Creating Graphical Interactive Application Objects by Demonstration

Brad A. Myers
Brad Vander Zanden
Roger B. Dannenberg

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Lapidary user interface tool allows *all* pictorial aspects of programs to be specified graphically. In addition, the behavior of these objects at run-time can be specified using dialogue boxes and by demonstration. In particular, Lapidary allows the designer to draw pictures of application-specific graphical objects which will be created and maintained at run-time by the application. This includes the graphical entities that the end user will manipulate (such as the components of the picture), the feedback that shows which objects are selected (such as small boxes on the sides and corners of an object), and the dynamic feedback objects (such as hair-line boxes to show where an object is being dragged). In addition, Lapidary supports the construction and use of "widgets" (sometimes called interaction techniques or gadgets) such as menus, scroll bars, buttons and icons. Lapidary therefore supports *using* a pre-defined library of widgets, and *defining* a new library with a unique "look and feel." The run-time behavior of all these objects can be specified in a straightforward way using constraints and abstract descriptions of the interactive response to the input devices. Lapidary generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration.

CR 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: User Interface Management Systems, Interaction, Object-Oriented Design, Direct Manipulation, Interaction Techniques, Programming by Example.

Introduction

Although a number of user interface management systems (UIMSs) support the specification and layout of menus, buttons, sliders and dialogue boxes, few help with the creation of objects used in application displays or with the specification of the interactive behaviors of these objects. The Lapidary user interface tool aims to help with all parts of user interface design by allowing *all* pictorial aspects of user interfaces to be specified using a direct manipulation graphical editor. In addition, Lapidary allows the run-time behavior of objects to be specified. Constraints can be defined graphically and by demonstration to describe how the objects relate to one another (e.g., that arrows stay attached to boxes), and the objects' responses to the mouse and other input devices can be specified or demonstrated using abstract *interactors* that embody particular kinds of interactive behaviors.

Lapidary supports both the creation and placement of *widgets* (also called interaction techniques or gadgets) such as menus, buttons, scroll bars, and dialogue boxes (also called forms or property sheets), as well as providing the ability to define application-specific graphical objects using the editor. For example, if the application involves creating and manipulating labeled rectangles connected by arrows (possibly, for a visual programming language [13] or a project scheduling program such as Apple Macintosh MacProject), the user interface designer can draw samples of these rectangles and arrows, and then describe their run-time behavior. Lapidary will automatically generalize from the graphics to create objects that the application and end user can create and manipulate at run-time. Therefore, Lapidary allows the *contents* of application windows to be defined graphically, in addition to the menus and palettes that surround the window.

The motivation for this project is the observation that a significant part of the effort when creating graphical, direct manipulation interfaces is designing the graphics and their behavior with respect to the input devices. Lapidary aims to allow the designer to specify these in the most direct, intuitive and straightforward manner possible: by graphical drawing and demonstration.

Lapidary is being created as part of the Garnet user interface development environment [15]. Garnet, which stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, is a comprehensive environment for creating graphical, direct manipulation [21] user interfaces. Garnet is being implemented in CommonLisp on top of the X window manager on IBM RT/PC computers using the Unix operating system. We also are considering porting Garnet to Display Postscript running on NeXT machines. Garnet currently has six components:

1. The Lapidary graphical editor,
2. An object-oriented programming system called KR (Knowledge Representation) [8],
3. An object-oriented graphics system called Opal (Object Programming Aggregate Layer) [17],
4. A constraint system called Coral (Constraint-Based, Object-Oriented Relations And Language) [23], which allows relationships among graphical objects to be specified easily and then maintained by the system,
5. Encapsulations of interactive behaviors independent from any graphical realization of that behavior [18], and
6. User interface toolkits containing several default sets of widgets.

A "Lapidary" is a workman who cuts, polishes and engraves precious stones, and here is a Lisp-Based Assistant for Prototyping Interface Designs Allowing Remarkable Yield. Lapidary, as well as the rest of Garnet, is now under active development and only part of the implementation is complete. This part is sufficient, however, to demonstrate that the ideas described in this paper are workable and powerful.

Related Work

Lapidary can be classified under the broad heading of User Interface Management Systems [1]. There have been a large number of previous systems that allow users to select from a pre-defined library of widgets, and place them on the screen. Some of these, including Menulay [5], Trillium [9], DialogEditor [6], vu [22] and the NeXT Interface Builder, use a graphical editor to allow the position and size of each widget to be specified in a direct manipulation manner using a mouse. Often, these systems also allow a few limited properties to be changed using a dialogue box. The Peridot system [14, 16] allows the widgets themselves to be created using direct manipulation. Lapidary combines both of these capabilities.

Most of these systems mainly support relatively static panels or dialogue boxes where the objects in the panels are menus, buttons or text fill-in slots that do not move around. XY-Wins [7] allows more mobile objects to be drawn (such as elements of visual programming languages), but they still can have only limited properties change at run time (such as their position). Other systems, such as EDGE [19] support more dynamic behavior by limiting the applications to a certain form, such as those that use graphs to display their data. Lapidary can handle almost any kind of direct manipulation interface except text editing.

Much of the behavior in Lapidary is specified using constraints on objects. A constraint is a relationship among objects that is defined once and then maintained automatically by the system, even when the objects change. Like Peridot [14] and Apogee [10], Lapidary uses one-way constraints, which means that a property of one object (e.g., the `LEFT` of object `A`) can depend on another object (`B.LEFT`), but the reverse is not implied (if `B.LEFT` changes, `A.LEFT` is changed automatically, but if `A.LEFT` is changed, `B.LEFT` is not changed by the system).[1] Other constraint systems, such as ThingLab [2], have two-way constraints, but these usually have long delays before they are able to respond to mouse events [3].

