Graphical Editing by Example

David Joshua Kurlander

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
July 1993

Columbia University Computer Science Department Technical Report CUCS-023-93

ABSTRACT

Graphical Editing by Example

David Joshua Kurlander

Constructing illustrations by computer can be both tedious and difficult. This thesis introduces five example-based techniques to facilitate the process. These techniques are independently useful, but also interrelate in interesting ways:

- **Graphical Search and Replace**, the analogue to textual search and replace in text editors, is useful for making repetitive changes throughout graphical documents.

- **Constraint-Based Search and Replace**, an extension to graphical search and replace, allows users to define their own illustration beautification rules and constraint inferencing rules by demonstration.

- **Constraint Inferencing from Multiple Snapshots** facilitates constraint specification by automatically computing constraints that hold in multiple configurations of an illustration.

- **Editable Graphical Histories**, a visual representation of commands in a graphical user interface, are useful for reviewing, undoing, and redoing sets of operations.

- **Graphical Macros By Example**, based on this history representation, allow users to scroll through previously executed commands and encapsulate useful sequences into macros. These macros can be generalized into procedures, with arguments and flow of control using graphical and constraint-based search and replace.

Individually and in combination, these techniques reduce repetition in graphical editing tasks, visually and by example, using the application’s own interface. These techniques have been implemented in Chimera, an editor built to serve as a testbed for this research.

Table of Contents

Table of Figures

Acknowledgments

Many people have contributed towards this research with their encouragement, suggestions, comments, criticisms, and sometimes their mere presence. My advisor, Steve Feiner, was a great sounding-board for ideas, and always pushed me to do a little more. You have Steve to thank for the number of pages you are about to read, and hopefully their quality. Eric Bier introduced me to graphical editing as a research issue during two summers at Xerox PARC, and suggested that I explore graphical search and replace, the first component of this thesis. Without Eric, this thesis would have been about something entirely different. Ken Pier, who hosted me my first summer at PARC, also taught me much about graphical editors, and what I learned from Ken made my own system easier to develop and more elegant as well.

I would also like to thank my committee members, Steve Feiner, Henry Lieberman, Chip Maguire, Kathy McKeown, and Brad Myers, for their insightful comments on an earlier version of this document. While in graduate school, I enjoyed interacting with all of these people, as well as their students. Henry and Brad were responsible for some of the seminal programming by demonstration research that inspired this work, so I am pleased they could be on my committee.

This research was funded partially by a grant from the IBM Watson Research Center. Dan Ling and Larry Koved were my advocates at IBM, and I enjoyed our many discussions about interface research problems. A Hewlett-Packard equipment grant supplied the workstations with which I developed early incarnations of my implementation. Chip Maguire's IBM Faculty Development award paid for the NeWS (Sun's Window System) source license. Chris Maio and I spent long hours porting the window system to HP hardware.

The Programming by Example Workshop at Apple, organized by Allen Cypher, provided a great opportunity to talk with other people doing related research. The co-editors of the resulting book suggested changes that have been incorporated into this thesis. Richard Potter has my thanks for posing a problem in the book's Test Suite that I address there, and here as well.

My years at Columbia were made much more enjoyable by having a wonderful group of friends in the department. Michael Elhadad was a great source of technical information and insights on just about everything. Frank Smadja and Jacques Robin taught me all about American Pop Culture, and helped me put graduate school in perspective. Dorée Duncan Seligmann, one of the most versatile people on this planet, was a great source of discussions, stories, and sanity checks. Graduate school would have been far less fun without regular dinners and conversations with Steve Kearns, Tony Weida, George Wolberg, JI, Henry Massalin, and Yitao Gong.

Finally, I would like to thank the folks at Microsoft Research for allowing me to finish this document there. Rick Rashid and Dan Ling at Microsoft encouraged me to complete the thesis, and gave me the necessary time to do it.

> *“Splendid,” he said happily, “for there are just three tasks. Firstly, I would like to move this pile from here to there,” he explained, pointing to an enormous mound of fine sand; “but I’m afraid all I have is this tiny tweezers.” And he gave them to Milo who immediately began transporting one grain at a time.*
>
> *“Secondly, I would like to empty this well and fill the other; but I have no bucket, so you’ll have to use this eye dropper.” And he handed it to Tock, who undertook to carry one drop at a time from well to well.*
>
> *“And, lastly, I must have a hole through this cliff, and here is a needle to dig it.” The eager Humbug quickly set to work picking at the solid granite wall.*
>
> *When they had all been safely started, the very pleasant man returned to the tree and, leaning against it once more, continued to stare vacantly down the trail, while Milo, Tock, and the Humbug worked hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour after hour—*
>
> — Norton Juster, *The Phantom Tollbooth*

Chapter 1

Introduction

Human beings are great problem solvers, but find repetitive tasks tedious. In contrast, computers usually must be taught how to perform new tasks, but then they can quickly repeat the steps on new data. Given that a computer’s strength lies in its ability to perform tasks repetitively, it is especially frustrating that many computer interfaces require the people using them to repeat interaction steps over and over again to achieve their desired goals. By building interfaces that can *automate* repeated interaction steps, we leverage off of the strengths of computers to make these goals easier to attain.

This thesis explores several new techniques to automate repetition in user interfaces. Repetition in interfaces can be local to a particular session, or more global in scope and span multiple sessions and possibly multiple users as well. Tasks repeated in multiple sessions tend to be more general than tasks encountered in a single session. Accordingly, global repetition often results when the interface lacks a command to address a commonly useful function, while local repetition usually addresses a less general problem. The techniques discussed here address both types of repetition. By incorporating new techniques to automate repetition in applications, software designers create more *extensible* applications. Extensibility is important in that it enables users to customize applications for tasks that they often perform, making these people more efficient at their work. Experienced users, or those with a particular technical skill, can also encapsulate their knowledge in a form that others can exploit, thereby benefitting the entire user population.

1.1 The Domain: Graphical Editing

Since repetition in user interfaces is often application-specific, we have chosen to focus mainly on one particular domain: graphical editing. The techniques described here have been developed for a 2D object-based illustration system, similar to such programs as MacDraw [Claris88] and Adobe Illustrator [Adobe90], although some of these techniques apply to 3D editing and other domains as well. Graphical editors are an apt focus for this research, because they can be used for many different tasks, such as constructing technical illustrations, organizational charts, network diagrams, flow charts, and architectural drawings. Since these editors have a multitude of uses, there is a real need to allow individuals to customize the system for their particular tasks.

Graphical editing also involves different types of repetition, making it an interesting target for this research. Often it is helpful to make repetitive changes to the shape or graphical properties of a set of objects. Graphical editing tasks sometimes require that objects be laid out in a repetitive fashion. The same geometric relationships may have to be established several times in a single scene, or in multiple scenes. Certain geometric relationships may need to be re-established whenever an object is manipulated. Arbitrary object transformations or manipulations may need to be applied many times. The techniques described in this thesis facilitate these types of repetition.

An assortment of other reasons make graphical editing an ideal domain for this research. Since graphical editing is a common application, familiar to many people, the ideas presented in this thesis hopefully will be of wide interest, and potentially be quite useful. By choosing a domain with which other researchers have worked, we can compare and contrast our approaches with those of others. Graphical editing is just one of many tasks that involves placing artifacts on an electronic page. Other applications, such as page layout, interface editing, and VLSI design have this as a component as well, and the techniques described here are directly applicable to them. In fact, the techniques presented in this thesis have already been used to construct interfaces as well as illustrations.

1.2 Example-Based Techniques

In all of the new techniques presented here, the user indicates the desired repetition, at least in part, by presenting an *example*. Hence, these methods are all example-based or *demonstrational techniques* [Myers92], in which the user presents to the application an example of the desired task, and the application uses this specification to perform similar tasks on other examples.

Alternatively, the user could write a program to perform the desired repetition, but this has several disadvantages. First, it requires that the user know how to program, and many computer users lack this skill, particularly users of such basic, ubiquitous programs as graphical editors. Second, even users with programming experience may be unfamiliar with the extension language or library interface of the editor. Third, the task of programming is very different from the tasks performed in applications such as graphical editors, and switching to a programming mindset involves a mental context switch.

In contrast, using demonstrational techniques is much closer to using the native application. Demonstrational techniques are accessible to anyone already possessing the skills to use the application's interface. Halbert, for example, describes one demonstrational technique, programming by example, as programming an application through its own interface [Halbert84]. Conventional programming skills are either not necessary or fewer are needed. Demonstrational interfaces also have the advantage that abstractions are specified using concrete examples, so those people that have difficulty working with abstractions will probably find these interfaces easier to use.

However, there are many possible abstractions or interpretations for a single concrete example, so there must be a way of resolving this ambiguity. Demonstrational systems often deal with this problem by using heuristics to choose the most likely interpretation, or requiring that the user explicitly specify the intended meaning. Also since few demonstrational techniques are Turing equivalent, sometimes there are useful abstractions that cannot be specified using this approach. Programming is still the easiest way to specify some complex extensions, and will be so for the foreseeable future. However, there are many extensions that can be expressed without programming, and the goal of this thesis is to identify classes of these, and to develop new demonstrational techniques for expressing them as well as new forms of visual feedback to represent the demonstrations.

1.3 Chimera

To test the ideas contained in this thesis, I have built the Chimera editor system. Appendix A describes many reasons for giving the editor this name. Chimera is actually an editor framework or substrate, in which other editors can be embedded. Three different editors currently reside in the Chimera system: editors for creating graphics, interfaces, and text. The new methods presented in this thesis have been implemented to work in both the graphics and interface editors. Both the graphics and interface editors of Chimera have extensive coverage of the primitives and commands one might want in such editors. Chimera's graphics editor can create and manipulate circles, ellipses, lines, text, rectangular boxes, rounded boxes, arcs, Bezier curves, freehand curves, beta splines, and cardinal splines (cyclic and non-cyclic). The interface component can edit windows, command buttons, radio buttons (exclusive and non-exclusive), menus, checkboxes, horizontal and vertical sliders, application canvases, scrollbars, labels, scrolling lists, text controls, text editors, graphical editors, and minibuffers. All of Chimera's interfaces were generated in Chimera. Both the graphics and interface editors have approximately 180 commands (most of them shared). Since it did not directly participate in this research, the text editor is much less sophisticated.

Chimera runs on Sun SparcStations under the OpenWindows 3.0 window system. It is coded in Lucid Common Lisp (with CLOS object-oriented extensions), C, and PostScript. Most of the editor is written in Lisp, but C procedures handle some numerically intensive components and communicate with Chimera's PostScript procedures that process input events, manipulate windows and widgets, and perform graphics output. Portions of Chimera's interface were inspired by GNU Emacs [Stallman87], and the Gargoyle Illustrator [Pier88]. From Emacs, Chimera borrows keymaps, a minibuffer for text-based commands, a Lisp-based extension language (Common Lisp though, instead of ELisp), the default text command bindings, and multiple editor modes. Chimera's graphics mode takes snap-dragging [Bier86] and many of its commands from Gargoyle. However, Chimera's implementation shares no code with either of these systems.

1.4 Graphical Editing by Example

This section introduces five new example-based techniques to automate repetition in graphical editor interactions. These techniques: graphical search and replace, constraint-based search and replace, constraints from multiple snapshots, editable graphical histories, and graphical macros by example, are briefly described together with examples of their use. Later in this dissertation, entire chapters present each of these techniques in detail. All figures in this dissertation were generated by the PostScript output of Chimera, and each figure depicting a demonstration of a technique was generated by the system running on a real example. A videotape is also available, showing an interactive demonstration of these techniques in Chimera [Kurlander93c].

1.4.1 Graphical Search and Replace

Often shapes are repeated many times in a single illustration. Similarly, many illustrations contain the same graphical properties, such as particular fill colors or line styles, repeated multiple times. When it becomes necessary to change one of these coherent properties, the task can be very tedious since these modifications will need to be made throughout the illustration. *Graphical search and replace* is a technique for automating tasks such as these. Users of text editors have long been familiar with the utility of textual search and replace, and graphical search and replace is intended to be its analogue in graphical editors.

Figure 1.1 shows a simple diagram of a computer network consisting of 18 terminals, a magnetic tape device, a file server, and a compute server. The network manager decides to

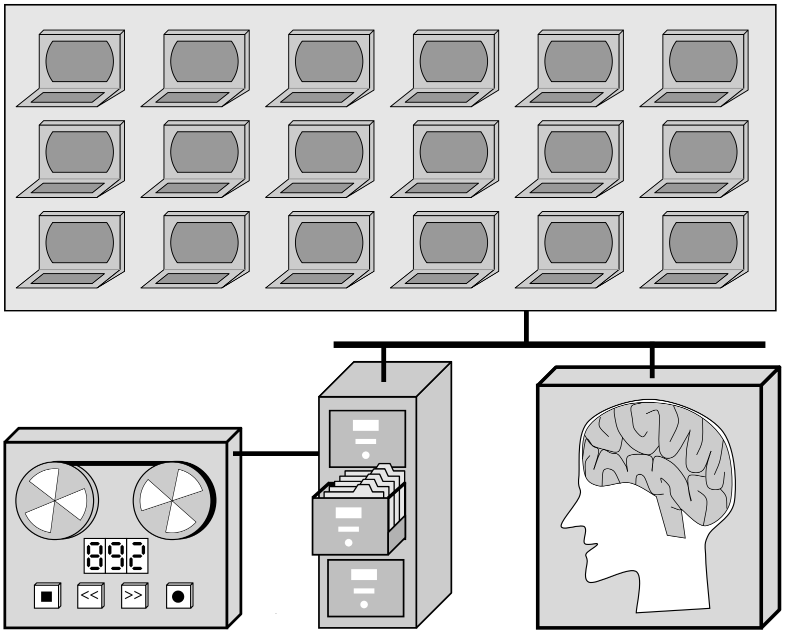
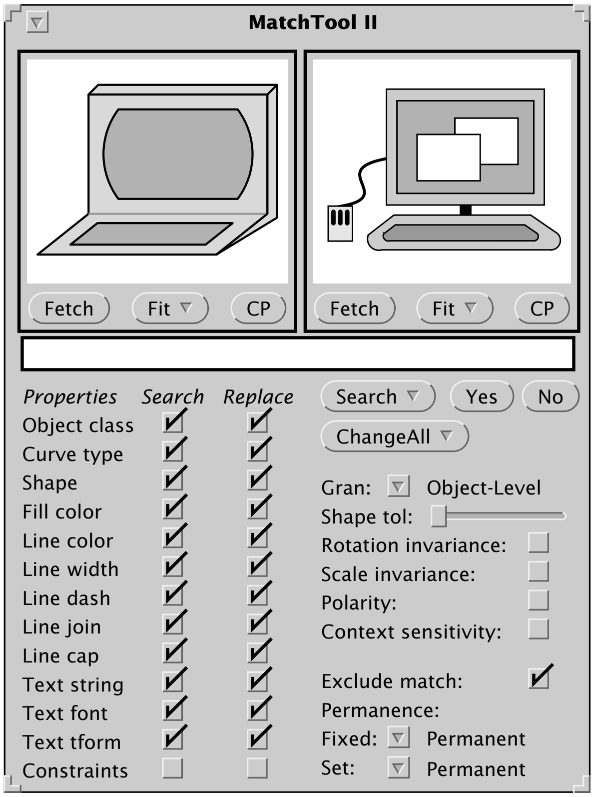


Figure 1.1 A network diagram drawn in Chimera.

replace all of the conventional terminals with workstations, and wants to update the diagram with as little effort as possible. One approach would be to delete each drawing of a terminal and replace it with a drawing of a workstation, but this would require repeating the copying and positioning operations for each new item. Since the artist did not group together all of the graphical primitives composing a terminal, and there is no pre-defined terminal object, this approach would be especially tedious. Another approach would be to use graphical search and replace.

Chimera's graphical search and replace utility is called MatchTool 2, and it appears in Figure 1.2. At the top of the window are two fully editable graphical canvases, in which objects can be drawn or copied. The left pane contains the search objects and the right pane contains the replacement objects. Unlike pure textual search and replace, there are many properties of graphical objects that can participate in queries and be modified by replacements. For example, we might want to search only for objects of a particular object class or having a certain line width, and replace these properties or others. Below the search and replace panes and to the left are rows of graphical properties followed by checkboxes. The first column of checkboxes specifies which properties of the objects in the search pane must be present in each match. During a replacement, the second column specifies which properties of the objects in the replace pane will be applied to the match. Here the user checks nearly all the graphical attributes in the two columns since he wants to match and replace all of these properties, and then presses the ChangeAll button. Figure 1.3 shows the resulting scene.



Properties	*Search*	*Replace*
Object class	☑	☑
Curve type	☑	☑
Shape	☑	☑
Fill color	☑	☑
Line color	☑	☑
Line width	☑	☑
Line dash	☑	☑
Line join	☑	☑
Line cap	☑	☑
Text string	☑	☑
Text font	☑	☑
Text tform	☑	☑
Constraints	☐	☐

Gran: Object-Level
Shape tol:
Rotation invariance: ☐
Scale invariance: ☐
Polarity: ☐
Context sensitivity: ☐

Exclude match: ☑
Permanence:
Fixed: Permanent
Set: Permanent

Figure 1.2 A MatchTool 2 window containing a search and replace specification that changes drawings of terminals to workstations.

Since the search and replace panes contain example objects as part of the specification, this technique is example-based. These objects represent large sets of potential matches and replacements, and the columns of checkboxes, which are explicitly set by the user, indicate how these objects are to be generalized for the search and replace specification. Note that these illustrations contain more repetition than the recurrence of terminals and workstations. The same fill color is used in many of the objects, and we could make the illustration darker or lighter with a few search and replace iterations. The drawers of the filing cabinet, the files in the middle drawer, the triangular holes in the reels of tape, and the button shapes on the tape recorder are all repeated shapes that can be altered easily using graphical search and replace. In fact, graphical search and replace proved useful here in increasing the diameter of all the polygons comprising the segments of the numeric LCD display, since they were too narrow in a preliminary drawing. As will be discussed in Chapter 3, graphical search and replace is useful for a number of other applications, such as producing repetitive shapes generated by graphical grammars, filling out graphical templates, and searching for shapes in multiple files using a graphical grep utility.

