Not. 23,* 11 (Nov. 1988), 36–45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA '88.
31. TANNER, P. P., MACKAY, S. A., STEWARD, D. A., AND WEIN, M. A multitasking switchboard approach to user interface management. Computer Graphics. In *Proceedings SIGGRAPH'86* (Dallas, Texas, Aug. 1986), pp. 241–248.
32. VANDER ZANDEN, B., AND MYERS, B. A. Automatic, look-and-feel independent dialog creation for graphical user interfaces. Human Factors in Computing Systems. In *Proceedings SIGCHI'90* (Seattle, Wash., April 1990), pp. 27–34.
33. WILSON, D. *Programming with MacApp.* Addison-Wesley, Reading, Mass., 1990.