Creating Objects

Graphical objects can be created in a number of different ways using Lapidary. As shown in Figure 1, the standard menus provide the usual range of graphical primitives, so objects can be created from scratch. For example, Figure 1 (a) shows a menu object being created from rectangles and strings. Alternatively, objects can just be copied from predefined sets of "prototypes." The `Prototype` menu item brings up palettes of standard graphical widgets. These presumably will have a consistent "look and feel." For example, Figure 2 shows a palette of Macintosh-like widgets and Figure 3 shows a palette of Open-Look style widgets.[2] The designer can then select widgets in the prototype window and drag copies into the user interface construction window. Here, the copies can be edited as desired in a direct manipulation manner. Typically, this will include modifying text labels and the size and position of objects. Constraints help keep all objects in perfect alignment as widgets are edited. Unlike other user interface tools, Lapidary also allows the graphics of the widgets themselves to be modified. Whereas some may feel that this defeats the purpose of having a standard "look and feel," we believe that user interface designers should be given full flexibility. It is clearly easier to use the standard widgets without modification, so this is what most people will do, and changes will only be made when necessary.

One example of where editing of the objects may be required is when the designer is creating custom application objects that should be consistent with the standard user interface. For example, if widgets have drop shadows at a certain offset and rounded rectangles ("roundtangles") with a certain curvature for the edges, the designer might want to copy some standard objects and edit out pieces for use in his new objects.

Lapidary objects are represented using KR [8], a frame-based knowledge representation system. In KR, there is no distinction between instances and classes; any instance can serve as a "prototype" for other instances [12]. All data and method are stored in "slots" (sometimes called fields or instance variables). Data and method slots[3] that are not overridden by a particular instance inherit their values from their prototypes. An instance can have any number of additional slots as well.

[1]Actually, the constraints in Lapidary are slightly more general, because cycles *are* allowed. Thus `A.LEFT` can depend on `B.LEFT` and `B.LEFT` can depend on `A.LEFT`. When either changes, the other is updated. What is disallowed, however, is any situation where a property has more than one constraint that calculates its value.

[2]These "looks and feels" are copyrighted by Apple and AT&T respectively. We are not suggesting that people illegally use these designs, but just want to show that our system can support various popular styles.

[3]There is no distinction between data and method slots in KR. Any slot can hold any type of value, and in CommonLisp, a function is just a type of value.

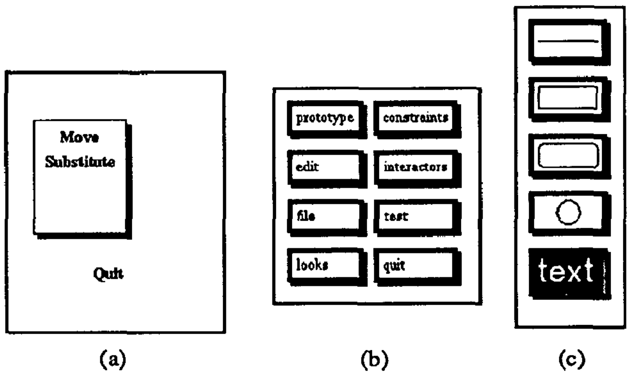


Figure 1.

The workspace window of Lapidary (a), where a menu is being created out of rectangles and strings, along with the standard command (b) and object menus (c).

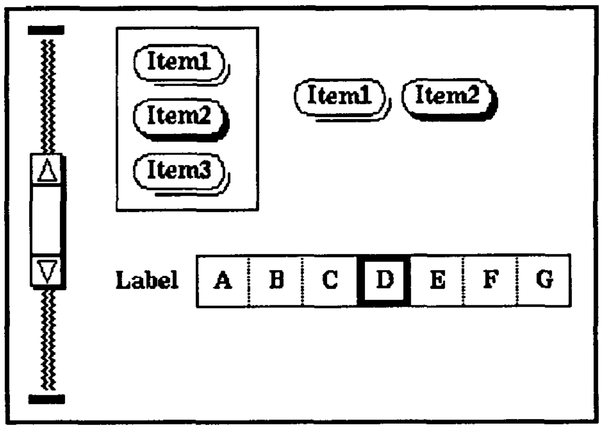


Figure 3.

Prototypes for OpenLook-like widgets.

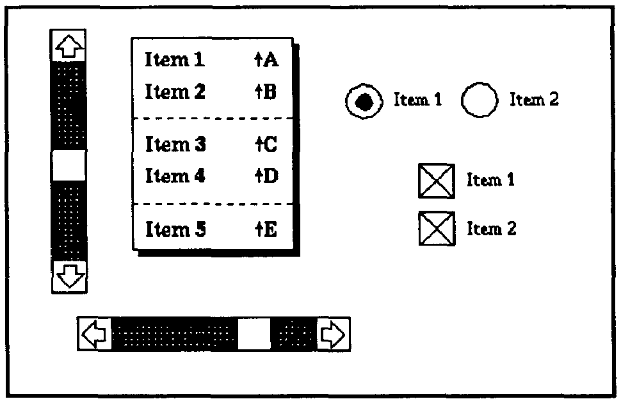


Figure 2.

Prototypes for Macintosh-like widgets.

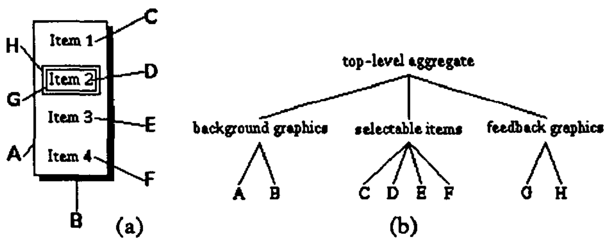


Figure 4.