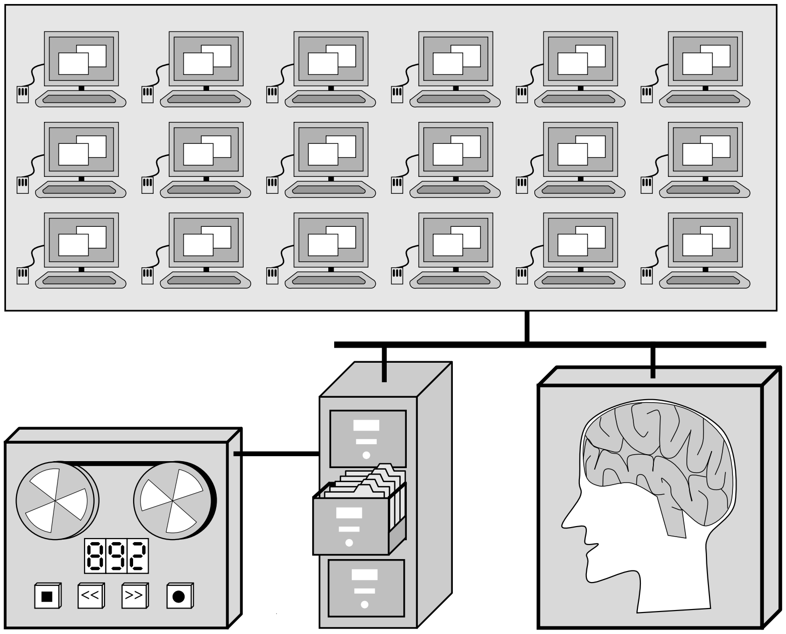


Figure 1.3 The network diagram of Figure 1.1, after applying the graphical search and replace specification of Figure 1.2. All terminals are replaced by workstations.

1.4.2 Constraint-Based Search and Replace

Graphical search and replace, as presented in the last section, lacks an important capability: it can be used to find and alter the complete shape of an object, but not individual geometric relationships. For example, if a search takes into account shape, every slope, angle, and distance is significant. Similarly, when a shape-based replacement is performed, the replacement receives its complete shape from the objects in the replace pane. However, sometimes it is useful to search for objects that match a few specific geometric relationships, and change some of these relationships. For example, we might want to look for lines that are nearly connected, and connect them. The angle between the lines is not significant for the search, nor are the lines' lengths. We also might want to leave some geometric relationships unaltered, such as the positions of the remote endpoints of the lines. To perform tasks such as this, an extension of graphical search and replace, called *constraint-based search and replace*, is useful.

Constraint-based search and replace specifications can have constraints in the search and replace patterns. Constraints in the search pattern indicate which relationships must be present in each match, and those in the replacement pattern indicate which relationships are to be established in the match and which are to remain unaltered. For example, consider the aforementioned task of connecting nearly connected lines. The search and replace panes for this task are shown in Figure 1.4, and contain graphical objects and constraints as they are visually represented in Chimera. The search pane in Figure 1.4a contains two lines with endpoints connected by a zero inch distance constraint. Note

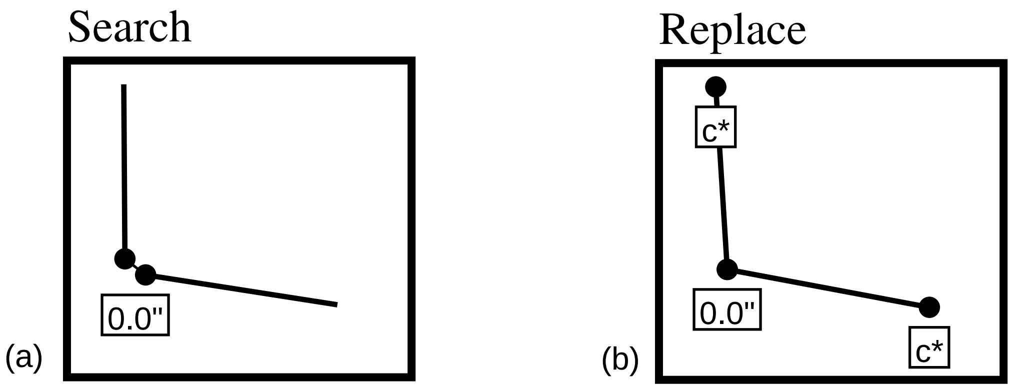


Figure 1.4 Constraint-based search and replace pattern to connect nearly connected lines. (a) the search pane; (b) the replace pane.

however that these endpoints do not obey this constraint. The tolerance of the search is provided by example—all pairs of lines that have endpoints at least this close will match the pattern. The search pattern graphically shows how far off objects can be from the specified relationships and still match.

The replacement pattern in Figure 1.4b contains two different kinds of constraints. The first constraint, the distance constraint that also appears in the search pane, indicates that the lines are to be connected together by the replacement. However there are two other constraints in the replace pane, marked by c^*, that fix the remote vertices of the match at their original locations when performing this change.

Figure 1.5 shows the result of applying this rule to a rough drawing of a W. All of the segments in Figure 1.5a are nearly connected, so they become precisely connected after the replacement as shown in Figure 1.5b. Both graphical and constraint-based search and

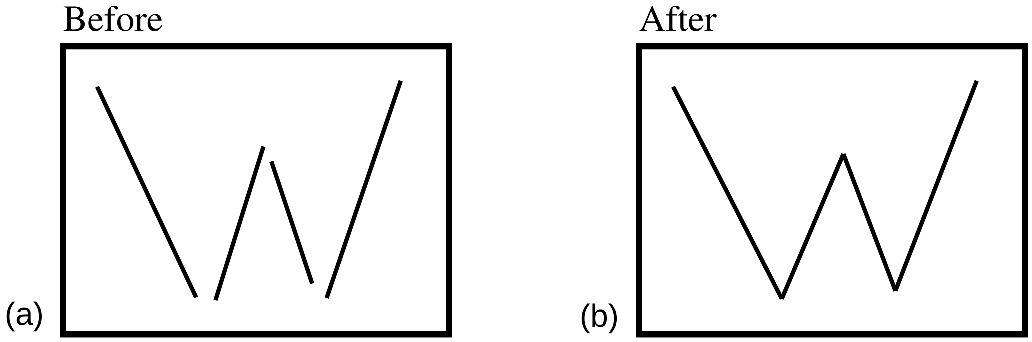


Figure 1.5 Application of the new rule. (a) a scene before the replacements; (b) the scene after the replacements.

replace rules can be collected in rule sets and archived. Rules can be applied to a static scene, or can be expanded dynamically as the scene is drawn and edited. Constraint-based search and replace provides a means to establish particular geometric relationships repeatedly in a graphical scene. As will be discussed in Chapter 4, this technique can be used for a variety of scene transformations, including illustration beautification, but unlike other graphical editors capable of illustration beautification, Chimera allows beautification rules to be defined by the end user, without programming, using a demonstrational technique.

1.4.3 Constraints from Multiple Snapshots

As in the last example, constraint-based search and replace can be used to infer the intended presence of certain constraints in a static scene. Often static scenes do not contain enough information to infer all the desired geometric constraints. Since these constraints govern the way objects move in relation to one another, it is often easier to infer the presence of constraints by determining which relationships remain invariant as the scene objects move. Another new technique, *constraints from multiple snapshots*, does just that. Given several valid configurations or *snapshots* of a scene, this technique determines which constraints are satisfied in all of them, and instantiates these. The initial snapshot completely constrains the scene. Subsequent snapshots remove additional constraints from the constraint set. Geometric constraints make graphical editing easier by automatically maintaining desired geometric relationships between scene objects. However, it is often very difficult for people using graphical editors to figure out all of the useful constraints that need to be established. This technique uses examples of valid scene configurations to automatically determine these constraints.

For example, Figure 1.6a shows a drawing of a balance. After completing this initial drawing, the user presses a button labeled “Snapshot” to indicate this illustration is a valid configuration of the scene. Next, the user rotates the balance’s arm on its axis, and translates each of the trays to reconnect them to the arm, producing the illustration shown in Figure 1.6b. The user presses the “Snapshot” button once more, and the system determines the relationships that are present in both snapshots. For example, the base remains fixed,

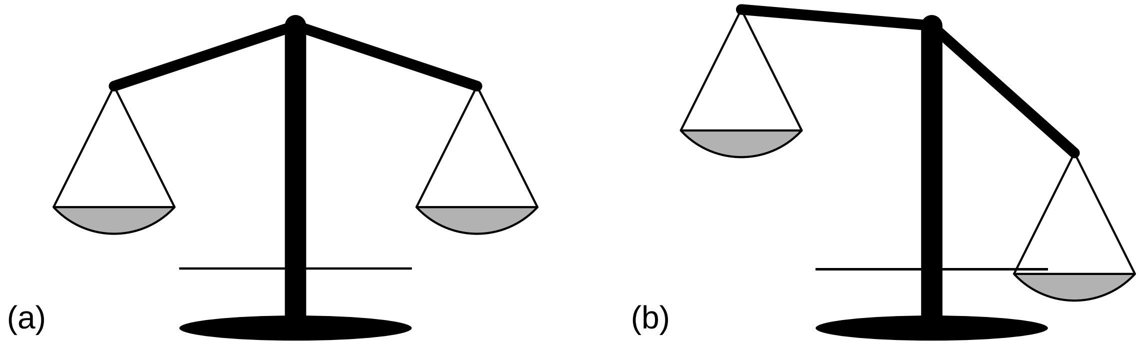


Figure 1.6 Two snapshots of a balance. (a) initial snapshot; (b) subsequent snapshot.

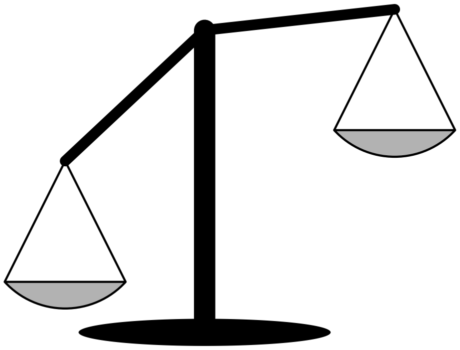


Figure 1.7 A third configuration of the balance, generated by the system, when one end of the arm is moved.

the arm rotates about its axis, and the trays remain upright, and connected to the arm. Up until now, all of the constraints inferred by the system have been ignored during graphical editing, since constraint maintenance was turned off. The user turns on constraints, and moves one end of the balance's arm. The various components of the balance automatically reconfigure as shown in Figure 1.7, maintaining the inferred geometric relationships.

After providing a few snapshots and manipulating scene objects, the user may find that the system inferred unintended constraints that were present in all of the snapshots. If unwanted constraints prevent scene objects from being moved into a desired configuration, the user can always turn off constraints, move the objects into this new configuration and take another snapshot. The new arrangement automatically becomes a valid configuration.

By inferring constraints from multiple snapshots, Chimera provides an alternative form of constraint specification to the traditional declarative method of explicitly instantiating all the constraints in an illustration. Further examples of this form of constraint inferencing, as well as the algorithm that Chimera uses to infer constraints from multiple snapshots, appear in Chapter 5. Having constraints in an illustration makes it easier to repeatedly alter scene elements when geometric relationships between the objects need to be maintained, so this technique, like the others discussed so far, addresses the problem of reducing repetition in graphical editing tasks.

1.4.4 Editable Graphical Histories

Another approach to reducing repetition in many applications is to save all of the operations as they are being performed and allow the user to select a set of these operations to reexecute or *redo*. There are several approaches to selecting the operations to be redone. Some applications limit redos to the last operation performed; others require that these operations be demonstrated in a special recording mode. Other systems detect repetitions and automatically extract out the repeated elements themselves. Another approach is to provide an understandable history representation from which the user selects operation sequences to redo.

Textual command histories are easy to represent—the lines of text can be laid out one after another sequentially. However, histories of applications in a graphical user interface present a special challenge, since graphical properties such as colors and line styles must be represented in a user-understandable fashion, and geometric characteristics such as position become important to the interpretation of commands. Another of the techniques introduced here, *editable graphical histories*, is a representation for commands in a graphical user interface.

Editable graphical histories use a comic strip metaphor to depict commands in a graphical user interface. Commands are distributed over a set of panels that show the graphical state of the interface changing over time. These histories use the same visual language as the interface, so users of the application should understand them with little difficulty. For example, consider the illustration of Figure 1.8 containing two boxes and an arrow. The history generated during its construction is shown in Figure 1.9. (Although Figure 1.9 appears to include two history windows, the figure really shows two successive scrolls of a single window.)

The first panel depicts grids being turned on from the editor control panel. The name of the command (Toggle-Grids) appears above the first panel, and the panel itself shows the checkbox that was toggled to invoke the command. The second panel shows a box created

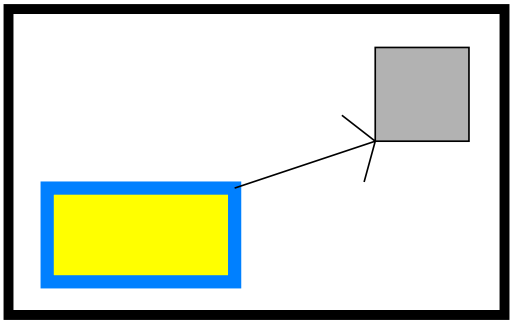
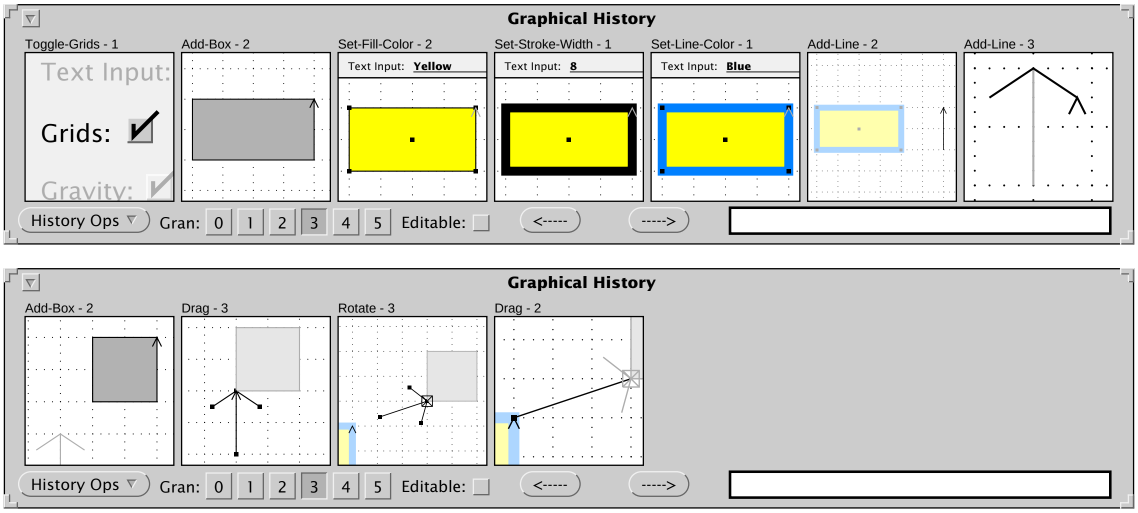


Figure 1.8 A simple scene.



Text Input: Yellow

Text Input: 8

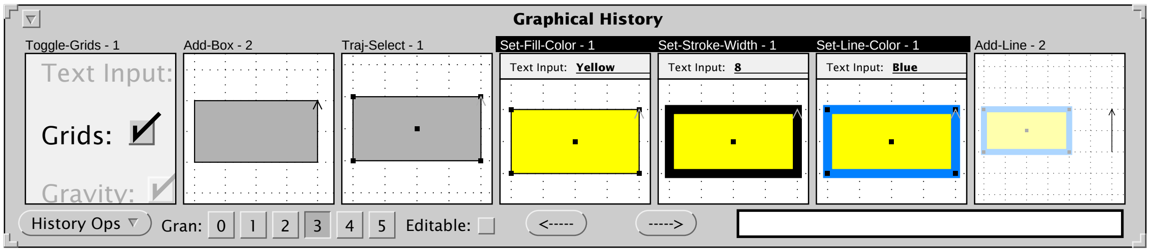
Text Input: Blue

Figure 1.9 Editable graphical history that generated Figure 1.8.

in the editor scene using the Add-Box command. In the third panel the box is selected, a color (yellow) is typed into the Text Input widget, and the Set-Fill-Color command is invoked. This panel is split to show parts of both the control panel and editor scene. The next panels show changes to the box's stroke width and line color, a line being added beside the box, and two lines being added above the first to create a hand drawn arrowhead. The scrolled window below shows in its four panels a box being added to the scene, the arrow being dragged to the box, the arrow being rotated so that its base aligns with the first box, and finally the arrow's base being stretched to reach the first box.

Several strategies are employed to make the histories shorter and easier to understand. Multiple related operations are coalesced in the same panel. Each panel's label indicates the number of commands that it represents, and we can expand high-level panels into lower-level ones and vice versa. For example, the third panel of Figure 1.9 contains two operations: one to select a scene object, and the other to change the fill color of selected objects. This panel expands into the third and fourth panels of Figure 1.10. So that the history panels will be less cluttered, each panel shows only those objects that participate in the operations, plus nearby scene context. Objects in the panels are rendered in a style according to their role in the explanation. By default, Chimera subdues contextual objects by lightening their colors, and objects that participate in the operations appear normally. In the first panel, the grid checkbox and its label stand out, since all other control panel widgets are subdued.

Editable graphical histories can be used to review the operations in a session, and to undo or redo a sequence of these operations. For example, suppose we would like to apply to the square the commands that set the fill color, stroke width, and line color of the lower box. We find the relevant panels in the history, and select them. In Figure 1.10, we have



Text Input:
Grids: ☑
Gravity: ☑

Text Input: Yellow

Text Input: 8

Text Input: Blue

Editable: ☐

Figure 1.10 The beginning of the history shown in Figure 1.9, with Figure 1.9's third panel expanded into the third and fourth panels here, and the fourth through sixth panels above selected for redo.

selected the fourth through sixth panels. The panel selections are indicated by white labels on a black background. We did not include the third panel depicting the selection of the original box, since we want to apply these commands to a different object. After selecting the square, we replay these commands verbatim by executing the Redo-Selected-Panels command, and it changes appropriately.

Editable graphical histories reduce repetition in Chimera by forming an interface to a redo facility. Chimera also has a mechanism for inserting new commands at any point in the history, which reduces repetition in a subtler manner. The histories can be made editable, which replaces each static panel with a graphical editor canvas. The panels can be edited and a command invoked to propagate these changes into the history. To insert new commands in the middle of the history, the system undoes subsequent commands, executes the new commands, and redoes the old ones. Since redo is being performed, repetition is automated. As an example, we make the panels editable, and modify the last panel of the first row of Figure 1.9 to draw a fancier arrowhead and change the width of the arrow's base. The new panel is shown in Figure 1.11. We edited this history panel rather than the editor scene directly, since at the point in time represented by the panel the arrow is still aligned with the grid axes, and later the change would be more difficult. Propagating these changes into the history results in the new scene shown in Figure 1.12. Chapter 6 discusses editable graphical histories in greater detail.