A menu with the primitive objects labeled (a). The aggregates used to construct that menu are shown in (b).

Lapidary allows prototypes to be constructed by example: any slot of an instance can be changed, and then that instance can be used as a prototype for new objects. Changes to a slot of the prototype are reflected in all instances that do not override that slot. This makes it easier to edit the look and feel of an entire interface by just editing the prototypes from which the interface was created. For example, if the scroll bar in Figure 3 was itself edited, all scrollbars created from it would change.

Objects in KR can be collected into aggregate objects (also called "groups" or "collections"). Each object (including each aggregate) must be in exactly one aggregate (except the top-level aggregate, of course). For example (see Figure 4), a widget might have a top level aggregate containing an aggregate of all the background graphics (the border and shadow), an aggregate of all selectable items, and an aggregate containing the feedback graphics. Another important capability, therefore, is to easily modify the contents of aggregates. Individual objects can be edited even while they are still grouped, and it is easy to add new objects to an aggregate and remove objects from an aggregate, without disrupting the aggregation hierarchy.

Lapidary also provides an unusual capability that allows the type of objects to be changed. For example, a square can be selected and converted into a circle. Its size, position, color, other attributes, and behavior will be maintained. To facilitate this operation, Lapidary provides a graphical replace mechanism that allows wholesale replacements of one type of object with another type of object [11]. For example, the designer could use graphical replace to convert the black-filled selection squares shown in Figure 5a to the three part selection aggregate shown in Figure 5b. This capability might also be used to create round "radio buttons" out of square "check boxes," while still retaining the correct offsets for shadows and object sizes. Since all graphical objects in the system have the same structure internally, this is easy to implement. When the change is ambiguous (for example, when changing a rectangle into a line, should the line go from the upper left to the lower right, or from the lower left to the upper right?), the user is queried.

Constraints

A central feature of Lapidary that makes it appropriate for creating run-time application graphics is the use of *constraints*. Constraints allow the designer to specify a relation between a graphic object and other objects in the scene, and have that relation maintained at run-time by the system. For example, the designer might specify that an arrow should be connected to the center of a box (see Figure 6). The system will maintain this constraint, even if the box is moved.

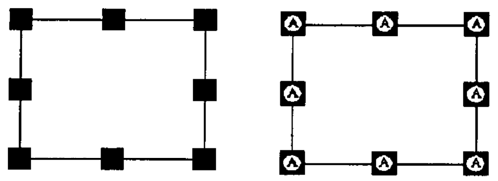


Figure 5.

(a) Eight small boxes are attached to a rectangle and will show which objects are selected; (b) the selection boxes have been changed to aggregates of three objects: a character, a circle and a grey rectangle, without affecting their behavior.

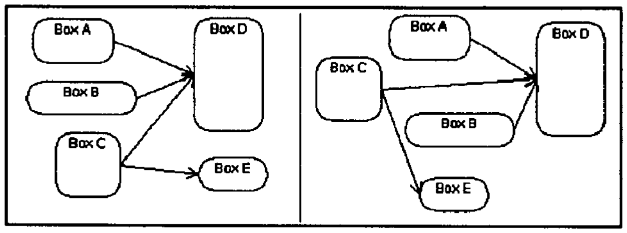


Figure 6.

Arrows are connected to the centers of the boxes by constraints so they stay attached even when the objects are moved.

Constraints are also useful for ensuring that the resulting interface looks attractive. For example, it is trivial to ensure that objects are lined up correctly, or are exactly centered. This is in contrast with interfaces created by some other UIMSs, where objects must be aligned by hand with the mouse and may look sloppy. It is also often easier to use constraints than gridding to align objects, since the desired locations do not always end up on a fixed grid.

If a constraint is one of a standard set, then it can be specified easily using the Lapidary menus. These menus support having objects be connected on their edges or in the middle, with optional offsets. The sizes of objects can also be related. There are some additional special constraints for lines and a few other objects. Experience with Peridot [14] demonstrated that these simple types of constraints make up the vast majority of those needed in typical user interfaces.

To specify a constraint, the designer selects one object as the "primary selection," another object as the "secondary selection," and then selects the constraint to apply (see Figure 7). The primary object is then made to depend on the secondary object (the primary object changes, and the secondary object does not change). The `left` (horizontal position), `top` (vertical position), `width`, and `height` each have their own section of the constraint menu. For the `left` and `top`, the small boxes are buttons that represent the primary selection, and the large box represents the secondary selection. The buttons constrain the primary selection to be at the left-outside, left-inside, center, right-inside, and right-outside of the secondary selection. The other choice is `unconstrain`, which means that there is no constraint on the `left` of the primary selection. The offset can be used to adjust how close the objects are (the default is zero). If the centered button is selected, the `offset` button changes to one labeled `percent` and selects what percent of the way across the secondary selection the primary selection should be. The default here is 50%, which is directly in the center. In Figure 7, the primary selection is the outline rectangle, and it is to the right of the gray rectangle, offset by 20 pixels.

The constraints for `top` are of the same form. For the `width` and `height`, the constraints are either on or off, and these can be modified either by offset (`A.Width = B.Width + offset`) or percentage (`A.Width = percent * B.Width`) or both. In Figure 7, the outline rectangle has no constraints on its width, and its height is 33% of the height of the gray rectangle.

When two objects are selected that already have a constraint attached to them, the constraint menu shows what constraints exist. When only one object is selected, the `Who` buttons can be used to show what other object is constrained to it. In the future, we plan to provide further debugging aids for investigating constraints, for example, to show all the objects that contribute to the selected object's display.

Sometimes, designers want to use relationships that cannot be created out of these simple choices. In that case, the `Custom` option is selected, and the designer is allowed to type in an arbitrary CommonLisp expression specifying the formula. As described in [23], this expression can use conditionals, loops, local variables, and any other Lisp form. Objects can be selected during this process, and the system will automatically include references to them in the constraint. These references take the form `(gv object slot)`, where `gv` stands for "get-value," and `object` is the name of the object referenced. An example is `(+ 10 (gv BlackRect0125 :left))`, which adds 10 to the value of the `left` slot of the object `BlackRect0125`.

As an example where a custom constraint is needed, in Figure 6, the arrows were attached to the centers of the boxes using the standard menus. In Figure 8, constraints on the `top` positions of the lines have been changed so that they are spread evenly along the side, sorted by the vertical position of the box at the other end of the line. To make this modification required writing only about 8 lines of Lisp code (using the built-in Lisp `sort` routine).

Clearly, writing Lisp code is not a job for user interface designers that are not programmers, but we did not feel that there was any reason to try to invent a special language that would be more accessible. The most common relationships are trivial to specify using the menus, and the more complex ones are not likely to be attempted by non-programmers anyway. Since the programmers are likely to already know Lisp (because the application is written in it), there is a tremendous disadvantage to making them learn a new language [20]. Furthermore, if it seems appropriate, we could easily integrate a more graphical expression for these complex constraints, such as in Graphical ThingLab [4], and use Lapidary to design the components of this graphical language.

Constraints can be put on any property of an object, not just on the numeric ones. For example, the string for a label might be constrained to be the name of the object, or there might be a constraint linking the color of a rectangle to a status variable. It is also easy to add new slots to objects which contain constraints, if desired. For example, in Figure 8, an extra slot was added containing a sorted list of the objects at the other end of the lines.

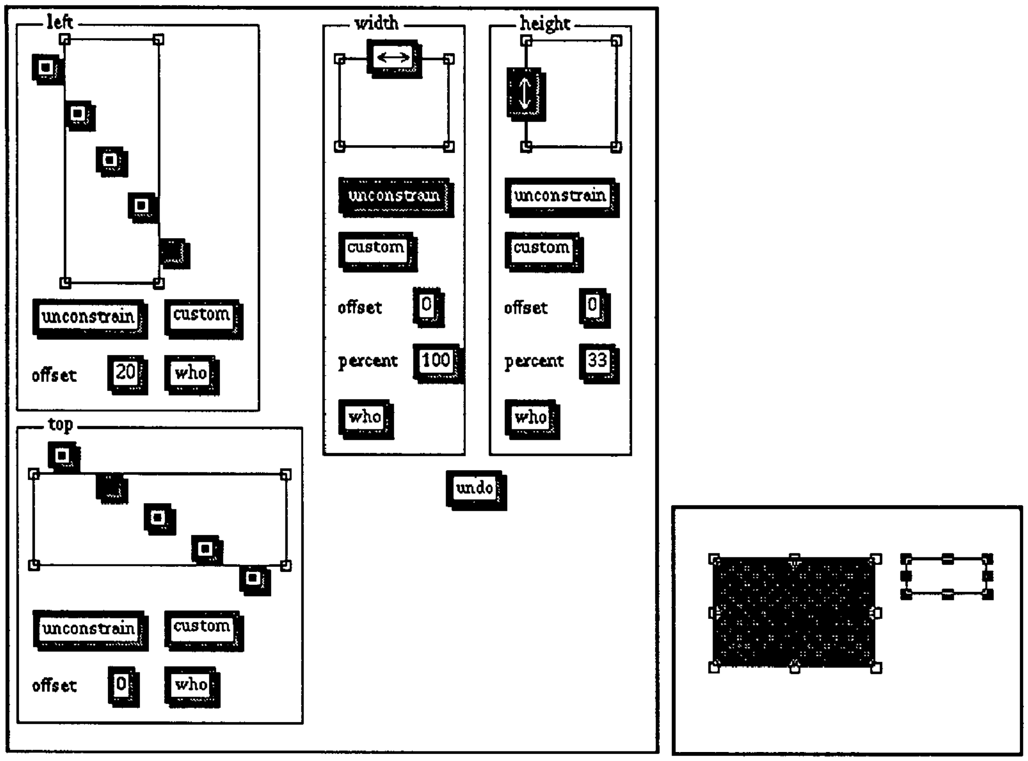


Figure 7.

The Lapidary constraint menu for rectangles on the left, and the workspace window on the right. The white rectangle in the workspace window is the "primary selection" and the gray rectangle is the "secondary selection." In the section labeled "left" of the constraint menu, the darkened box shows that the white rectangle is constrained to be offset from the right of the gray rectangle by 20 pixels, the "top" constraint is that the white rectangle is aligned at the top-inside of the gray one, its width is not constrained, and it is 33% as tall. If the gray rectangle changes, the white one will be adjusted automatically.

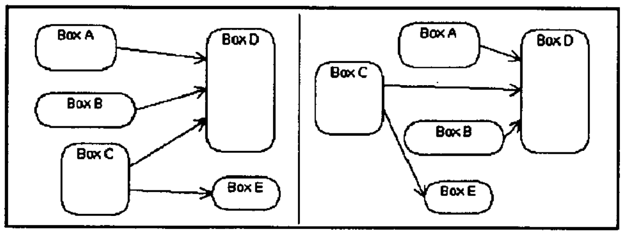


Figure 8.