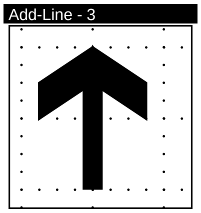


Figure 1.11 A fancier arrow. Operations added in place, in a history panel, embellish the original arrow.

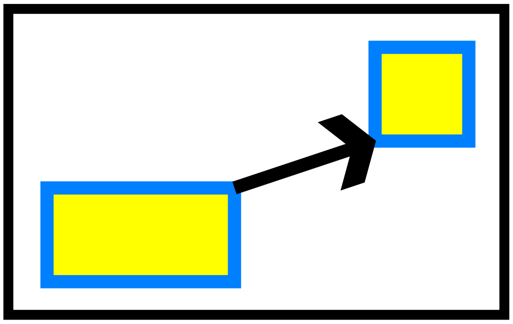


Figure 1.12 The scene of Figure 1.8, after the modifications described in this section.

1.4.5 Graphical Macros by Example

The basic redo operation discussed in the last section is limited in that it can only play back commands verbatim. Command sequences executed in one application context often lack the generality to perform the same high-level function in others. The process of generating a procedure by demonstrating a task in an application is called *programming by demonstration*, and one of the major challenges involves generalizing the demonstrated commands to work in different contexts. Another challenge is providing a visual representation of these programs. Chimera includes a programming by demonstration or *macro by example* component that uses editable graphical histories as its visual representation for reviewing, editing, and generalizing the program, as well as reporting errors.

For example, consider an editing task in which we left-align two boxes. The steps are captured in the graphical history of Figure 1.13. Initially we create the two boxes (panels 1 and 2). Next we turn on 0° and 90° slope alignment lines (panels 3 and 4), and select the upper left corner of the bottom box (panel 5) and the lower right corner of the top box (panel 6) to generate these lines. Finally we select the top box, and drag it until it snaps to the appropriate intersection of two alignment lines (panel 7).

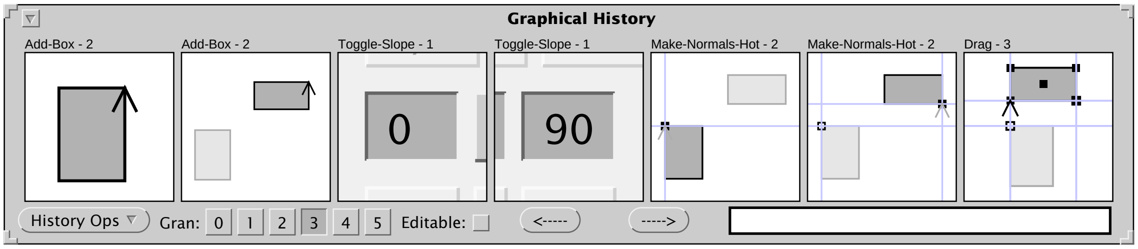
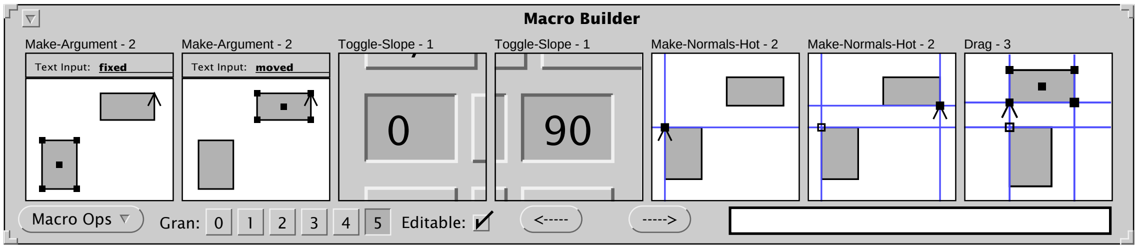


Figure 1.13 Graphical history showing the creation of two boxes, and the left-alignment of the top box with the bottom one.

At some point we realize that these operations are generally useful, and decide to encapsulate them in a macro. There is no need to repeat the operations in a special learning mode. We scroll through the history, find the relevant panels, and execute a command to turn them into a macro. Here we select all the panels, except those showing the Add-Box commands, since we want the boxes to be arguments to the macro. A macro builder window appears, containing the panels that were selected in the history window.

In the next step we choose the arguments of the macro. To do this we make the panels editable which allows objects in the panels to be selected. We select an instance of each argument, give it a name, and invoke the Make-Argument command. This appends argument declaration panels at the beginning of the history. Here we select the lower left box from a panel, name it “fixed” since it doesn’t move, and execute Make-Argument. We do the same for the other box, but call it “moved” since this is the box that was translated. Figure 1.14 shows the resulting macro builder window, with the argument declaration panels just created.


Text Input: fixed

Text Input: moved

Figure 1.14 Macro builder window, containing a macro to left-align two boxes.

Next we execute a command that chooses default generalizations of all the commands, according to built-in heuristics. Users can view and alter these generalizations. Finally we choose to invoke this macro on another set of boxes. A macro invocation window appears that allows us to set and view the arguments. We test this macro on a sample scene, and it works as expected. Chapter 7 discusses the many ways that editable graphical histories support the macro definition process.

1.5 Synergy

The techniques discussed in the last section automate several different types of editor repetition, and though they may at first seem unrelated, they actually fit together in a coherent whole and support each other synergistically. Figure 1.15 is a graph with the five techniques represented as nodes, and edges indicate which components are related to one another. The edges on the left show components that are currently related in Chimera, and those on the right show other relationships that might be established in the future.

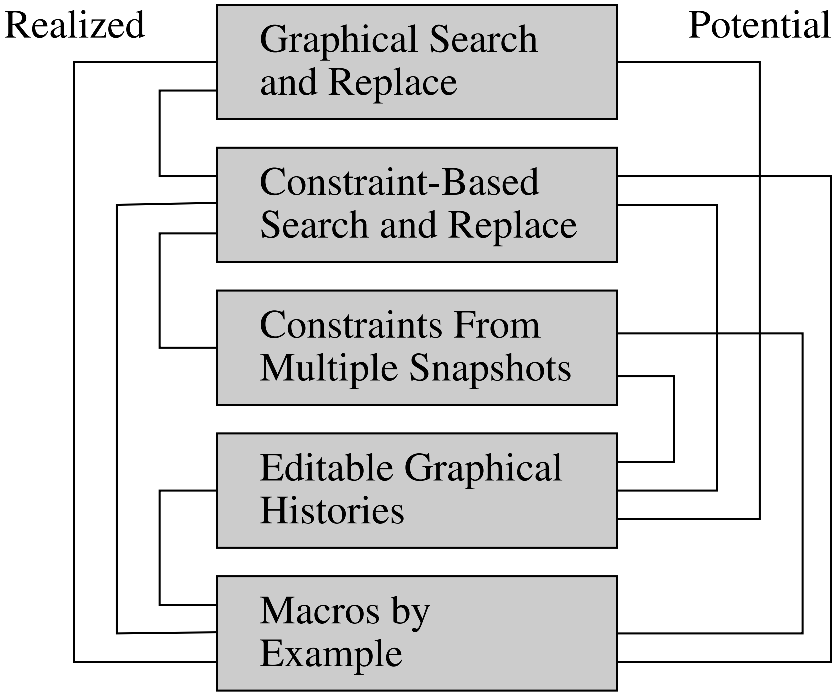


Figure 1.15 Relationships between the five techniques. Edges on the left represent existing relationships, and those on the right potential relationships.

Listed below are the existing relationships between the components:

- Constraint-based search and replace is an extension to graphical search and replace, allowing geometric relationships to be sought and changed.
- Constraint-based search and replace infers constraints from static scenes. Constraints from multiple snapshots infers constraints from dynamic scenes, since static scenes often contain insufficient information.
- Editable graphical histories form the visual representation for Chimera's macro by example facility.
- Graphical search provides an iteration construct for macros by example. The system can execute a macro for all objects fitting a given graphical description.
- Constraint-based search, like graphical search, can be used to specify macro iteration.

Listed next are relationships that could potentially link the components:

- Graphical search could be used by the system to find unique scene objects to serve as landmarks in the graphical history panels. Graphical search and replace operations could also be represented in the graphical history.

- Constraint-based search and replace operations could be represented in the graphical history.
- Constraint-based search and replace could be used to define higher-level semantic operations to be understood by the macro system. For example, using constraint-based search and replace we can currently define a rule to bisect all nearly bisected angles. When the system sees that an angle is being bisected using lower level commands, it could then assume that one possible generalization of the command sequence is angle bisection.
- Constraint inferencing from multiple snapshots could be represented in the graphical history.
- Constraint inferencing from multiple snapshots could be used to find which constraints are invariant during a graphical macro, and these invariants could be enforced.

The relationships between the components are numerous, and interestingly Figure 1.15 would be a complete graph if it were not for the missing edge between graphical search and replace and constraints from multiple snapshots.

1.6 Contributions

This work makes several important contributions to the fields of user interface and computer graphics research:

- **Repetition**: Innovative ways of automating repetition in graphical user interfaces are introduced.
- **Encapsulation**: New methods are presented that allow expert users to encapsulate their knowledge in a form that other users can exploit.
- **Extensibility:** Individuals can use the technology described here to extend or customize their interfaces for the tasks that they need to perform, and thereby select interface abstractions that are more suitable for their work.
- **Demonstration:** The techniques introduced are example-based, and do not require conventional programming skills.

Particularly, the five new techniques introduced in this thesis fulfill these goals within the domain of graphical editing:

- **Graphical Search and Replace**, the analogue to textual search and replace in text editors, enables repetitive changes to graphical properties or shape to be made throughout a graphical scene at once.
- **Constraint-Based Search and Replace**, an extension to graphical search and replace, makes it easy to change geometric relationships repetitively throughout an entire scene, or to establish precise geometric relationships as a scene is being drawn.

- **Constraints from Multiple Snapshots** infers geometric constraints from multiple valid configurations of a scene, allowing users to specify constraints in a scene by demonstrating how objects are permitted to change, rather than through explicitly itemizing these relationships.
- **Editable Graphical Histories** provides a new means of representing sequences of commands in a graphical user interface. These histories are useful for reviewing the commands invoked during a session with an application. They also provide a convenient mechanism for undoing and redoing commands.
- **Macros by Example**, based on this history mechanism, provide an interface for generalizing, debugging, editing, and reviewing procedures defined by demonstration.

A final contribution of this work is the implementation of a single system, the Chimera editor. While Chimera was designed as a testbed for these five individual techniques, it has also shown how these components support each other and work together. Through the synergy of these techniques, Chimera offers a coherent interface, and satisfies the goals of facilitating repetition, encapsulation, and customization using demonstrational techniques.

1.7 Guide to this Thesis

Understanding the context of this research is crucial to understanding its contributions, hence the next chapter surveys related work. It summarizes techniques that are already commonly used in editor systems to reduce repetition and promote extensibility, and next it considers more experimental, demonstrational approaches. The chapter concludes with a discussion of important demonstrational systems in domains other than editing.

The subsequent five chapters focus in further detail on the techniques introduced earlier in this chapter. They each present additional motivation and examples, relate the work to other research (including many of the systems described Chapter 2), and present algorithmic and implementation information. Chapter 3 covers graphical search and replace. An extension to this technique, constraint-based search and replace, is discussed in Chapter 4. Chapter 5 explains a technique for inferring geometric constraints from multiple snapshots of a graphical scene. Chapter 6 describes editable graphical histories, a means of representing sequences of commands in a graphical user interface. Then Chapter 7 presents a macro by example facility that uses this history representation for numerous purposes.

The final chapter, Chapter 8, summarizes the research, reiterates the contributions, and discusses future directions for this work. Appendix A addresses the naming of Chimera. Chimera’s constraint set is described in Appendix B, and Appendix C explains Chimera’s interface for constraint specification and browsing, as well as its visual notation for constraints. Appendix D presents the heuristics used by Chimera’s macro by example system. All of the chapters of this thesis, with the exception of the last, are self-contained and can be read independently; however to appreciate the interrelationships between the techniques, Chapter 1 and Chapters 3 through 7 are all important.

In the spirit of facilitating repetition and promoting reuse, some of this material has already been published, or will soon be published elsewhere. Many of the ideas presented in Chapter 3 also appear in a SIGGRAPH paper [Kurlander88a], though the text and many of the examples differ. Chapter 4 derives from a CHI paper [Kurlander92a]. The material in Chapter 5 will soon appear in *ACM Transactions on Graphics* [Kurlander93d]. Elements of editable graphical histories have been described in [Kurlander88b], [Kurlander90], [Kurlander91] and [Kurlander93b]. Some of the material from Chapter 7 appears in [Kurlander92b]. Even part of this introduction is included in [Kurlander93a].

Heed my words, Loddfafnir, listen to my counsel;
you'll be better off if you believe me,
follow my advice, and you'll fare well:
never laugh at long-bearded sages!
You may learn a lot listening to the old,
and find wise words in shriveled skins:
among the hides hanging,
among the pelts dangling,
with rennets swinging to and fro.

— *The Elder Edda*, Patricia Terry, tr.

Chapter 2

Related Work

Ignore the rennets swinging to and fro, grab a cup of mead, and consider the wisdom of the field's sages, many of whom, thanks to the relative newness of the subject, have only moderately long beards and mildly shriveled skin.

This chapter summarizes existing approaches to making interfaces more extensible by the end user, and automating repetition in these interfaces. The first section surveys traditional, well-understood techniques for extended and customizing editors of all sorts. It describes techniques that use examples, as well as those that do not, and it classifies editor extensibility mechanisms into a set of five domains. The second section reviews several important experimental systems for extending editors by example. These systems support editing techniques which have yet to migrate into commercial products. The third section describes some important demonstrational systems in other domains. These systems, aside from being important historically, provide context for discussing the new techniques presented in subsequent chapters.

2.1 Editor Extensibility

A simple editor architecture appears in Figure 2.1. The editor interface processes user input into command invocations. Associated with each command is a body of code that performs the necessary computations. This code can create and modify the document objects themselves, as well as accompanying document structure. The command implementations also provide feedback to the user interface, store and retrieve documents from the file system, and send documents to the printer.

This simple architecture suggests a scheme for classifying editor extension *mechanisms* or techniques into a set of *domains* to which they apply. Since the user interface is largely concerned with mapping user input into commands, editor extension mechanisms in the *interface extensibility* domain mainly deal with establishing mappings for new commands, and changing the mappings of old. A different class of extension facilitates the input of document objects. We refer to this domain as *object input extensibility.* Mechanisms that support the actual definition of new commands and the modification of existing ones belong to the *command extensibility* domain. The *document structure extensibility* domain contains those editor extensions that allow new structures to be defined and old ones changed. A fifth domain, *media extensibility*, has no corresponding box in the figure, since it pertains to mechanisms by which new media editors can be added to multimedia editing systems or new media can be supported by existing editors. Each of the next sections focuses on one of these domains, and the extension mechanisms that support it.

2.1.1 Interface Extensibility

There would be little point in allowing the end user to implement new commands, if there were no means of adding these commands to the application interface. Flexible interfaces allow new commands to be invoked, but also permit modifications to the way existing

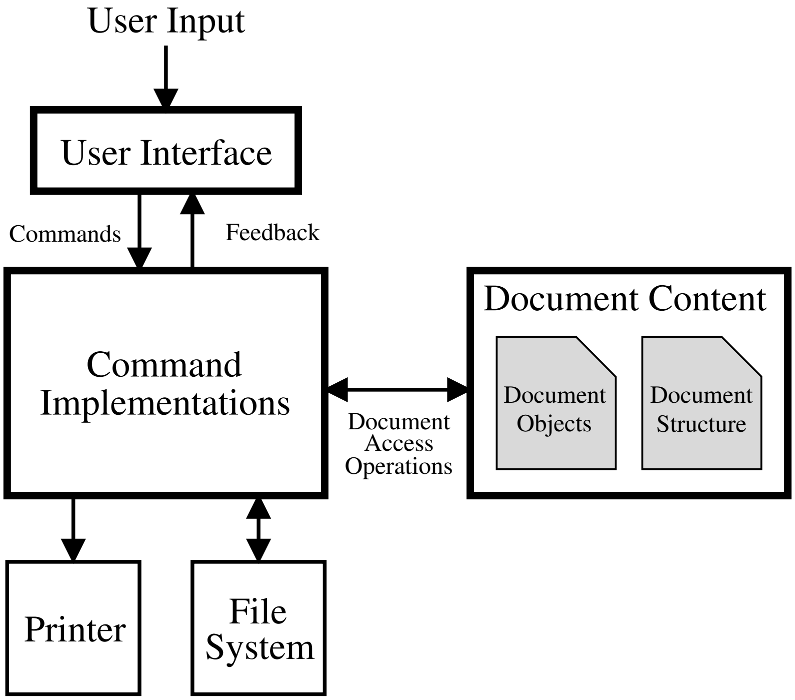


Figure 2.1 A simple editor architecture.

commands are initiated. Text-based interfaces and graphical user interfaces provide alternative mechanisms to support extensibility.

2.1.1.1 Command interpretation

In some editors, users invoke commands by typing the command names directly to an interpreter. Many non-screen-oriented text editors, such as TECO [Digital80] and *ed* [Kernighan78b] exclusively use an interpreter for command input. These editors historically have been popular mainly with programmers, since the task of editing becomes akin to programming. Though this is especially true of TECO, which is a full programming language with loops and conditionals, users of *ed* also need to remember a command syntax, including command names, arguments, and separators. Other editors, such as **`sam`** [Pike87] and *vi* [Joy80b], provide a command interpreter to supplement the screen editing capabilities of the editor. In these editors, the command interpreter is typically used only when other methods of command invocation cannot call the necessary operations, or as is especially the case in **`sam`**, commands need to be connected in such a way that a command interpreter is very convenient.

Extensible command interpreters include a language construct enabling new commands to be defined. GNU Emacs, for example, provides a full Lisp interpreter whereby new commands can be defined and subsequently invoked as though they were primitives [Lewis88]. Though this interpreter can be employed by the end user to launch commands, it primarily serves as a language for programming editor extensions. Section 2.1.3.3 describes extension languages in greater detail. GNU Emacs has a second command interpreter that lacks a full set of language constructs, but is far more suitable as an user interface. Sophisticated users can register commands with this interpreter, along with a set of typed arguments and prompts. When commands are invoked from this interpreter, these prompts fetch arguments so the user need not remember argument count, order, nor syntax. Since the arguments are typed, the system can validate input prior to proceeding, suggest reasonable defaults, and provide automatic argument completion in some cases.