The arrows' vertical constraints have been modified from Figure 6 so the lines spread themselves along the box edge, sorted by the position of the top of the box at the other end. Note that the arrows from B and C have switched their location on the left edge of D in the two views.

Restricting the constraints to be one-way allows their execution to be very fast. Lapidary can handle dozens of constraints on objects that follow the mouse in real-time. Another advantage of having one-way constraints is that the designer can write arbitrary Lisp code for a constraint, and not have to worry about whether there is an inverse formula to handle the reverse direction.

Generalizing the Constraints

In order for the graphical objects to be useful at run-time, the specific constraints must be generalized to work on run-time objects, rather than on the specific example objects used in the editor. For example, in Figure 5, the eight selection boxes are attached to a particular rectangle, and the constraints will refer to that rectangle. In order for the selection boxes to appear over whatever object is selected, the constraints need to be generalized to reference objects indirectly through variables, rather than by specific object names. To do this, the reference to the object is replaced with an expression that calculates the object desired. Usually, this has the form `(gv (gv object slot-containing-object) slot)`. For example `(gv (gv :SELF :other-object) :LEFT)`, where `:SELF` is a special reference to the object containing the constraint. Since this form is so common, we allow a short-hand `(gv object slot slot slot ...)`, where, starting from the left, each slot is used to get the object from which the next slot is accessed. The constraint system ensures that whenever any object referenced in these variables is changed, the constraints are updated.

Therefore, for Lapidary to change an object reference to be a variable, it is only necessary to change references of the form `(gv obj slot)` to `(gv :SELF obj-reference slot)`, and then store the original object in the `obj-reference` slot of the object. For example, the eight selection boxes of Figure 5 might be made to refer to the object they are attached to using a slot called `obj-over`. The constraint system then automatically ensures that the selection objects move to whatever object is assigned to the `obj-over` slot, thus making it easy to place the selection boxes on any object. Therefore, the complete specification for the top-left selection box might be:

```
(Create-Instance 'TopLeftSelectionBox :Rectangle
  (:Left (- (gv :SELF :obj-over :left) 10))
  (:Top (- (gv :SELF :obj-over :top) 10))
  (:width 21)
  (:height 21)
  (:visible (gv :SELF :obj-over)) ; visible
                  ; when there is an obj-over
  (:obj-over 'OutlineRect0125))   ; initial
                ; value is the example object
```

It is important to emphasize that Lapidary makes these transformations automatically. The user interface designer never sees any of this code. Even if the designer created custom constraints by typing Lisp code, the references in the expression can be to example objects (selected by pointing at them with the mouse), and the system will convert these references to be general variables where appropriate.

Another way that Lapidary generalizes from the examples is to automatically copy objects at run-time if necessary. For example, the designer will demonstrate one set of selection boxes over a particular example object (as in Figure 5), but may desire that multiple objects be selectable. Therefore, Lapidary arranges for the selection objects to be duplicated at run-time if necessary. One complication of copying objects is that a set of objects may be copied together (for example, all eight selection boxes), and any constraints from one of these objects to another must refer to the correct other object. For example, if the `left` of the left-middle selection box was constrained to be the same as the `left` of the first box (rather than based on the left of `obj-over`), it would be necessary to make sure that each middle-left selection box referenced the correct top-left selection box. Using indirect variable references to name the objects in these constraints is therefore used to make the copy operation simpler. Again, Lapidary deals with this complication automatically, so the designer does not have to know about it.

Interactive Behavior

Although it is useful to prototype the graphic appearance of user interfaces, it is much more useful if the interactive behavior can also be specified easily. Lapidary therefore provides this capability. When objects are copied from the prototype libraries (Figures 2 and 3), they bring with them the prototypical behaviors. By selecting the `Test` command from the Lapidary main menu, the buttons, menus, etc. can be operated by the mouse and keyboard, just as they will by the end user.

The graphics of objects can be edited without affecting the behavior attached to them. For example, the selection boxes of Figure 5a could be made hollow and larger without affecting the behavior. In fact, the boxes could even be replaced by aggregates of many objects (Figure 5b).

In order to edit the behavior of objects, or to add behavior to new objects, we have encapsulated a number of kinds of interactive behaviors into "interactor" objects [18], each of which has its own dialogue box for specifying properties (see Figures 9 and 13). For example, to change which mouse button operates a menu, it is only necessary to change the button indicated in the dialogue box.

The interactors represent pure input device behaviors, and are devoid of any graphical presentation.[4] The graphics are linked to the interactors using the dialogue boxes.

The interactors supported by Lapidary include:

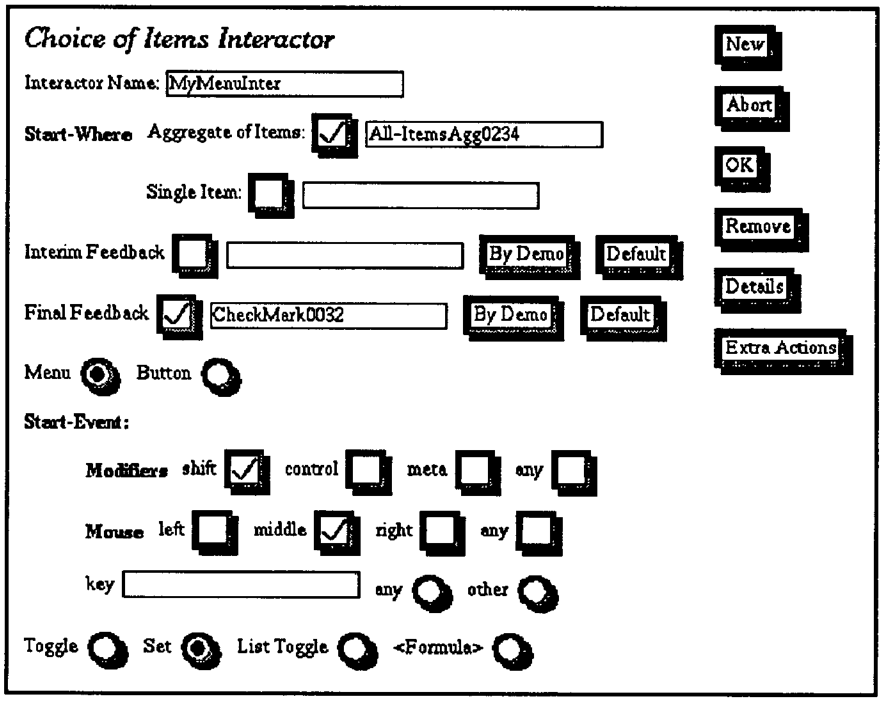
- Choice-of-Items - for selecting one or more of a set using the mouse (e.g., for menus and buttons),
- Move-Grow-Interactor - for changing the size and/or position of objects with the mouse,
- New-Point-Interactor - for entering new points with the mouse (e.g., for creating new objects),
- Angle-Interactor - for measuring angles that the mouse moves around a point (e.g., for circular gauges),
- Edited-Text-String-Interactor - for entering edited single lines of text using the keyboard.

Each of these interactors has a number of parameters that can be specified using a dialogue box. The most unusual of these is the particular graphics that will be used by the interactor. For example, for the Choice-of-Items interactor (Figure 9), the designer can select an aggregate which contains the items to be chosen among, the graphics that highlight the item the mouse is currently over (called the `interim feedback`), and the graphics that show the final selection (the `final feedback`). These parts are labeled in Figure 10 for a Macintosh-like radio-button. To specify the connection, the designer simply selects the appropriate object in the picture, and then clicks on the check box next to the role that that object will play. Lapidary automatically modifies the constraints on the object to allow it to operate as specified. There is a command to show which object is used by each role, so the designer does not have to remember what the object names are.

If the designer selects the "default" button, Lapidary tries to guess what would be appropriate based on the information already provided. For example, if the selectable items are circles, Lapidary might use a dot, as shown in Figure 10, and if the items are squares, it might use a square that has the same size and shape as a selected item. For interim feedback, it might choose to XOR the feedback object over a selected menu item, thus inverting the selection.

Another way for the designer to specify how the picture changes for an interactor is by demonstration. This is useful when objects in the picture should be modified, for example so that the currently selected item in a menu is shown in italics (see Figure 11). Another use is to have buttons move to cover their shadows (and therefore look more "3-D"), as is done for Lapidary's own user interface (Figure 12). The conventional layered model, where the interim and final feedback are objects that are displayed on top of the selected items, does not work very well in these cases. To specify the changes by demonstration, first the designer selects the objects that will change, and then uses the `By Demo` button in

[4] Therefore, this use of the term "interactor" is different from Cardelli's [6]. In his system, an "interactor" is what is here called a widget—a combination of graphics and behavior.



Choice of Items Interactor

Interactor Name: MyMenuInter

Start-Where Aggregate of Items: [✓] All-ItemsAgg0234

Single Item: [ ]

Interim Feedback [ ] By Demo Default

Final Feedback [✓] CheckMark0032 By Demo Default

Menu [✓] Button

Start-Event:

Modifiers shift [✓] control [ ] meta [ ] any [ ]

Mouse left [ ] middle [✓] right [ ] any [ ]

key any other

Toggle Set [✓] List Toggle <Formula>

Figure 9.

Dialogue box for specifying the Choice-Of-Items Interactor.

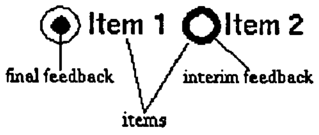


Figure 10.

A Macintosh-like radio button, with the parts labeled as to the roles they play in the Choice-of-Items interactor.

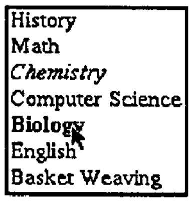


Figure 11.

A menu where the the item under the mouse changes to bold and the final item selected shows as italics.

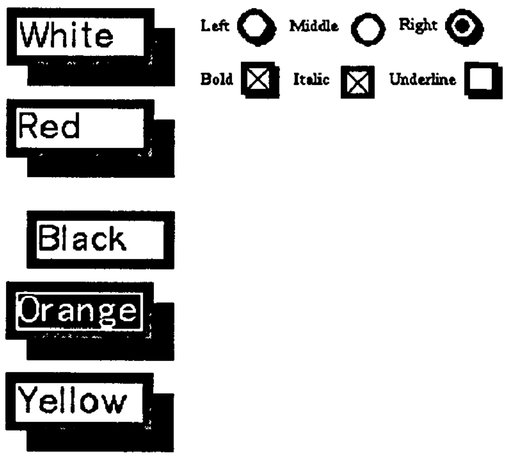


Figure 12.

Three groups of buttons. The center button of each group has moved in "simulated 3 dimensions" (towards the shadow) to show that it is being pressed by the mouse (this serves as interim feedback). The reverse video rectangle, the black dot, and the X's are final feedback objects.

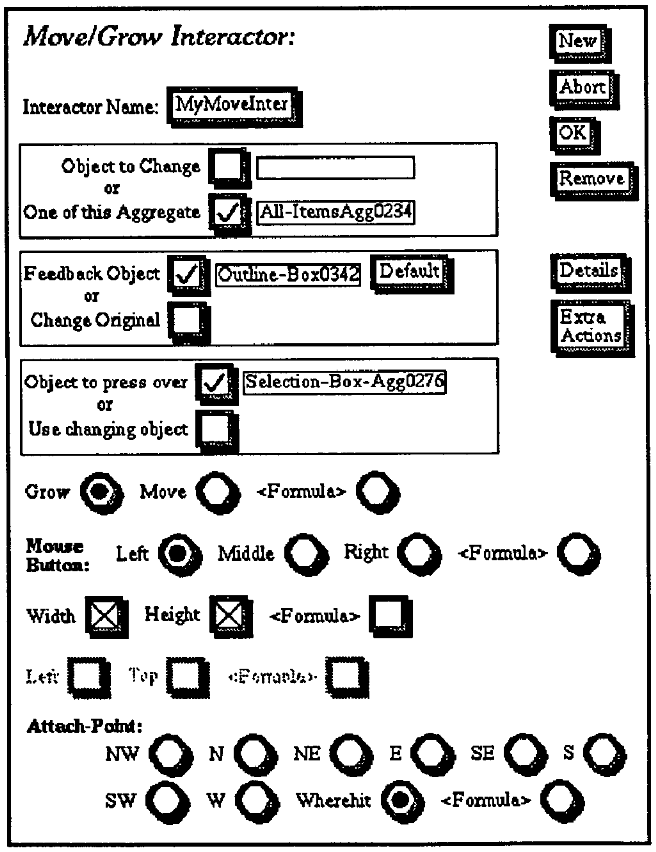
the dialogue box (see Figure 9). The full current state of the selected objects is remembered. Then, the designer edits the objects in whatever way desired, for example to make the string be italic (in Figure 11). Then, the `By Demo` button is hit again, and Lapidary creates a constraint that will choose between the two values based on whether the object is selected or not. Lapidary can smooth the transition between these graphical states by automatically adding intermediate positions. For example, the movement of the pushdown buttons in Figure 12 might be smoothed in this fashion. Changes can be made to as many properties as desired, and correct constraints will be created for all of them.[5]

Other properties of the interactor can also be selected in the dialogue boxes. These include which mouse button starts and stops the action; whether single or multiple objects can be selected; whether the item under the mouse is added, removed or toggled in the set of objects selected; an application procedure to be called when the interactor is complete; etc. It is useful to note that an attached procedure is only needed if the application needs to be notified. Usually, all graphical updates are handled automatically by the constraints.

As with constraint specification, if the dialogue boxes do not provide sufficient flexibility, then arbitrary Lisp code can be used, by selecting the `<Formula>` option. Naturally, we do not expect this to be needed often. In fact, usually most fields of the dialogue boxes will contain appropriate default values, either proposed by Lapidary or because the interactor was copied from a prototype.

As an example of the use of the interactor dialogue boxes, to program the eight selection boxes of Figure 5 to appear over objects that are specified by the mouse, the designer would first ensure that all the selection boxes were grouped into an aggregate. Then, the choice of items interactor dialogue box would be used (Figure 9). The `Aggregate of Items` is the top level aggregate in the window which holds all the objects that can be selected by the end user. The selection box aggregate is the `Final feedback`. The interactor operates immediately when the shift key is depressed and the middle mouse button goes down, so there is no interim feedback. After hitting `OK` in the interactor dialogue box, the selection box interaction can then be tested immediately and written out to a file for use by application programs. The file will contain all the code to display the selection boxes as well as the interactor to control them.

To demonstrate the flexibility of the interactors, suppose the designer wants to allow the end user to change the selected object's size by pointing at the selection boxes (as in Apple MacDraw). To specify this, the designer brings up the Move/Grow Interactor dialogue box (Figure 13). The object to be changed is an element of the selectable set of objects, the feedback object might be an XORed outline rectangle as in MacDraw (this is the default feedback object), the object that the user must press on is the selection box aggregate, the object is to grow, and the attach point is where the mouse hits (`where-hit`). This is all that needs to be specified. Suppose now that the designer wanted to change it so that the selection boxes on the sides changed the position of the object, and the selection boxes on the corners changed the size. In this case, two different interactors could be specified, one for the four side selection boxes and one for the four corner ones. Alternatively, the designer could use one interactor. The `<Formula>` choice would be used for whether to move or grow, and a custom constraint (Lisp code) would be written to differentiate whether to move or grow. Again, when the standard, simple kinds of interactions are desired, these can be specified easily, and a straightforward path is provided to allow more complex interactions to be created.



Move/Grow Interactor:

Interactor Name: MyMoveInter

Object to Change [ ]
or
One of this Aggregate [✓] All-ItemsAgg0234

Feedback Object [✓] Outline-Box0342
or
Change Original [ ]

Object to press over [✓] Selection-Box-Agg0276
or
Use changing object [ ]

[✓ Grow] Move <Formula>

Mouse Button: [✓ Left] Middle Right <Formula>

Width [✓] Height [✓] <Formula> [ ]

Left [ ] Top [ ] <Formula> [ ]

Attach-Point: NW N NE E SE S SW W [✓ Wherehit] <Formula>

Figure 13.

Dialogue box for specifying the Move/Grow Interactor.

As another example, to allow the boxes of Figures 6 or 8 to be movable by the mouse, it is only necessary to associate a Move/Grow Interactor with them. Of course, the labels in the boxes and the arrows to and from each box will stay connected automatically, because they are defined with constraints.