2.1.1.2 Input event translation

A second approach to command invocation maps low-level input events, such as key clicks, mouse button choices, and mouse motions, and high-level events, such as menu selections and command button activations, directly into editor actions. For input event translation to be extensible, the mechanism must allow new commands to be bound to events. Typically the same mechanism can change the bindings of old commands. Screen editors and editors with direct manipulation interfaces usually rely on input event translation since command interpretation provides a lesser degree of responsiveness.

In the days before menus and mice, the keyboard was the source of all input events. Some screen editors with their roots in the days before workstation technology became popular, such as Emacs [Stallman84] and VAXTPU [Digital86b], rely on translation tables to associate key sequences with commands. These are called *keymaps*. Newer systems, such as the Diamond Multimedia Editor [Crowley87], designed from the outset to accept other forms of graphical input, also use this mechanism for dispatching on keyboard events.

When editors are built using a User Interface Management System (UIMS), they often rely on the UIMS to translate events for them. In the Seeheim model of user interfaces, the presentation component converts low-level events into descriptive tokens, and the dialogue control component processes the token stream, generating calls to the third component, the application interface [Green85]. As part of its presentation component, the Cedar environment includes a Terminal Interface Package (TIP) that acts as a lexical analyzer, reading input events such as those directly generated by the keyboard and mouse, turning these events into higher-level tokens such as DELETE and DOREPLACE [Swinehart86]. These tokens pass in turn to a notifier that maps them to appropriate procedures. Having an event translation component in the UIMS reduces redundant code, since event translation must be performed in some way by nearly all interactive applications.

2.1.1.3 Multiple interfaces

Editors with particularly extensible interfaces make it easy to provide multiple front ends to the same internals. A single editor system can mimic the interfaces of several distinct systems, allowing individual end users to adopt editors that appear similar to ones they have worked with in the past, or that are more suitable for the tasks they need to perform. Since much of the main application code is shared, this reduces the amount of redundant implementation, debugging, and maintenance necessary than if supporting completely separate editors.

GNU Emacs, for example, has interfaces to emulate the VAX EDT editor, UNIX (Gosling) Emacs, and two different interfaces that emulate *vi* to varying degrees. Interfaces have been written for the VAXTPU editor to simulate EDT and *vi*. None of these additional interfaces require changes to the underlying editor code—they rely on new commands written in an editor extension language and a set of keymap entries to call these new commands.

The VAXTPU editor is particularly interesting because it has no built-in interface, yet relies on no external user-interface support. When the editor is invoked, an interface in the VAXTPU language is specified, which is responsible for mapping editor windows onto the screen, and filling in keymaps. Two system supplied interfaces can be invoked: EVE (Extensible VAX Editor), and a VAX EDT emulator. New interfaces can be written easily as well. Each of the two system supplied interfaces were written so that they too can be extended. This is facilitated by the VAXTPU **SAVE** command, which adds all the new procedures and keymap definitions entered during the current session to an interface file.

In addition to simulating other editors, GNU Emacs presents many specialized interfaces adapted to particular editing tasks, such as writing programs in LISP, C, or PostScript, composing outlines, or editing plain text. Each such interface in Emacs is called a *major* *mode*. Associated with each major mode is a keymap containing the local key-command bindings for that interface. Thus when a user switches between major modes, active key bindings automatically update to those appropriate for the environment. When no binding is found for a key locally, Emacs' global keymap is consulted. Changing a buffer to a new editor mode executes a procedure that typically creates new buffer variables and sets these and other variables to values appropriate for the new interface. These buffer variables are manipulated by the specialized routines installed in the current keymap. Historically editors with flexible interfaces have been used as quick and dirty interface management systems and interface builders. The Emacs editors particularly have been extensively exploited in this way. UNIX Emacs, for example, has been used as an interface to an electronic mail system, bulletin boards, net news, spreadsheets, an animation program, a database system, a Basic interpreter, and an on-line help system [Borenstein88].

2.1.2 Object Input Extensibility

People spend more time inserting text, graphics, and other objects into documents than performing any other type of editor operation. This section deals primarily with mechanisms for extending the ease of adding objects to editor documents. Typically in these techniques the user extracts information on how the current input task differs from the general input task, and extends the environment to make the current task easier.

The next five sections discuss mechanisms for input extensibility. Adding an object to a document consists of two steps: construction and positioning. Sections 2.1.2.1 through 2.1.2.3 discuss mechanisms that reduce the time necessary for object construction. Sections 2.1.2.3 and 2.1.2.4 consider mechanisms that facilitate the positioning task. Finally Section 2.1.2.5 deals with customizing various input attributes, thereby extending the variety of objects that can be entered.

2.1.2.1 Abbreviation expansion

Abbreviation expansion allows small sequences of objects, usually characters, entered into a document to expand into new sequences. In many text editors, users can define abbreviations for frequently typed, long character sequences. In this respect, abbreviation expansion is similar to keyboard macros, to be discussed later, but the keys in the sequence being expanded are already in the document or in the process of being added to it.

Abbreviation expansion mechanisms differ in whether they expand characters before (vi) or after (Emacs) they have been added to a document, and whether the expansions trigger automatically (vi) or in response to a command (Tioga). Expansions in Emacs and Tioga can retain the capitalization of the document text, and in Tioga, formatting information can be inherited either from the document text or specified in the expansion definition. Tioga's command-based macros can also be included in this definition, to be executed after finishing the expansion [Paxton87]. An additional GNU Emacs facility, called dynamic abbreviation expansion, uses the buffer contents in lieu of explicit abbreviation definitions to determine expansions. If the expansion is not appropriate, the user can cycle through other words in the document which begin the same way, and select the appropriate one.

Abbreviations are commonly used in text editors because users can easily specify a unique abbreviation unambiguously in only a few characters. In a graphical editor it is typically harder to enter a precise shape that can be interpreted unambiguously as a particular abbreviation. Gestures in some interfaces provide elements of this function [Rubine91]. Abbreviation expansion is an elaborate form of search and replace, usually employing multiple replacement rules. When a more sophisticated abbreviation facility is not available, a sequence of search and replace operations can be used for abbreviation expansion. Graphical search and replace, discussed in Chapter 3, can be used in this way for graphical abbreviation expansions.

2.1.2.2 Object libraries

Graphical editor users frequently rely on *object libraries* to extend the set of objects they can easily add to a document. When an object to be added to the scene already exists in a library, the user need only copy it and position it appropriately. This technique is rarely used for small quantities of text, since the process of changing the focus of attention from the document and locating the text in a library is often slower than retyping the text. Also, abbreviation expansion usually suffices, which effectively includes a library plus an automatic instantiation facility. Some editors, such as Ready, Set, Go! [Letraset87], have text libraries called glossaries which bind text to be inserted to command characters, thus accelerating the lookup process. ClickPaste, a product for the Macintosh, allows both text and graphics to be placed in hierarchical menus, which provide fast access to library objects [Mainstay91]. The objects can be pasted into any editor.

Several companies sell object libraries oriented toward particular domains. Such libraries for CAD systems include graphics for mechanical and electrical engineering, residential and commercial architecture, and other purposes. Libraries of illustrations, called *clip art*, are available for draw and paint programs. These often have special themes for applications such as advertising, business, or education. Editors that support object libraries also include facilities to create and extend them. Though objects in libraries are typically complex, they can be inserted in documents essentially as easily as primitives.

2.1.2.3 Templates

Object libraries reduce the effort of adding complex objects to a document, by amortizing the high cost of initial construction over the many times the objects are used. *Templates* also reduce the cost of entering information in documents by providing partially specified input data that can be reused. For the document’s unspecified components, they often reduce the cost of positioning, since special empty fields can be placed in the template in advance. Basically templates are input data with pieces missing. These pieces might be physical components of the objects, such as names and addresses to be added to a form letter, or other style properties, such as fill color and formatting. In the latter case, since most editors do not allow undefined properties, templates may contain default values to be overridden. Clip art and other library objects are frequently used as templates, when copied verbatim from libraries, then modified to have different style properties.

Text editors provide support for templates by introducing empty field elements, placed in the document where additional text needs to be added, and supplying commands to advance the cursor to the next or previous fields. During program composition, abbreviation expansion and templates work together in Tioga to accelerate the construction of Cedar programs. Many Cedar keywords are defined as abbreviations, which expand into templates for the language constructs that they initiate. BibTeX, the bibliography database tool for LaTeX [Lamport86], has two BibTeX modes written for GNU Emacs to provide template support for its many reference types [Chen88] [Martensson88].

Because equation formatting, like equations themselves, tends to be formulaic, interactive equation editing is an application that is well-suited for templates. Equation editors, such as CaminoReal's Meddle [Arnon88] and Expressionist [Bonadio88], provide templates for a large number of mathematical operations, such as integration, summation, and addition, and mathematical constructs, such as matrices. Templates can include style and formatting information, without supplying any accompanying media, such as text or graphics. Most commercial desktop publishing programs have libraries of page templates for specialized document types, like slides, newsletters, brochures, catalogues, and business stationary. These templates include column specifications, style information for various text components, margins, rules, and other page attributes.

2.1.2.4 Grids

Positioning characters in traditional text editors is an easy process because characters are constrained to fall within an implicit rectangular grid. Other types of editors, including graphical editors and page layout editors, also have grids to facilitate the process of positioning. However, since positioning tasks are more varied in these editors they often allow custom grids to be constructed for the task at hand. Generally useful grids can be saved in libraries, extending the positioning tasks that can easily be performed.

Grids in graphical editors control positioning precisely during object construct and transformation. Many graphical editors only supply fixed square grids while others allow grid configurations with customizable distances between rules. Brown University's Picture Layout System [Feiner82a] [Feiner82b] supports three grid classes: rectangular, circular, and perspective, parameterized by aspect ratio, rotation, and offset. The grid itself can be edited in a special mode, and grid configurations can be saved and retrieved.

Bier points out that graphical relationships involving multiple points are often difficult to specify with grids [Bier86]. The ability to create custom grids increases the relationships that can be expressed; however, two other techniques, *snap-dragging* [Bier86] and *constraints* [Nelson85] have greater expressive power. The former technique uses compass and ruler constructs, called *alignment objects*, to precisely specify geometric positioning relationships. The latter allows the specification of high-level geometric relationships to be maintained as the illustration is modified.

Graphic artists use design grids as a tool for constructing page layouts [Müller-Brockmann81]. These grids assist in positioning and sizing page components. Since design grids facilitate the construction of consistent page layouts, well-designed grids should be placed in libraries and reused. Design grids differ from page layout templates, discussed in the last subsection, in that they are less constrained—they specify entire classes of layouts that meet design criteria. In contrast, layout templates specify complete layouts, usually with auxiliary information, such as font styles and sizes to be used. Fewer electronic page layout editors offer grids than templates, perhaps because they require more effort and experience to use; although some systems, such as Ready, Set, Go! support them [Letraset87].

2.1.2.5 Customizable style properties

Many style properties, such as line pattern and fill color, have a vast range of potential values. Those values included in default menus reflect a set that the implementers consider useful; however, no set suffices for all applications, so flexible editors allow new property values to be defined, and then selected as easily as the defaults.

Text formatters often allow additional fonts to be loaded and added to the font menu. Many editors allow fonts to be scaled to custom sizes, and the line leading to be set to arbitrary values. Though many of these text properties can be modified directly, it is usually better to add an extra level of indirection and assign text a logical style, such as Caption Text or Subheader, and then choose style properties for the logical style. This extra level of indirection, discussed further in Section 2.1.4.3, promotes consistency of style properties, and facilitates changes throughout a document.

2.1.3 Command Extensibility

All of the mechanisms described in the last section add a degree of extensibility to an editor, but none make fundamental changes to the set of tasks that can be performed. This section describes methods of constructing entirely new commands, and modifying the behavior of existing ones. Two different mechanisms can be employed to define new commands: macros and programs. Macros typically allow groups of existing commands to be executed together as a single operation, so instead of providing new functionality, they make tasks that can already be accomplished easier. In contrast, programs can exploit operations often not included in editor command languages, such as iteration, arithmetic calculations, and conditionals, and can thereby perform more complex computations.

2.1.3.1 Interface event macros

Perhaps the simplest form of editor macro is the *interface event* macro, which plays back a sequence of events that in turn undergoes input translation and invokes commands. *Keyboard macros* are a common specialization of these macros that expand only to keyboard events. There are several reasons why interface event macros are enticing. First, they are extremely easy to implement—since editors already have a means of translating input events, implementers need only add a logging and playback mechanism. Second, users can construct keyboard macros by demonstration, so this technique is accessible to an editor's entire user-population.

Interface event macros have several drawbacks. Without an accompanying interface translation specification, these macros contain no semantic information. For example, if the current keyboard translation changes, the meaning of keyboard macros changes as well. An interface event macro that executes as desired in one user's environment very likely will execute differently in another's, making these macros difficult to share. Since users may modify their keyboard mappings over time, a keyboard macro saved last month may no longer work today, making these macros poor entities to archive. Also keyboard macros can be cryptic to edit, since there is a layer of abstraction between the characters being examined and the commands to which they map.

For these reasons, more robust macro facilities refer directly to editor commands. Tools can translate interface event macros to command-based macros, but these often need extensive information about the key bindings and system state when the macro was generated, as well as a means of extracting arguments from the input stream and associating them with commands. When editor commands interactively prompt for additional information, it can be tricky to filter this information out of the input stream.

2.1.3.2 Command-based macros

Command-based macros differ from interface event macros in that they expand directly to a set of editor commands. Unlike keyboard macros, they can have parameters and various macro directives or macro-time statements that control expansion. Command-based macros in editors are often difficult to distinguish from procedures. Emacs is so named because it was intended to be a set of Editing MACroS for the TECO editor. Richard Stallman later wrote that Emacs is a misnomer because his editor was implemented as a set of functions in the TECO language [Stallman84], yet the TECO manual refers only to macros, never functions [Digital80].

2.1.3.3 Command programming

The only way to give an editor radically new functionality is through programming. All of the techniques for extensibility discussed in this chapter rely on existing mechanisms to provide their extensibility. Only programming itself can create entirely new mechanisms. There are widely differing viewpoints on what kind of programmability editors should support, and the languages that are best used. We refer to a programming language used to define application extensions as its *extension language*. The Andrew EZ editor's extension language is C, together with additional object-oriented constructs. GNU Emacs is extensible in Emacs Lisp (elisp), UNIX Emacs in Mock Lisp (mlisp), Zmacs (the Symbolics Lisp Machine editor) in Zeta Lisp or Common Lisp, and the original ITS Emacs in TECO. The Tioga editor can be extended through its Cedar language interfaces. The

VAXTPU editor can be extended with the TPU language. This list of extension languages is by no means exhaustive. The rest of this section describes issues in choosing an extension language, and how they impact extensibility.

One dimension along which extension languages differ is their size. The advantage of small languages is that they are easy to implement, learn, and use. If the tasks that are to be performed can be expressed easily in a simple language, then it is to everybody's advantage to minimize the language's size. Unfortunately it is impossible to foresee all future extensions when the extension language is being chosen, so an informed decision is necessary to balance simplicity and flexibility.

Mock Lisp and Emacs Lisp, extension languages for UNIX Emacs and GNU Emacs respectively, make an interesting comparison because they reflect two divergent views on language size, though the editors themselves are similar. Mlisp is a simple Lisp-like language, though it lacks many Lisp fundamentals, such as lists and atoms. In a survey about this extension language, Borenstein and Gosling found that though users praised its simplicity, this and its incompleteness were also the cause of much criticism. Particularly, users complained about the lack of data types (such as arrays, floats, pointers, and structures), the method of referencing parameters, and the poor variable scoping. Borenstein and Gosling conclude that most of the problems in mlisp result from the implementer thinking of it only as an extension language and not a complete programming language. They write, "This is probably the most important lesson to be learned from mlisp: extension languages are real languages, not toys" [Borenstein88].

Extension languages also vary in whether they are compiled or interpreted. If execution speed is not critical, interpreters are generally superior to compiled languages for editor extensions. Interpreted languages allow a faster cycle time for testing and modifying extensions. Code can be dynamically loaded into a running editor, so the editing process does not undergo major interruptions whenever a new extension needs to be installed. Because extensions can be loaded on demand, run-time memory requirements may be diminished if there are large, rarely used extension libraries. Disk requirements may be reduced, since a single editor executable suffices for all users, regardless of the extensions they require. When adding new extensions to a compiled system, all of the relevant editor and extension object files need to be linked together, which demands greater effort and expertise than loading interpreted extensions. If a compiled extension language is chosen, both the editor and extensions must be compiled anew for each new instruction set. If the extension language is interpreted, the editor alone must be recompiled.

The major disadvantage of interpreters is that they must remain in memory during execution time. If the interpreted language is large, then the interpreter will be too. Interpreted extension languages have proven to be sufficiently fast for text and graphical editors. A reasonable alternative to interpreted extensions is dynamically linking and loading compiled extensions. These have the speed of compiled programs (which they are), and many of the conveniences of interpreted code. The Andrew EZ editor can dynamically load compiled C extensions, although some criticize the process as being too awkward for spur-of-the moment customizations [Borenstein88]. An editor in the Emacs family, called SINE, dynamically loads extensions in a compiled Lisp-like language [Stallman84]. VAXTPU, which has its own custom compiled extension language, can also load procedures dynamically.

When designing an extensible editor, the system architect must also decide whether to choose an existing language for extensions. If an existing language fulfills all the requirements, the principle of parsimony suggests that it be exploited. Programmers already familiar with the language will avoid the trouble of learning a new one. The labor associated with designing, implementing, and documenting a new language is eliminated. Existing languages frequently have utilities such as source-level debuggers and profilers, and often an entire environment to support the programming process. When extension languages are custom written for an editor, these important tools are neglected more often than not.

Environments such as that of the Lisp Machine [Symbolics86], Smalltalk, and Cedar [Teitelman84], all allow extensions to be coded in their host language. Editors employing existing languages for extensions need only supply a programming interface.

2.1.4 Document Structure Extensibility

Documents typically have several different types of structure. In the text domain there is an underlying syntactic structure, consisting of such constructs as words, sentences, and paragraphs. Graphical documents also contain a syntactic structure, consisting of primitives, such as lines, and collections of these primitives, such as polylines. Logical structure records the high-level role various elements play in composing a document, such as chapter headings, picture captions, and body text. Documents can also have a physical structure, such as the page structure, that is independent of logical structure. Sometimes object hierarchies are constructed in graphical editors to accelerate hit-testing and rendering. These physical structures need not conform to logical graphical structures.