Current Status and Future Work

The Lapidary editor, as well as the entire Garnet project, is now under active development. The design for Lapidary is mostly complete, and significant portions have been implemented, as shown by the figures in this paper. Objects can be drawn and constraints attached to them using the menus. The constraints can

[5]In the future, we will probably add a feature that will allow properties of objects that depend on "active values" [14] to be demonstrated this way, so that the graphics that change under application control (such as progress and status indicators) can be demonstrated as easily.

be automatically generalized and objects can be replicated at run-time. The interactors are implemented, but not all the dialogue boxes for them, so only some can be attached to the objects in the editor. The libraries of prototypes also do not yet exist.

We plan to re-create Lapidary's own user interface using Lapidary, as well as make it usable by outside projects. We also will implement a variety of types of applications and interaction techniques, to test Lapidary's range.

In the future, we want to explore further ways to make application-specific behavior easy to specify. In particular, we will be looking for kinds of interactions that this system cannot handle, and trying to add them to this framework. Techniques for making custom constraints on objects easy to specify and debug will probably be particularly important.

We will also be looking to minimize the need for dialogue boxes by making more inferences from demonstration. For example, if the designer moves a mouse back and forth over a list of objects, Lapidary might guess that a menu interactor is desired without requiring any additional specification. If there are nearby objects that resemble feedback objects (e.g., they are Xored to the screen), Lapidary may infer that these objects are feedback objects and after a few mouse motions, attach one of these objects to the mouse. Of course, if Lapidary makes the wrong guesses, the designer can always correct them using the dialogue boxes.

Another problem we will be investigating is how to provide composite interactors for higher-level functionality. For example, a useful one might understand the concept of selected objects in a graphical editor, or how to create objects selected from a palette. This will make it easier to create graphical-editor-style programs.

Finally we would like to provide an "output toolkit" that parallels the "input toolkit" we currently provide. This toolkit will help organize the output of application objects into higher-level structures such as graphs and trees. Specific parameters in these structures will be inferred by demonstration.

Conclusions

Through the use of a direct-manipulation graphical editor, constraints which can be automatically generalized, and a small number of primitive interactive behaviors, Lapidary allows the user interface designer to create widgets and run-time application objects with either a custom or a standard "look and feel." The behavior can be specified separately from the graphics by selecting among a small number of options using dialogue boxes and by demonstration. If the designer requires more complex behavior, specialized constraints can be written in Lisp and attached to any property of an object. The use of graphical, direct manipulation techniques to specify the graphics and constraints, and the use of programming-by-demonstration to specify the behavior, should make the creation of application objects significantly easier. Using pre-defined prototypes will also help create objects with a standard look and feel.

Although there are only a small number of built-in constraints and interactors, these are able to cover a wide range of user interfaces, including most forms of widgets and many kinds of application-specific interactions. We believe that significant numbers of user interfaces can therefore be built without programming using Lapidary.

Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

The Garnet system is being designed and implemented by Brad Myers, Dario Giuse, Roger Dannenberg, Brad Vander Zanden, Dave Kosbie, Ed Pervin, Andrew Mickish, Jake Kolojejchick, Lynn Baumeister, Philippe Marchal, and Pedro Szekely. For help with this paper, we want to thank Bernita Myers and Dario Giuse.

References

1. ACM SIGGRAPH. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, Oct., 1988.

2. Alan Borning. Thinglab--A Constraint-Oriented Simulation Laboratory. Tech. Rept. SSL-79-3, Xerox Palo Alto Research Center, July, 1979.

3. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

4. Alan Borning. Defining Constraints Graphically. Human Factors in Computing Systems, SIGCHI'86, Boston, MA, April, 1986, pp. 137-143.

5. W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith. Towards a Comprehensive User Interface Management System. Computer Graphics, 17(3), SIGGRAPH'83, Detroit, Mich, July, 1983, pp. 35-42.

6. Luca Cardelli. Building User Interfaces by Direct Manipulation. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 152-166.

7. Alessandro Giacalone. XY-WINS; An Integrated Environment for Developing Graphical User Interfaces. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 129-143.

8. Dario Giuse. KR: Constraint-Based Knowledge Representation. Tech. Rept. CMU-CS-89-142, Carnegie Mellon University Computer Science Department, April, 1989.

9. D. Austin Henderson, Jr. The Trillium User Interface Design Environment. Human Factors in Computing Systems, SIGCHI'86, Boston, MA, April, 1986, pp. 221-227.

10. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.

11. David Kurlander and Eric A. Bier. Graphical Search and Replace. Computer Graphics, SIGGRAPH'88, Atlanta, GA, Aug., 1988, pp. 113-120.

12. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems". *Sigplan Notices 21*, 11 (Nov. 1986), 214-223. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

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

14. Brad A. Myers. "Creating Interaction Techniques by Demonstration". *IEEE Computer Graphics and Applications 7*, 9 (Sept. 1987), 51-60.

15. Brad A. Myers. The Garnet User Interface Development Environment: A Proposal. Tech. Rept. CMU-CS-88-153, Carnegie Mellon University Computer Science Department, Sept., 1988.

16. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

17. Brad A. Myers, John A. Kolojejchick, and Edward Pervin. *Opal: Garnet Project Graphical Object System.* Carnegie Mellon University, School of Computer Science, 1989.

18. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.

19. Frances J. Newbery. An interface description language for graph editors. 1988 IEEE Workshop on Visual Languages, Pittsburgh, PA, Oct., 1988, pp. 144-149. IEEE Computer Society Order Number 876.

20. Dan R. Olsen, Jr. "Larger Issues in User Interface Management". *Computer Graphics 21*, 2 (April 1987), 134-137.

21. Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages". *IEEE Computer 16*, 8 (Aug. 1983), 57-69.

22. Gurminder Singh and Mark Green. Designing the Interface Designer's Interface. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 109-116.

23. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices 23*, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.