Structure is used by editor operations to limit processing to particular elements, such as left-justifying only the current paragraph, or to aid in positioning the cursor, such as moving forward a word. Document structure also facilitates manipulating and changing the attributes of related objects. Existing document structures often do not suffice for a given editing task, so flexible editors provide mechanisms for defining new structures and changing old definitions. The next three subsections describe popular mechanisms for extending document structure: syntax specification, grouping and instancing, and styles.

2.1.4.1 Syntax specification

The command to move the cursor ahead one word should behave differently in Lisp than in C, because the two languages define identifiers differently. One solution would be to change the forward-word function in the interface when the language being edited requires it. A more elegant approach uses the same function throughout, but relies on a flexible structure specification stored as data. Editors with this capability allow their view of a document's syntactic structure to be redefined without programming, so they are more readily extended to new editing applications.

The VAX EDT editor allows the default syntax for words, sentences, paragraphs and pages to be modified with commands that change the delimiters [Digital86a]. GNU Emacs stores syntax information such as word-constituent characters, opening and closing delimiters (used for example in parenthesis and bracket matching), quote characters, whitespace, and comment delimiters in *syntax tables* [Stallman87]. The *vi/ex* editor has commands to advance the cursor to the next paragraph or section, as determined by a string search for particular troff macros [Joy80a]. Since troff is itself extensible, the editor has user options to declare which macros are relevant.

2.1.4.2 Grouping and instancing

The two mechanisms discussed next appear most frequently in graphical editors, though they are relevant to others as well. Both techniques facilitate the task of making identical changes to multiple related objects. *Grouping* is a mechanism to cluster multiple objects together as a logical unit, sometimes associated with a name. In the most general systems, objects can belong to multiple groups. Entire groups can be selected with a single operation, and all the constituent objects modified together. Changing a graphical property, such as line width or fill color, for all objects in a group is approximately as easy as changing the attribute for a single object. Most commercial graphical editors, such as MacDraw [Claris88] and Adobe Illustrator [Adobe90], have a grouping facility.

Instancing is really a special case of the object libraries mentioned in Section 2.1.2.2. Instead of copying an object from a library, a reference is inserted into the document pointing back to the original. Thus multiple instances of an object may appear in a document, but changes to the original effectively propagate to each. Ivan Sutherland's Sketchpad system, the earliest interactive drawing system, included this mechanism [Sutherland63a].

2.1.4.3 Styles

In the publishing world, style specifications have long been used to promote consistency of form within and among documents. Styles used in document editing programs support this same goal, but also reduce the effort to change all appearances of a logical component. Editors with a style mechanism employ an extra level of indirection. Text is assigned a style, and an associated style rule specifies the textual style properties. For example, a style rule called Picture Caption might have the attributes: Bodoni family, bold face, 10pt, centered text. All document text with the Picture Caption style would be set the same way. Users can define new style rules as they create new logical components, so styles embody an extensible form of document structure.

Styles bear similarities to both instancing and grouping. Styles can even be considered a form of instancing, but whereas most instancing mechanisms employ library objects, style systems rely on libraries of attribute settings. Grouping is a technique for associating multiple objects together so they can be manipulated and modified together, which is also a motivating factor for styles. Johnson and Beach outline the history of style rules, and survey a set of design issues to be considered when building a style mechanism [Johnson88]. Many modern formatter/editors have style support, including Tioga [Beach85], PageMaker [Aldus90], FrameMaker [Frame90], Microsoft Word [Microsoft91b], EZ [Palay88], and Ready, Set, Go! [Letraset87].

2.1.5 Media Extensibility

Multimedia editing systems integrate such disparate objects as text, illustrations, scanned images, voice, and spreadsheets. Since the number of potentially useful media types is vast, architects of these systems typically build them so new classes of media elements can be added without significant modifications to the editor. Editors with a framework that readily supports additions of this type exhibit *media extensibility.*

Editors capable of producing multimedia documents typically consist of multiple component editors, one for each media type, integrated in either a strong or weak fashion. As discussed in [Crowley87], strongly integrated editors allow different media objects to be viewed and edited in the same window, execute under a single process, present a uniform user interface where possible, and may even permit easy conversions among media types. In contrast, weakly integrated media editors each run in their own window as separate processes, with no single, uniform interface.

The Diamond Editor [Crowley87], originally built for a multimedia message system [Thomas85], the EZ editor of the Andrew Toolkit [Palay88], and Quill [Chamberlin88] are all strongly integrated multimedia editors. In each of these, the editor's framework determines which media elements interact with each other and the general editor services. This dictates a particular set of procedures or methods that must be supplied by a new media type for it to snap into the system. These procedures, called external procedures in Quill and generic media functions in Diamond, typically provide functionality to create and destroy new media objects, pass them input events, select and deselect them, print them, perform file I/O on them, and execute operations related to refresh. In addition to these common routines, each media extension defines commands specific to editing its own media type. Multimedia systems can follow several different conventions in deciding which media-specific command set is active. When a media element is selected in the Diamond editor, a keymap and command menu for that particular media type is activated. Quill and Andrew allow media elements to be nested. In Quill, the current editor is selected by clicking the mouse near the relevant object and walking up the hierarchy if necessary. Command buttons for the current editor's media-specific operations are displayed in one pane of the editor window. In Andrew, each parent media object determines which input events to pass on to its children, and which to interpret as media-specific commands.

More loosely integrated editors, not sharing the same address space, must rely on protocols for inter-editor communication. An example of this is Chen's VorTeX system [Chen88], which contains a source editor for editing TeX code directly, a formatter, and a target editor for viewing the formatted page. Additional media editors, executing as independent processes in separate windows, can be incorporated into the system by defining new protocols between them and the source and target editors. A more loosely integrated multimedia editing environment is that provided by the Macintosh. To transfer a media element from one editor to another, the user cuts the element, placing it into a temporary file, and then pastes it into another editor. Again, a protocol must be agreed upon between the editors sending and receiving the media element; here that protocol is the PICT data format [Jernigan88]. Using Apple's Publish and Subscribe protocol, multiple media editors can automatically notify one another of document changes [Meadow91]. This capability is also present in Microsoft's OLE 2.0 system, which permits multiple media classes to be embedded and edited within the same window [Microsoft92].

2.2 Experimental Systems for Editing by Example

This section describes nine research systems that use novel demonstrational techniques to facilitate editing of either textual or graphical documents.

2.2.1 Juno

Greg Nelson's Juno graphical editor, like the systems described in the next two sections, is a two-view editor [Nelson85]. Juno provides a WYSIWYG graphical view of an illustration, and a source view that contains a procedural description. Users can define procedures demonstrationally by executing direct manipulation commands in the graphical view, which generates code in the procedural view. Changes made to either view propagate to the other.

Illustrations and procedures in Juno rely on constraints to establish precise geometric relationships. The editor allows four different types of constraints: congruent (equal lengths), parallel, horizontal, and vertical, and uses Newton-Raphson iteration to solve non-linear systems. Dijkstra's guarded commands inspired Juno's procedural language. The system first finds bindings for the variables that satisfy the constraints, and then executes graphical commands that can reference these bindings. Variables in Juno are restricted to 2D points, and only these points can serve as parameters to procedures.

During the drawing process, the user distinguishes between *independent variables*, used as input by the constraints, and *dependent variables*, produced as output. When the user changes the current command into a procedure, the independent variables automatically become parameters. Procedures can be saved in the environment and call one another, though there is no way to take a previously defined procedure and edit it from the graphical view. Juno's procedures belong to the object-input and document structure extensibility domains—they can add new objects and constraints to the scene, and nested procedures impose a procedural hierarchy on the scene description.

2.2.2 Tweedle

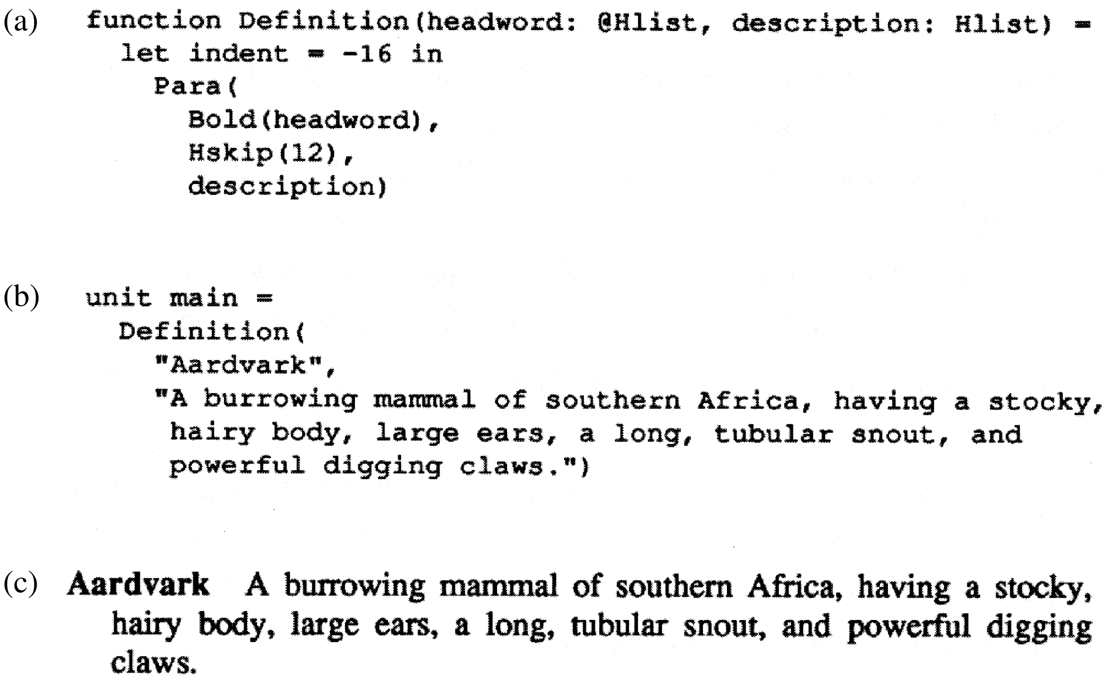
The Tweedle system, built by Paul Asente, consists of two components: the Dee graphical editor and the Dum graphics language [Asente87]. Like Juno, Tweedle is a two-view system, and modifications made to one view automatically propagate to the other. Programs in the Dum language serve as the internal representation of objects in the graphical view—the graphical editor contains no redundant data structures to be kept consistent with the program. As a result, it is critical that Tweedle supports incremental execution, so that when graphical operations modify a small portion of the scene, the dependent sections of code can be quickly identified and reinterpreted. Accordingly, Asente designed Dum so that a simple static semantic analysis of the code indicates dependencies, and subroutines have no external side effects. To make the code fast to parse, the syntax resembles Lisp, though the language is more imperative and includes an imaging model inspired by PostScript.

Object definitions in Tweedle are simply subroutines that specify how to draw objects, and these definitions can be hierarchical. Tweedle supports instancing by allowing multiple calls to a single object definition, but it also permits *variants*, another form of extensibility in the document structure domain. When users edit pre-existing objects in the graphical editor, they can choose to modify the original definition, create a new definition, or define a variant. Variants are object definitions expressed in terms of differences from a parent object definition. Changes to the parent definition propagate to its variants, except as explicitly overridden by the variant definition. Variants first execute their parent definition, followed by code that modifies subobjects defined by the parent, possibly adding, deleting, transforming, or changing the overlap order of these components.

2.2.3 Lilac

In Kenneth Brooks' two-view document editor, Lilac, logical document structures are also defined procedurally [Brooks88]. Corresponding to each use of a structure in the formatted *page view* is a procedure call in the algorithmic *source view.* An example from Brooks' thesis appears in Figure 2.2, demonstrating how a simple dictionary entry structure might be defined and used. Figure 2.2a displays the user-defined document element, called Definition, as a function taking two arguments, the headword, which is the term being defined, and the description. Note that formatting instructions appear within the function's body. The main program in Figure 2.2b contains a call to this function, and the result, as it appears in the formatted view, is given in Figure 2.2c.

The idea of formatting a document with directives specifying logical structure rather than typesetting information was pioneered by Brian Reid's Scribe system [Unilogic85]. Brooks' contribution involves integrating this technique into a two-view editor, and providing interactive support for inserting structural instances into the page view. Lilac



(a)

```
function Definition(headword: @Hlist, description: Hlist) =
  let indent = -16 in
    Para(
      Bold(headword),
      Hskip(12),
      description)
```

(b)

```
unit main =
  Definition(
    "Aardvark",
    "A burrowing mammal of southern Africa, having a stocky,
     hairy body, large ears, a long, tubular snout, and
     powerful digging claws.")
```

(c) **Aardvark A burrowing mammal of southern Africa, having a stocky, hairy body, large ears, a long, tubular snout, and powerful digging claws.**

Figure 2.2 Document extensibility in Lilac. Reprinted from [Brooks88]. (a) A new structure element definition; (b) an invocation of this structure element; (c) The resulting formatted text.

permits either of the two views to be edited, and changes made to one automatically propagate to the other. A vast majority of editing is done in the page view, though new structural elements, such as Definition, must be supplied procedurally. Nevertheless, once a new element has been specified, several commands in the page view can take advantage of the user-defined document structure. The return key invokes a function that replicates the nearest ancestor of the structure being edited, subject to certain criteria, and places the insertion point in its first text field. Assuming a Dictionary element type was defined to hold a variable number of Definition elements, if we finish filling out one definition and hit return, then a new Definition structure is created directly below the last, and the insertion point positioned so we can begin filling it in. The tab key advances the insertion point from field to field, and after we finish typing the headword of the new Definition, a tab will move the cursor to the description field. The mouse can be used to select arbitrary elements of the user-defined document structure by selecting a leaf in the hierarchy, and promoting the selection to its ancestors if necessary.

Function definitions in Lilac replace style rules in more conventional editors. Modifying the formatting information in the body of a function reformats all of the corresponding structure elements in the page view. One of the design issues discussed in [Johnson88] is which structural elements should be associated with styles. In most editors with style rules, rules can be applied to only a few object elements, such as characters and paragraphs. Brooks' system is particularly interesting in that it is a highly-interactive editor that allows new structural elements to be defined, as well as procedural definitions of the elements' styles. Brooks' system has impact on input extensibility as well, since his structural elements are a kind of template. It should be noted that Paul Asente's two-view graphical editor uses function definitions and invocations in a similar way. In both systems, function definitions specify structural elements, and invocations declare instances of those elements.

2.2.4 The U Editor

Robert Nix built an editing by example component for the U text editor to automate repetitive textual transformations [Nix83]. His system takes examples of text sequences to be found, and examples of how they should be changed, and synthesizes a procedure to find similar text and make similar changes. Instead of monitoring the user performing the transformation, and producing a generalized procedure based on the *trace* or set of commands invoked to perform this task, Nix's algorithm builds procedures directly from *input and output pairs*. The advantage of this is that the user need not repeat the same ordering of commands, or even the same commands at all, when providing multiple example transformations. Also since the editing by example system relies on no particular editor command set, the editor can easily be extended to include new commands and the system readily ported to other editors. The disadvantage of relying on input and output pairs is that it ignores a potentially rich source of information in the execution trace.

Out of these input and output pairs, Nix's system builds *gap programs*, which have the form $G \Longrightarrow R$, where G is a *gap pattern*, and R is a *gap replacement*. Conceptually, a gap pattern is a set of constant strings separated by single gaps, which are variables that can match most strings. Gap patterns form a sub-language of regular expressions. A gap replacement is a mix of constant strings and gaps. For example, the following is a gap program: (-1-) q -2- — -3- . ⇒ -1- — -2- — -3- . , where gaps are identified by -*n*- (*n* is an integer, and -1- is the first gap) and q represents the space character [Nix83]. This program finds all telephone numbers of the form "(212) 316–9369." and changes them to the form "212–316–9369.".

Though Nix proved that determining the existence of a gap program for a set of three or more input and output pairs is NP-complete, he developed a heuristic-based algorithm that usually finds gap programs quickly. The algorithm requires more input and output pairs to converge on the desired program as the number of gaps increases, the constant sequences become shorter, and the gap substitutions become longer. An additional set of heuristic techniques that Nix implemented reduces the number of input and output pairs typically required.

The interface of this editing by example system allows programs to be tested on sample text, and the changes to be undone if necessary. Additional input and output pairs can refine an existing program. Users can view gap programs synthesized by the system, and edit them directly if they have the expertise.

2.2.5 TELS

Ian Witten and Dan Mo took a different approach in building a system for specifying textual transformations by demonstration, in that their editor, TELS, builds procedures from interaction traces rather than input and output pairs [Witten93]. Witten and Mo argue that synthesizing procedures from traces better allows exceptions in the transformation rules, and permits procedures to be learned from less regularly structured text. For example, both TELS and the U editor can learn by example how to take an address list, with one address per line, and add carriage returns after commas so that each component of the address is on its own line; however, only TELS can learn the exception that street names including “, N.E.” or “, S.E.” should not be split after the comma.

To reduce the number of commands TELS must reason about, it uses a simple textual computation model with four operations: insert, delete, locate, and select. Most higher-level commands can be expressed in terms of these. Since users left on their own might not repeat interaction steps in the same order, complicating the loop inference step, TELS builds procedures after the first demonstration, and then invokes them on subsequent text. Users debug procedures interactively by indicating steps that should or should not have been performed on the new examples.

To support the generalization step, TELS records a large amount of context with every command recorded in the trace. Insert and delete commands store the string being inserted or deleted. Locate and select commands store the text surrounding the cursor position or selection, the lexical position relative to the current unit (e.g., nth character in a word, nth word in the paragraph), as well as the lexical position relative to the previous position (e.g., 2 words forward). This information is generalized when collapsing multiple trace steps to the same program step, to derive the intent of the operation. TELS can make overgeneralization and overspecialization mistakes, both of which can be corrected by presenting additional examples.

2.2.6 Tourmaline

Tourmaline differs from the U editor and TELS in using demonstrational techniques to facilitate text *formatting*, rather than textual transformations [Myers91b]. Brad Myers’ system uses three techniques to achieve consistent formatting in a document. The first technique defines composite styles, multiple style elements appearing together, by demonstration. For example, a chapter header might include the string “Chapter”, a decorative rule, a chapter number, and a name. Each of the text elements might have its own font and size. A user draws these elements on the page, using a WYSIWYG editor, and declares it an example of a new composite style. Tourmaline parses the example into its components: a reserved constant string “Chapter”, constant decorative elements, a number (interpreted as a count in a sequence), and parameterized text (the name). To add new chapter headers in the same composite style, users type only the parameterized text and indicate the style class. The system automatically inserts derived elements, such as constants and counts, and formats all elements according to the example. This mechanism extends the set of objects that can be easily added to a document, as well as the document structure.

Users of Tourmaline also specify table styles by example. The user draws an example table, from which the system infers the formatting of its components. For instance, strings that are nearly left justified are assumed to be left justified, while nearly centered strings are assumed centered. Columns or rows with nearly equal dimensions are considered to be equal, and nearly connecting lines are automatically connected. In addition to beautifying the table, these inferences can be applied to create new tables of the same style.

The third demonstrational component of Tourmaline formats bibliography entries by example. To instruct the system how a particular reference class should be formatted, the user formats an example from a Scribe or Refer structured bibliography database. The system infers how each of the components should appear, including their ordering, the presentation of the author names, and their typesetting characteristics. The system then automatically creates formatted bibliography entries for other references in the database.

2.2.7 Metamouse

David Maulsby’s Metamouse system [Maulsby89a] [Maulsby89b] [Maulsby93a] provides a macro by example capability for graphical editors. Metamouse works in conjunction with A.sq, a simple graphical editor that manipulates lines and boxes. The editor has a special teaching mode in which an iconic turtle jumps to the cursor location between commands. The user is told that the turtle pays attention to touch relations involving either it or an object in its grasp. As objects are manipulated, the touch relationships highlight to indicate whether they are considered important or insignificant. A set of heuristics determines this classification, but the classification can be overridden by toggling the feedback with the mouse. The system stores these touch relationships as *postcondition* constraints for the saved operation.

As operations are executed in teaching mode, the learning subsystem builds a graph representing the program. The graph contains cycles for loops and branches for conditionals. An integral component of this graph building phase involves matching the current operation against previously observed ones. A new step matches an older one if its operation and constraints are the same, and Metamouse generalizes graph nodes by dropping constraints that it deems unimportant. Immediately upon detecting a loop, Metamouse tries to execute it. This reduces the possibility that auxiliary, irrelevant operations may be interspersed in a loop demonstration. In order to determine whether the next node of the graph can be executed, Metamouse checks to make sure that its postconditions can be achieved from the current state. Thus a loop will terminate when a set of postconditions cannot be satisfied by Metamouse’s constraint solver.

Each touch constraint has several degrees of freedom, including which objects are touching, which portions of these object, and which positions. Touch constraints can be completely specified (*determined*), or partially specified (*strong* or *weak*). In the case of partially specified touch constraints, the path of the turtle helps the constraint solver choose a particular solution. For example, if one of the touch relationships is not complete because it lacks an object specification, this object becomes a *variable*. The constraint solver will follow the turtle's path looking for an object that can instantiate the variable.

Although Metamouse is limited to inferring touch relationships, its users build objects, called *tools*, to help make non-obvious relationships explicit. Metamouse works in the command extensibility domain, and the turtle can be taught to perform some surprisingly complex tasks, such as sorting rectangles by height.

2.2.8 Mondrian

Henry Lieberman's Mondrian is also a graphical editor with a programming by example component [Lieberman93b]. It was built concurrently with Chimera's macro by example facility, and uses a visual representation inspired in part by Chimera's editable graphical histories. Mondrian provides a storyboard of screen miniatures, showing the changing state of the display during editing. Each panel contains a single command, and the history cannot be edited. Command buttons in Mondrian appear as before and after dominoes that iconically portray the effect of executing the command. When users demonstrate new programs by example, the editor automatically adds a new domino to the existing set, giving new commands the same presentation and invocation interface as existing ones. Since Mondrian's programming by example facility expands its command set, it works within the command extensibility domain.

Mondrian users can examine two other representations of demonstrated programs. They can view the Lisp code that Mondrian synthesizes from an example, and they can listen to a natural language listing of the program steps, uttered by a speech synthesizer. Spoken language can be used as a supplementary input mechanism for Mondrian. Alan Turransky developed a speech input mechanism for Mondrian, that allows users to verbalize their intent while performing inherently ambiguous direct manipulation commands [Turransky93]. Speech input and direct manipulation work well together because they are largely complementary [Cohen92]. By pointing with a mouse, people can easily select particular object instances, while speech better communicates abstractions.

2.2.9 Turvy

Like Mondrian, David Maulsby's Turvy experiment uses speech and graphical interaction together to synthesize a procedure by demonstration [Maulsby93b]. Turvy is actually a *Wizard of Oz experiment,* meaning that a human being plays the role of the computer during the tests, to gather human-interaction data without the cost of building an actual implementation. Though Turvy's inferencing model was designed to be domain independent, most of the experiments involved repetitive bibliography reformatting tasks within

Microsoft Word, a combination text editor/formatter. In contrast with Tourmaline which also automates repetitive bibliographic tasks, Turvy restricts itself to low-level syntactic information, like words, paragraphs, separators, and font style, and has no knowledge of higher-level domain concepts, like author and publisher.

Turvy learns from multiple examples. Like Metamouse and TELS, the user provides an initial example, and the system predicts subsequent actions. If the user corrects these actions, Turvy modifies the inferred procedure to take the new data into account. In this experiment, users can point to features of an example that indicate it must be treated specially, and Turvy biases its generalizations to take these features into account. This type of "focusing" can also be done verbally. Turvy's experimental subjects rarely used pointing gestures, but verbal focusing was much more common. The experiment helped refine Turvy's dialogue and inference model, and it was noted that Turvy's subjects adopted its verbal terminology, and quickly learned to refer to those syntactic features that Turvy mentioned in its speech.

2.3 Other Demonstrational Systems

Not all research in demonstrational interfaces has focused on editing systems. This section describes six systems that take an innovative approach to programming by example outside of the editing domain.

2.3.1 Pygmalion

David C. Smith's Pygmalion is the granddaddy of all programming by example systems, and inspired much of the subsequent research in the field [Smith77] [Smith93]. Smith built Pygmalion as an alternative to traditional non-interactive, abstract programming languages. People using Pygmalion construct programs by drawing pictorial, analogic representations of the computation, using real sample data. In Pygmalion, programmers construct movies of their functions' computations. The first frame of a movie includes the arguments of the function being defined, and the last frame returns the function's results. In between, the programmer demonstrates the steps of the function by creating icons on the screen, and dragging old values and new into these icons.

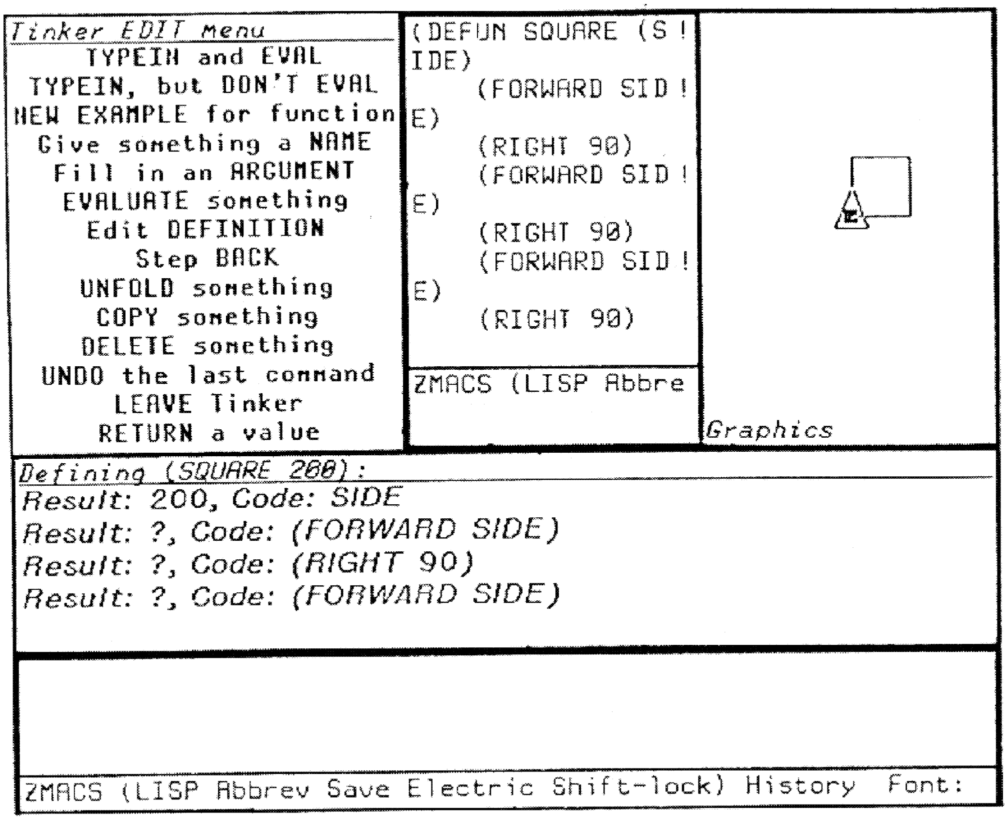
Icons in Pygmalion are simple graphical representations for code and data together. For example, a multiplication icon has two place holders (or icons) for its arguments, with a multiplication sign between them. A conditional icon has an icon for the test, with arrows pointing to additional icons for the true and false branches. Incompletely defined functions in Pygmalion can still execute. After the user supplies arguments to a function call, the function proceeds until it reaches a section that has not yet been defined, and Pygmalion halts for the missing steps to be demonstrated. Conditionals pose a challenge for programming by demonstration system, since only one branch of a conditional can be traced for each evaluation of the test. In Pygmalion, the user demonstrates a branch's computation the first time the branch is taken.

As users of Pygmalion define functions, Pygmalion records the users' demonstration steps into a log. These steps include actions meaningful to the computation, like dragging a number into an arithmetic icon, as well as purely aesthetic operations, like moving an icon from one part of the screen to another. When a function is invoked, Pygmalion plays back both types of steps in an animation, keeping the visual presentation identical to that demonstrated during definition, with the exception of data values that can differ.

2.3.2 Tinker

Motivated by the observation that people not only learn better by example, but that they *teach* better by example too, Henry Lieberman built a system to facilitate the construction of Lisp programs through examples. His programming aid, called Tinker [Lieberman84] [Lieberman86] [Lieberman93a] allows people to develop Lisp code as they manipulate real data. Tinker is especially useful in developing graphics programs, since it displays the effects of graphics commands interactively.

The Tinker workspace consists of five windows, as shown in Figure 2.3. The upper left window (Edit Menu) provides a static menu of commands. The upper right window (Graphics Window) contains the graphical output of the Lisp program being developed. In between these two windows lies the Function Definition Window, which shows the Lisp



```
(DEFUN SQUARE (S !
IDE)
    (FORWARD SID !
E)
    (RIGHT 90)
    (FORWARD SID !
E)
    (RIGHT 90)
    (FORWARD SID !
E)
    (RIGHT 90)
```

Defining (SQUARE 200) :

```
Result: 200, Code: SIDE
Result: ?, Code: (FORWARD SIDE)
Result: ?, Code: (RIGHT 90)
Result: ?, Code: (FORWARD SIDE)
```

Figure 2.3 The Tinker Screen. Reprinted from [Lieberman86].

functions built by Tinker. At the bottom of the screen is the Typing Window in which the user types Lisp expressions. The final window (Snapshot Window), in the middle of the screen, is where the user builds these expressions into functions.

The Snapshot Window maintains a correspondence between code that has been entered, and its value in the current context. New Lisp expressions added to this window can rely on code and values appearing on previous lines. As programmers fill in arguments to function calls, they work with the current values of the arguments. Tinker maps these values to the expressions that generated them, and uses them in constructing its code view. Since LISP code often consists of deeply nested functions, Tinker can help make difficult programming abstractions more concrete.

In Tinker, another type of refinement process adds conditionals to Lisp functions. When two examples of the same function are developed that contain different code, Tinker asks for a boolean expression that distinguishes between the two. It then wraps a conditional around the relevant expressions, using the given test.

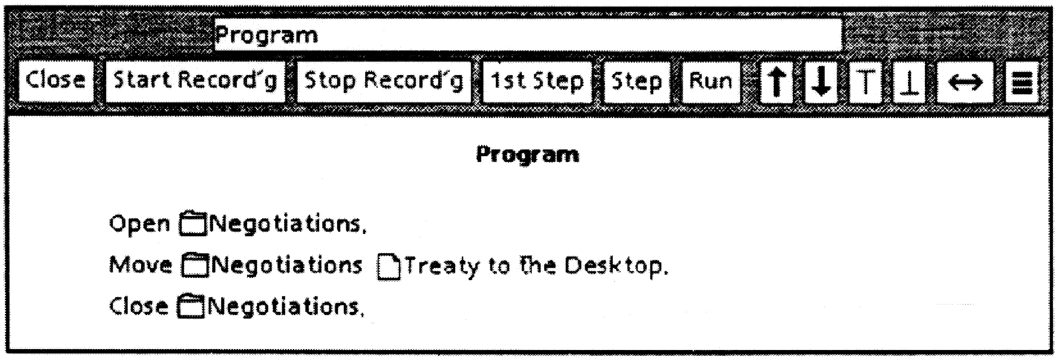
Lieberman points out that Tinker facilitates example-based programming in three ways. If a system is to generate procedures from examples, it must generalize. Tinker does this mainly through its mapping of values to the code that generated them. Multiple examples, representing different cases, can be developed independently and combined to form further generalized code. The system allows functions to be developed, a piece at a time, through refinement. Examples of programs that Tinker has assisted in generating include an alpha-beta search implementation and a pong video game.

2.3.3 SmallStar

The Star user interface pioneered the desktop metaphor as a means of manipulating and interacting with files on a computer system [Smith82]. A small prototype version of this interface had been built called SmallStar, and for his doctoral thesis, Daniel Halbert devised a mechanism for programming this system by example [Halbert84] [Halbert93].

To begin defining a program, the user either invokes a StartRecording command, or opens a program icon. As users perform operations, a text and icon-based description of these commands appears in the program window. References to objects acting as arguments to these operations also become part of the command listing. Figure 2.4 shows an example from Halbert’s thesis. This listing represents the sequence of operations in which a folder named *Negotiations* is opened, a file named *Treaty* is moved from this folder onto the desktop, and the folder is then closed.

Note that the operands in this listing are all named constants. It might be the case that the user performed this set of operations with a slightly different intent in mind: perhaps he wanted to define a macro that moved the first file in the folder to the desktop, and although

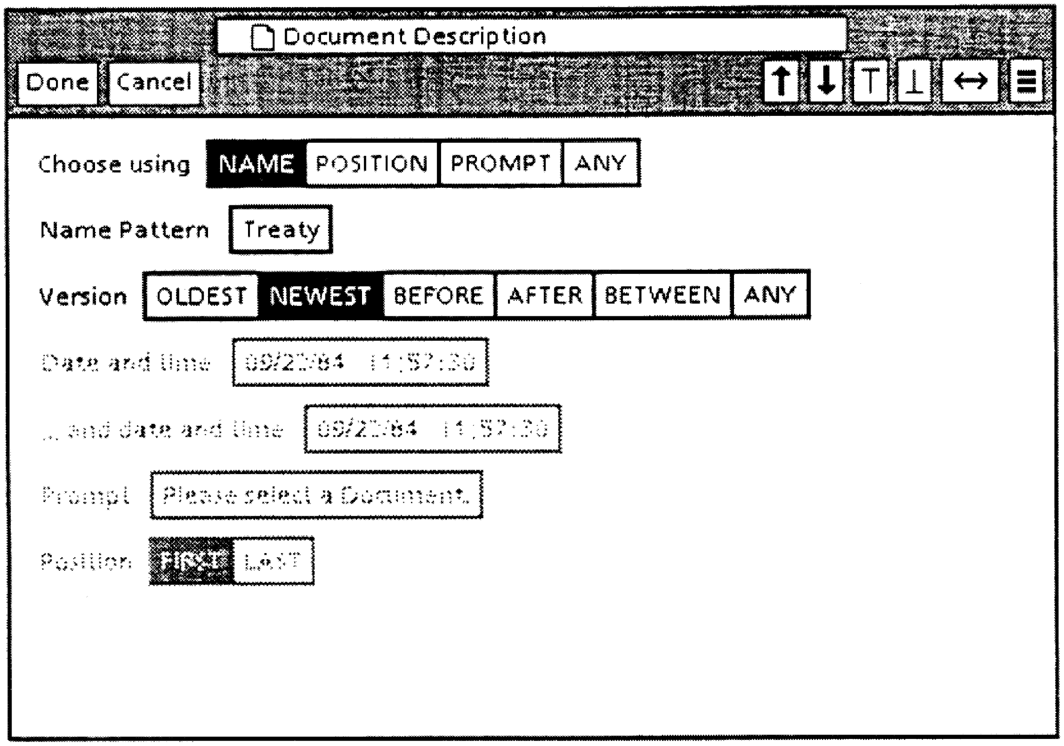


Program

Open Negotiations.
Move Negotiations Treaty to the Desktop.
Close Negotiations.

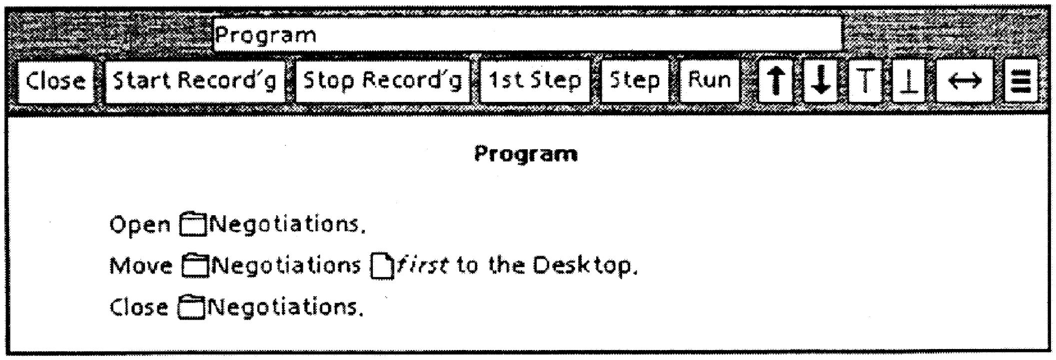
Figure 2.4 A SmallStar program listing. Reprinted from [Halbert84].

it was true that the first file in the folder was named *Treaty*, it is irrelevant to the macro definition. All arguments in the listings are represented by entities called *data descriptions*. These descriptions can be modified after an example of the macro has been recorded, to reflect the true intent of the operations as they were performed. Data descriptions are represented visually as property sheets for a piece of data. The property sheet can be used to select the relevant aspects of an argument to an operation, capturing the intent behind its selection. For example, in Figure 2.5, we see the property sheet for the *Treaty* argument, referred to in Figure 2.4. To modify this argument to be the first file in folder



Choose using [✓ NAME] POSITION PROMPT ANY
Name Pattern Treaty
Version OLDEST [✓ NEWEST] BEFORE AFTER BETWEEN ANY
Date and time 09/22/84 11:57:30
... and date and time 09/22/84 11:57:30
Prompt Please select a Document.
Position [✓ FIRST] LAST

Figure 2.5 The property sheet associated with the *Treaty* folder arguments. Reprinted from [Halbert84].



Program

Open Negotiations.
Move Negotiations *first* to the Desktop.
Close Negotiations.

Figure 2.6 Updated listing after data description change. Reprinted from [Halbert84].

Negotiations, the *Choose using* field is changed to **POSITION**, and the *Position* field must contain **FIRST**. After this data description is modified, the listing in the program window is updated accordingly, as in Figure 2.6.

In SmallStar, both set iterations and conditionals are inserted after demonstration mode, and both make use of data descriptions. SmallStar allows a sequence of commands to be wrapped in a loop that is executed for all objects that match a data description. Similarly, a sequence of commands can be wrapped in conditionals that are executed only when the piece of data matching one description has a chosen relation to the data matching another.

Halbert considers the three main contributions of his work to be the use of data descriptions to specify intent, the conclusion that some operations are better added *after* a program has been demonstrated, and the evidence that programming by example has practical use in commercial systems.

2.3.4 Peridot

Brad Myers' Peridot system [Myers86] [Myers88] [Myers93] allows the construction of virtual interaction devices by demonstration. Such devices include menus, scroll bars, buttons, and other widgets often found in user interface toolkits. Interface designers work with Peridot's graphical editor to create and position the components of the interaction devices, and demonstrate their interactive behavior. Peridot infers object-to-object relationships and a restricted class of iterations and exceptions. Peridot lists its inferences in another window, and the user must confirm them. Afterwards, Peridot produces Lisp code that implements the demonstrated interfaces.

Myers' system disambiguates graphical intent by examining objects in the scene, and *inferring* why an object was drawn the way it was. Peridot has a set of 50 object-object rules, classified by the types of objects that they act upon. This classification provides an extra set of implicit conditions for each rule to fire, and also speeds up the search. The search is ordered to compare a newly drawn object against the second most recently drawn object, and then the objects in the immediate vicinity. The rules have an ordering (with the most restrictive rules first), and as soon as a rule fires Peridot prompts for validation. Upon approval, it places one or more constraints on the new object, simultaneously beautifying the picture and insuring that the inference remains valid through further editing.

The object-to-object relationships inferred by Peridot were specifically chosen for the domain of creating interaction devices. All of Peridot's primitives have the same 4 degrees of geometric freedom (x and y of lower left corner, width and height), and can be thought of as boxes aligned directly with the x and y axes. Peridot usually infers correctly during the initial drawing stage, but has more trouble when editing a previously drawn object in a densely occupied region of the screen, so it allows the user to hint which object is relevant.

Iteration in Peridot can take one of two forms. In the simplest case, the user draws two copies of the same shape, and specifies a replication factor. This integer value can be a constant, a parameter to the program, or an active value. Peridot can also detect iterations over elements of its argument lists. When the first list element is used in the construction of a scene element, and the second list element is used in the construction of another, Peridot asks whether to iterate over the entire list.

Peridot has no visual representation of the generated procedure intended for the user interface designer. This is not a problem, because the interaction device constructions are based on sets of declarative constraints. Since these constraints have no temporal component, it is not necessary to edit the components of the procedure in an ordered fashion. Rather, constraints between objects can be edited directly at any point in time. Users can also teach Peridot the dynamic behaviors of the interaction devices by demonstration.

2.3.5 Eager

Allen Cypher's Eager system, a programming by example utility for Apple's Hypercard application, has a novel method for communicating its generalizations to the user [Cypher91]. Many programming by example systems interrupt the demonstration process to verify that their generalization hypotheses are correct. Though this is acceptable in applications that have a special teaching mode, if the programming by example facility is always turned on, as it is in Eager, this dialogue becomes intrusive and distracts from the demonstration process. When Eager detects an action sequence performed twice in a row, it pops up a green animated icon in the corner of the screen. Eager then predicts the actions that the user will perform in repeating the sequence a third time. It communicates these to the user by anticipating which buttons, menu items, and selections will be chosen next, and turning these green. Effectively, Eager conveys its generalizations *by example*, by highlighting the object instantiations and interface controls that participate in each step.

When users become confident of Eager's predictions, they can press Eager's icon, and the system synthesizes a program to automate the repetition.

The user is always free to deviate from Eager's predicted course. When this happens, Eager refines its inferred generalizations to take into account the newly demonstrated step, or abandons predicting this repetition. In detecting repetition, Eager typically matches only commands with the same operator, though there are some cases where the effect of the operations are noted instead. For example, any sequence of commands that navigates to the same card are considered equivalent by Eager. User tests indicate that this is a critical feature, since people are not always consistent in repeating a set of operations the same way. These tests also show that users are often able to learn to work with Eager with no instruction whatsoever.

2.3.6 Triggers

Richard Potter built Triggers to investigate the possibility of adding programming by example to existing applications and environments not developed to work with such a facility [Potter93a]. One difficulty that he faced was that of accessing application data—existing applications often provide no protocol for making their data available to programming by example systems. Furthermore those applications that can share their data provide varying interfaces for accessing this information, and have different data formats. Triggers relies on the pixels of the computer display to determine the state and data of running applications.

In Triggers, a program is represented by a sequence of rules. Each rule has two components: a set of conditions, and a set of actions. Triggers sequentially tests the conditions of each rule, and when the conditions are all true the actions execute. Among the conditions that can be tested is a set of pixel pattern comparisons. Triggers can examine regions of the screens for particular pixel patterns, and set markers based on the results of the search. The action sequence in each rule can simulate device actions, such as mouse and keyboard events, and set flags to be tested by the conditionals.

There are disadvantages of relying on screen state. Screen pixels are not an official protocol, and they may change with new versions of an application, device resolution, and user customizations. Pixel pattern searches may find unintended matches elsewhere on the screen. However it is hard to imagine other techniques for extracting information from an application that makes no provisions for sharing its data. Using Triggers it is possible to automate complex tasks, such as shortening horizontal lines to the point where they intersect a circle, and creating floating menus. One of Richard Potter's example tasks, creating rounded rectangles around a text string, will be visited in later chapters to illustrate two of Chimera's example-based techniques.

> *The idea of a man falling into raptures over grave and sombre California, when that man has seen New England's meadow-expanses and her maples, oaks and cathedral-windowed elms decked in summer attire, or the opaline splendors of autumn descending upon her forests, comes very near being funny.... Change is the handmaiden Nature requires to do her miracles with.*
>
> — Mark Twain, *Roughing It*

Chapter 3

Graphical Search and Replace

3.1 Introduction

In an ideal world, the first version of every document would always suit its purpose, and no edits, updates, modifications, and restructurings would subsequently be necessary. But in fact, few documents are right the first time, and even the document construction process typically involves adding new material and editing the contents in tandem. One of the great advantages of using computers to compose documents, rather than traditional pen and ink or paint and canvas, is the support that the new medium provides for making changes. It is far easier to change bits on a computer's simulated canvas or page, than to retype or paint over the real thing. One of the most promising types of changes that computers can effectively facilitate is repetitive change. Perhaps an author might want to change the name of a person or city everywhere it appears in a novel. Or a graphic designer might decide that the ducks on the wallpaper in an illustration of a room need to be a bit more yellow. These kinds of changes occur frequently along the way towards producing the final version of a document, and they are the type of task that seems to take forever when attempted by hand. Computers should be able to help here.

And they do. Textual search and replace has proven to be so useful that it is next to impossible to find a text editor on the market without this feature. This leads to the question whether or not a graphical search and replace utility might also prove to be useful, and what graphical search and replace might mean. This chapter describes an effort to define graphical search and replace, build an implementation, and investigate the classes of tasks it can facilitate.

Search and replace is a technique for making *coherent* changes to documents. Search finds a set of items sharing some of the same properties. Replace takes this set of items, and changes a subset of their properties in a consistent fashion. In traditional textual search and replace, the only property that could be matched and changed was the actual character sequence of the text. Later, as WYSIWYG combination text editors and formatters were developed, search and replace utilities were built that could match on other properties of text, such as style and font. Xerox PARC's Bravo, the first WYSIWYG text editor, did not have this capability, but Bravo's successor, Tioga, has a powerful search and replace utility called the EditTool, which can search on multiple properties [Paxton87].

However, graphical editors have traditionally had other mechanisms for making coherent changes easy. In systems that provide instancing, such as Sutherland's Sketchpad editor, all instantiations of a master or library object can be changed by manipulating the master. Individual instances have their own transformations, but typically other properties, such as color, are determined by the master. Grouping is a second means of making coherent changes in graphical editors. Multiple objects can be grouped together, and their properties changed together as though they were a single object. A disadvantage of these techniques is that they require the document to be structured according to the kinds of coherent changes that will be likely later on. If the person composing a document does not know what types of changes will be necessary, guesses incorrectly, or is just too plain lazy to put in appropriate structuring, then these techniques will not help. Graphical search and replace has the advantage that it can be used to make coherent changes without relying on the document structure at all. When structuring is present, grouping and instancing still have their uses, and as will be discussed later, graphical search and replace could be implemented to work cooperatively with these techniques.

Methods for finding graphical objects obeying certain properties have been studied by computer science researchers in other fields. Computer vision researchers have developed techniques for matching objects in complex scenes. Many of these techniques assume that the matching is being performed on an image, so they include edge detection and segmentation components. Since searching in the graphical editor domain can take advantage of a display-list (this is an advantage of dealing with "draw" programs, rather than "paint" programs!), tricky problems, like overlapping objects and rasterization to a grid, do not surface. The curve matching techniques discussed in this section were developed particularly for this task, though existing curve matching research is relevant, and could perhaps accelerate some of our searches [Pavlidis78] [Levine83]. Searching a graphical scene is also related to making queries on a graphical database, and other researchers have examined this topic [Weller76] [Palermo80] [Chang89].

Graphical search and replace has a number of additional applications, aside from making mundane coherent changes. Graphical search and replace can serve as a tool for constructing repetitive, recursive shapes defined by *graphical grammars*. It can also help to make complex scenes out of simple ones, using *graphical templates*. Graphical search can form the basis of a *graphical grep* utility, allowing graphical scene files to be retrieved by content rather than name. Also, graphical search can serve as an iteration mechanism for *graphical macros*.

In the process of studying graphical search and replace, we have built two implementations of a utility called MatchTool. The original MatchTool, hereby referred to as MatchTool 1, was implemented during the summer of 1987 at Xerox PARC's Computer Sciences Laboratory. MatchTool 1 runs on Dorado workstations [Pier83] under the Cedar Environment [Swinehart86]. It works in conjunction with the Gargoyle graphical editor [Bier86] [Pier88]. A videotape of MatchTool 1 appears in [Bier89]. In some ways, MatchTool 1's interface was patterned after Tioga's textual search and replace utility, the EditTool. It was hoped that people already familiar with the EditTool would find MatchTool 1 easy to learn. Later we built MatchTool 2 to experiment with extensions of graphical search and replace. MatchTool 2 is a companion utility to the Chimera graphical editor, is written mostly in Common Lisp, and runs on SparcStations. The capabilities described in this chapter are generally shared by both MatchTools, with a few exceptions. Here, the name MatchTool will refer to the combined feature set of both implementations.

The next section describes graphical search and replace through an example. Section 3.3 presents MatchTool's interface in greater detail. The algorithms used by MatchTool to perform graphical search and replace are described in Section 3.4. Section 3.5 presents four additional applications of graphical search and replace. Section 3.6 summarizes this chapter, presents a few observations about MatchTool's use, and describes possible future extensions to the implementation.

3.2 An Example: Yellow Jay in an Oak Tree

Chapter 1 shows an initial example of graphical search and replace being used to replace terminals with workstations in a simple network diagram. Here is a more natural example, or at least an example that is closer to nature. Figure 3.1 shows an illustration drawn with Chimera of a yellow jay sitting in an oak tree. In all likelihood, the most time consuming tasks in creating such an illustration would be drawing the jay, and copying, scaling, rotating, and positioning the many leaves. After finishing the drawing, the artist learns that yellow jays do not actually live in oaks; they live in maples. Changing the illustration by hand would take too much time, and effectively waste a large portion of the work performed earlier. If the artist had defined all of the oak leaves as being instantiations of a master library leaf, changing the master would do the trick. However, lacking the sophistication or foresight, this artist did not bother and created the many oak leaves through copying.

Fortunately, the artist's problem can be solved using graphical search and replace. The MatchTool window shown in Figure 3.2 displays the complete search and replace specification. At the top of the MatchTool window are two panes containing the search and replace objects. The top left pane is the search pane, and into it the artist copies an

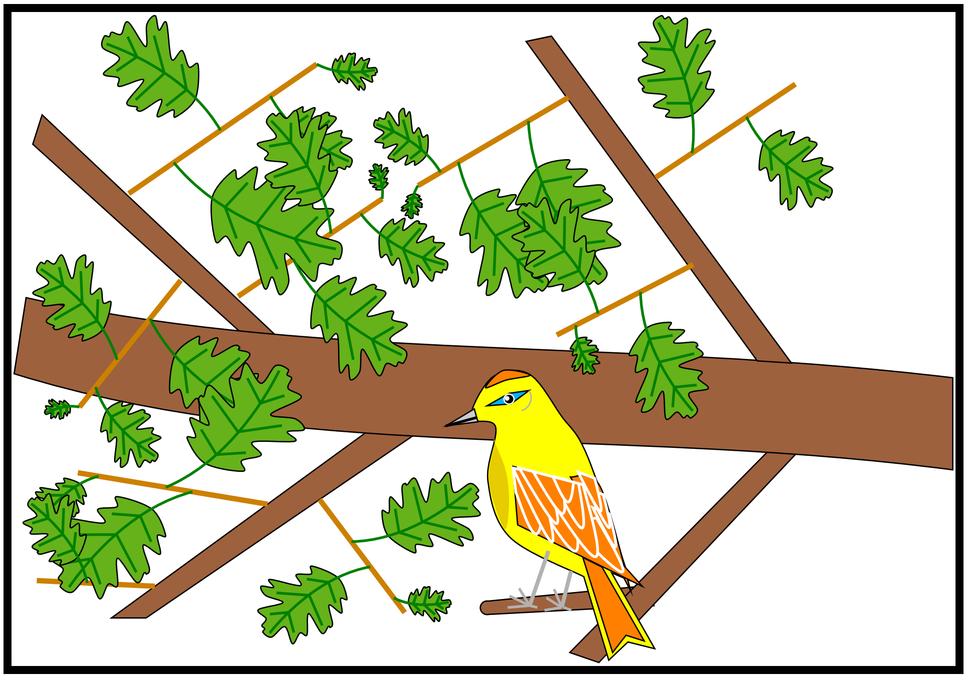


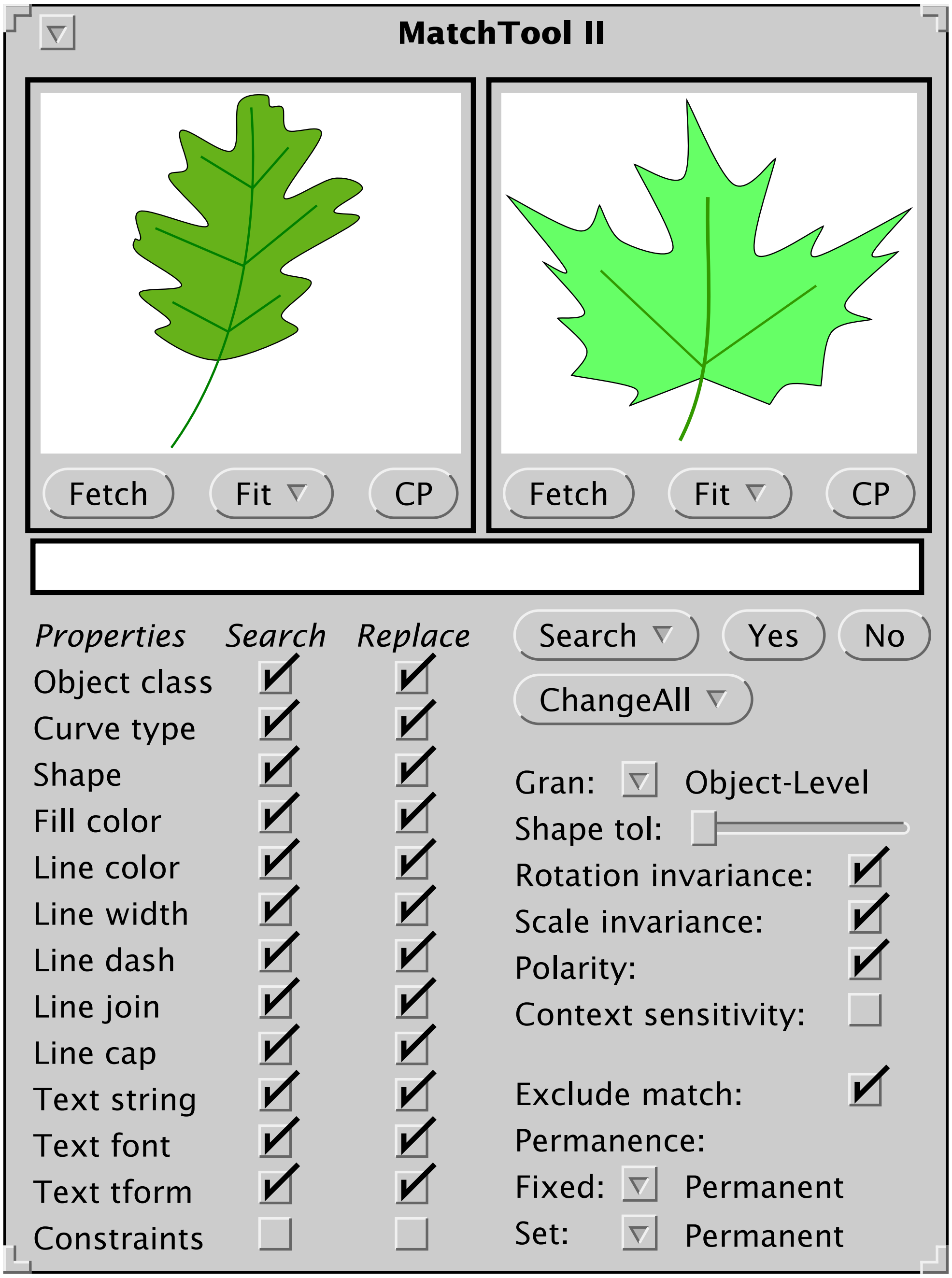
Figure 3.1 A yellow jay in an oak tree.

example of the object that will be the target of the search—in this case, an oak leaf. In the replace pane on the right, the artist draws a maple leaf as a replacement object.

Below the search pane, on the left hand portion of the viewer lies a column of properties, followed by two columns of checkboxes. These are the properties that can participate in search and replace requests. The first column of checkboxes, called the search column, specifies which properties of the objects in the search pane will be matched in the search. MatchTool does not attempt to infer these properties—they must be specified explicitly by the user.

Beside this is a second row of checkboxes, called the replace column, that specifies those properties of objects in the replace pane to be applied to a match during a replacement. The next section will describe these properties, but basically since the search column is nearly completely selected, MatchTool will be looking for objects that are nearly precise matches for the oak leaf contained in the search pane. Since the replace column is also nearly completely selected, the replacement will be taken nearly verbatim from the replace pane.

In the lower right of the MatchTool is a set of search and replace parameters that further specifies details of the search and replace process. Two of these parameters instruct the system whether or not to ignore differences in rotation and scaling between the objects in the search pane and those in the scene. Scale and rotation invariance are both turned on



Properties	*Search*	*Replace*
Object class	☑	☑
Curve type	☑	☑
Shape	☑	☑
Fill color	☑	☑
Line color	☑	☑
Line width	☑	☑
Line dash	☑	☑
Line join	☑	☑
Line cap	☑	☑
Text string	☑	☑
Text font	☑	☑
Text tform	☑	☑
Constraints	☐	☐

Gran: Object-Level
Shape tol:
Rotation invariance: ☑
Scale invariance: ☑
Polarity: ☑
Context sensitivity: ☐

Exclude match: ☑
Permanence:
Fixed: Permanent
Set: Permanent

Figure 3.2 The MatchTool viewer, containing a search and replace specification that turns oak leaves into maple leaves.

here, so the oak leaf in the search pane will match objects in the scene, at any rotation or scale. Figure 3.3 shows the scene after the transformation carried out by this search and replace specification. The yellow jay now appears in its native habitat.

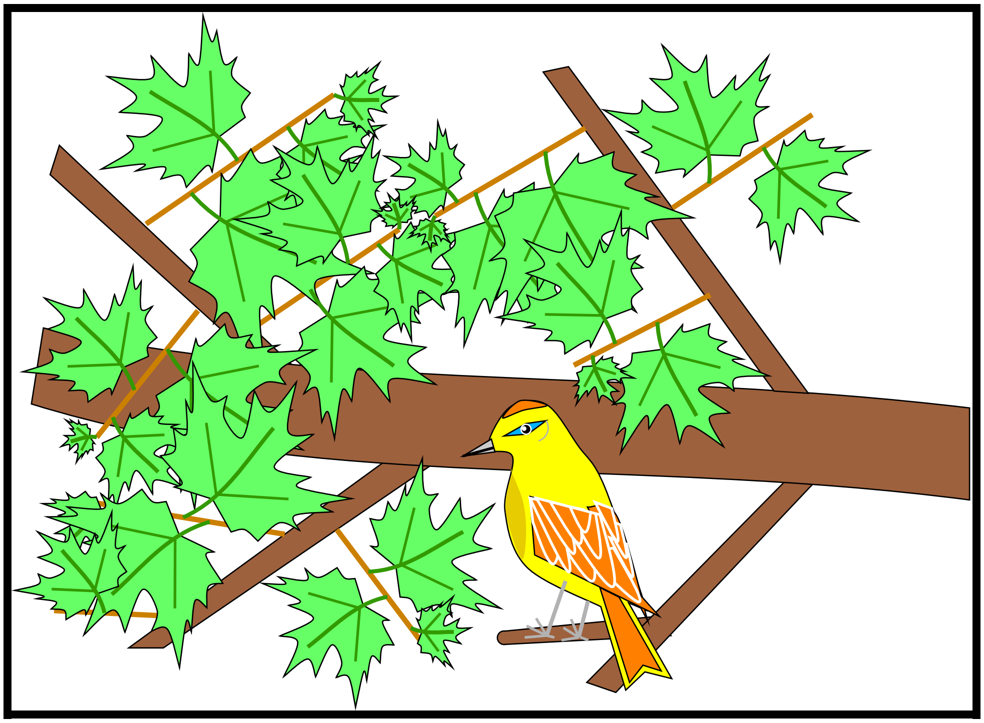


Figure 3.3 A yellow jay in its natural setting, a maple tree.

In this example, graphical search and replace proved especially useful because it facilitates coherent changes without relying on extra scene structuring. Here, the oak leaves were not instances of a master object, nor were the individual graphical primitives composing the leaves grouped together. The editor does not have a built-in oak leaf object. To the MatchTool, these oak leaves are sets of polycurves, containing straight lines, Bezier curves, beta splines, cardinal splines, and arcs. In performing this search, it matches the primitives in the search pane to those in the graphical scene.

Another important point is that the search and replace specification is graphical and example-based. Instead of having to specify the query in textual terms, the artist can simply copy an existing object, and check off the significant attributes. At no point does the end user need to become concerned with textual representations for properties like color, shape, or object class. Typically searches are applied to scenes containing at least one match, and it is an easy matter to copy an example match into the search pane and check off those properties that make it a valid match. If no sample replacement already exists that can be copied into the replace pane, the user knows how to draw one, since it requires knowledge of only the basic editor interface.

3.3 The MatchTool Interface

The oak leaf to maple leaf replacement introduced some components of the MatchTool interface. This section examines the interface in much greater detail.

3.3.1 Search and Replace Panes

Graphical search and replace is an example-based technique, because it uses examples as part of the search and replace specification. The search and replace panes hold example objects that are similar in some ways to the objects to be found and changed. Objects in the search and replace panes are called the search and replace objects. These panes are actually instances of the Chimera editor, and the complete Chimera command set can be used to draw search and replace objects in place.

Alternatively, the user can copy objects into these panes from other Chimera windows. Below each pane is a “Fetch” button that copies selected objects from the current Chimera window into that pane. Objects fetched into a pane may be much larger than the pane, or located outside of the visible portion of the pane’s coordinate system. Accordingly, the “Fit” menu contains several commands to alter the pane’s coordinate system in useful ways. For example, one command modifies a pane’s view so that all of its objects appear to fit precisely within the canvas.

3.3.2 Search and Replace Columns

The search objects need not be identical to the objects we are trying to match in all of their properties. Objects in the scene that match the selected search column properties of the search objects are considered valid matches. Similarly, only those properties of replace objects selected in the replacement column will be applied to a match during a replacement. MatchTool includes the following properties in the search and replace columns:

- **object class** - the type of an object. Some of the existing classes in Chimera include polycurve, text, box (rounded or rectangular), and circle.
- **curve type** - the kind of segment(s) composing an object. Curve types in Chimera consist of lines, arcs, Beziers, and several spline varieties. Curve types are lower level than object classes. For example, four line segments might match a rectangular box or a polycurve.
- **shape** - the form or outline of a set of objects. The shape of a circle, for example, might match the shape of a polycurve of two connected arcs.
- **fill color** - the internal color of an object.
- **line color** - the color of an object’s outline.
- **line dash** - the pattern used to draw an object’s outline
- **line join** - the way two curves are connected. Chimera allows the set of line joins defined in the PostScript imaging model: miter, round, and bevel.
- **line cap** - the style used to draw the ends of curves. Chimera allows the set of cap styles defined in the PostScript imaging model: butt, round, and square.
- **text string** - the characters composing a text string.
- **text font** - the font of a text string (e.g., Times-Bold)
- **text tform** - the transformation associated with a text string, including the scale and rotation components, but ignoring the translation.
- **constraints** - this property is explained in detail in Chapter 4.

In general, these attributes can be searched for and replaced independently or in combination. For example, Figure 3.4a. shows a replace pane, containing a crescent. In Figures 3.4b and 3.4c, the MatchTool has matched the dashed star on the left, and performs a replacement. In Figure 3.4b, we apply the shape, object class, and curve type of the crescent to the star. This effectively changes the shape and structural attributes of the star into those of a crescent, while keeping its other graphical properties, such as its fill color, line color, and line dash unchanged. Alternatively in Figure 3.4c, we change these graphi-

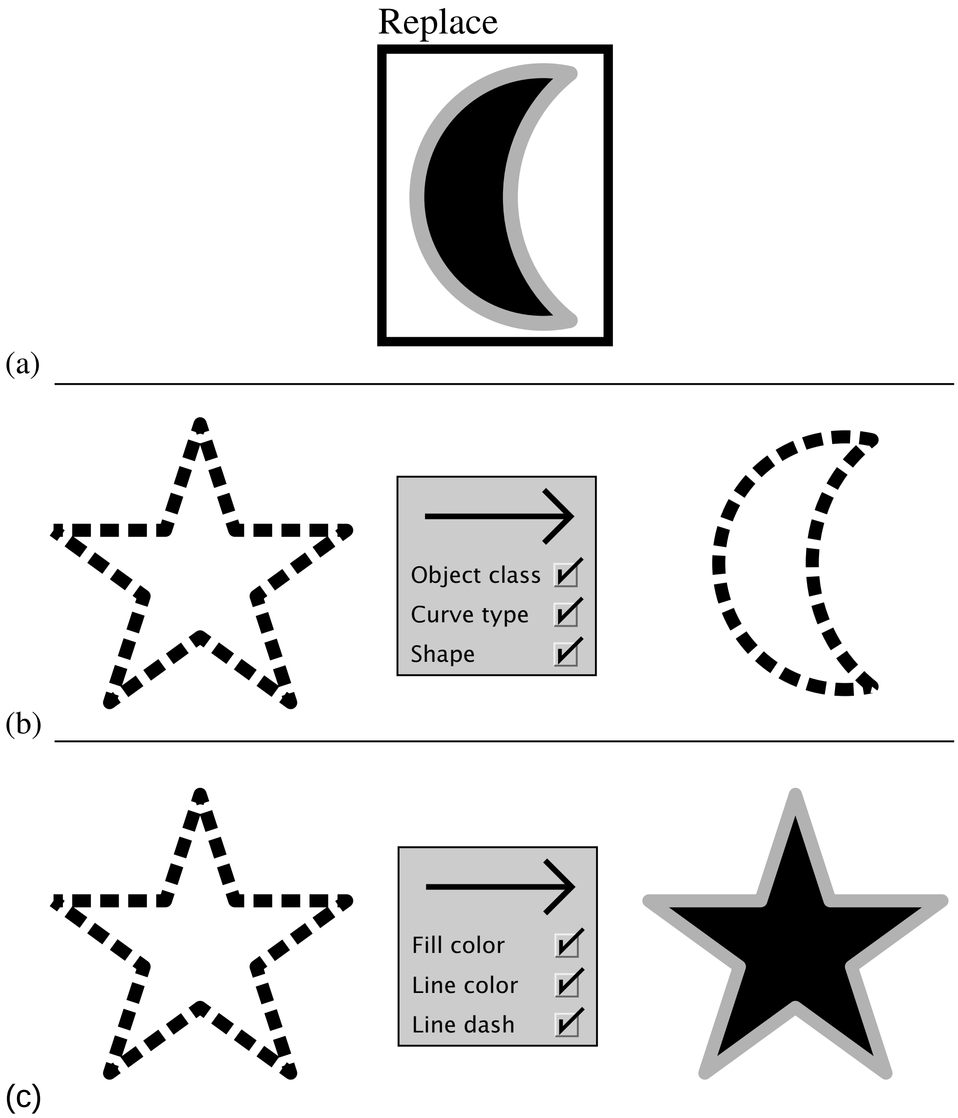


Figure 3.4 Application of different property sets. (a) the replace pane; (b) result of replacing the object class, curve type, and shape properties of the dashed star with those in the replace pane; (c) result of replacing the fill color, line color, and line dash properties of the dashed star with those in the replace pane.

cal properties to those of the crescent, leaving the star's shape and structural attributes intact.

Although MatchTool's graphical properties are generally independent, there is an exception. It is often impossible to change the shape of an object without changing its underlying representation (its object class and curve types). For example, we cannot take a circle and shape it into a box, without modifying its object class to be something other than a circle. Similarly, we can not change a circle's object class to box without changing its shape. So MatchTool has the restriction that the replace column's settings for object class, curve type, and shape must all be on or off. Toggling any one of these checkboxes automatically toggles the other two. However, these three properties can be set independently in the search column, and it is possible, for example, to search for boxes of any shape.

3.3.3 Search and Replace Parameters

Additional information must be specified to complete the search and replace specification. The search and replace parameters described next provide this. Though other search and replace parameters appear in Figure 3.2, these are relevant primarily for extensions to this technique described in the following chapter.

3.3.3.1 Granularity

As mentioned earlier, an advantage of graphical search and replace is that it does not require any extra hidden scene structure to support making coherent changes. However, sometimes the user may want to specify *existing* structure in the search. For example, in some text-based search and replace utilities, users can restrict the search to match only complete words. There the lexical structure of textual documents can serve to constrain the search. The same is true for graphical documents. In graphical editors like Chimera, there is a hierarchical structure. Multiple curves are composed together to form a higher level object (e. g., a box or polycurve). Higher level objects, can be composed together to form clusters, and these clusters can be grouped together to form still higher-level clusters.

With the *granularity* parameter, users specify how existing scene structure should affect the matching process. If this parameter is set to *anywhere*, then complete objects in the search pane can match objects at any level in the scene, though a single object in the search pane must match parts of a single object in the scene. For example, a polycurve in the search pane could match a contiguous run of curves in another polycurve segment in the editor scene. The granularity parameter can also be set to *object*. This means objects in the search pane must match objects in the scene in their entirety. MatchTool 1 had a third level, called *cluster*, which would match complete clusters in the search pane to complete clusters in the scene. This worked with a special kind of cluster in Gargoyle, containing no child clusters, but there is no reason why this feature could not be enhanced to work with clusters of arbitrary nestings.

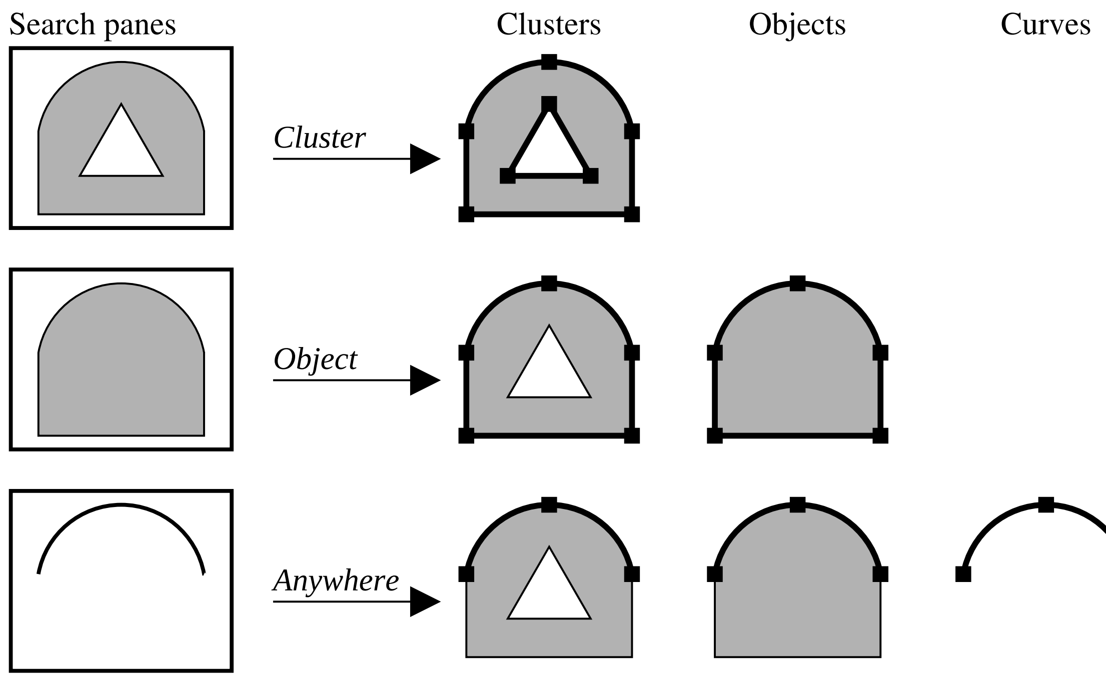


Figure 3.5 The granularity parameter. Cluster-based searches only match on complete clusters. Object-based searches find complete objects whether or not they are part of clusters. Anywhere-based searches find matches anywhere in the scene.

As an example, the top row of Figure 3.5 shows a search pane containing an arch with a triangle inside. The arch and triangle are grouped together in a cluster, and granularity is set to cluster. When MatchTool searches the scene, it finds similar objects grouped together in a single cluster. The second row of Figure 3.5 shows a search only on the outer arch. This arch is actually an object (specifically a polycurve) comprising an arc and three lines. When the granularity is set to object, this arch matches similar complete objects in the scene, whether or not they are grouped together with other objects as clusters. The search in the third row seeks a match for a single arc. Since the granularity is set to anywhere, this arc will match a curve that is part of an object, whether or not this object is part of a cluster, as well as curves that stand alone (these technically are classified as objects too, but they are the only curve in their object).

MatchTool cannot ignore all existing scene structure when performing a match. For example, even when the granularity is set to anywhere, a curve in the search panel cannot match *part* of a curve in the scene. There are several reasons for this. First, if we replace part of a curve, then it may not be possible to represent the remainder of the curve with a new curve of the original type. For example, if we splice off part of a natural spline the remainder may no longer be a natural spline. Potentially we could try to re-parameterize the curve to another type that can be represented, but this would be changing properties of scene elements not selected for replacement. Another reason why MatchTool cannot match on parts of curves is that this significantly complicates the curve matching algorithm. In practice, it is rare that we have found the need to match parts of curves, so for now the curve is the smallest atomic unit in the matching process. This means, for example, that Matchtool cannot find the center square of a standardly-drawn tic-tac-toe board by looking for a square. However MatchTool can match the shape of a circle against that of a two-arc polycurve shaped like a circle, since the polycurve and circle are both complete objects.

3.3.3.2 Shape tolerance

The user can also specify how similar shapes in the search pane must be to other shapes in the scene for a match to occur. The *shape tolerance* can be adjusted through a slider in the MatchTool window. Figure 3.6 shows an example of this parameter's effect. Here we

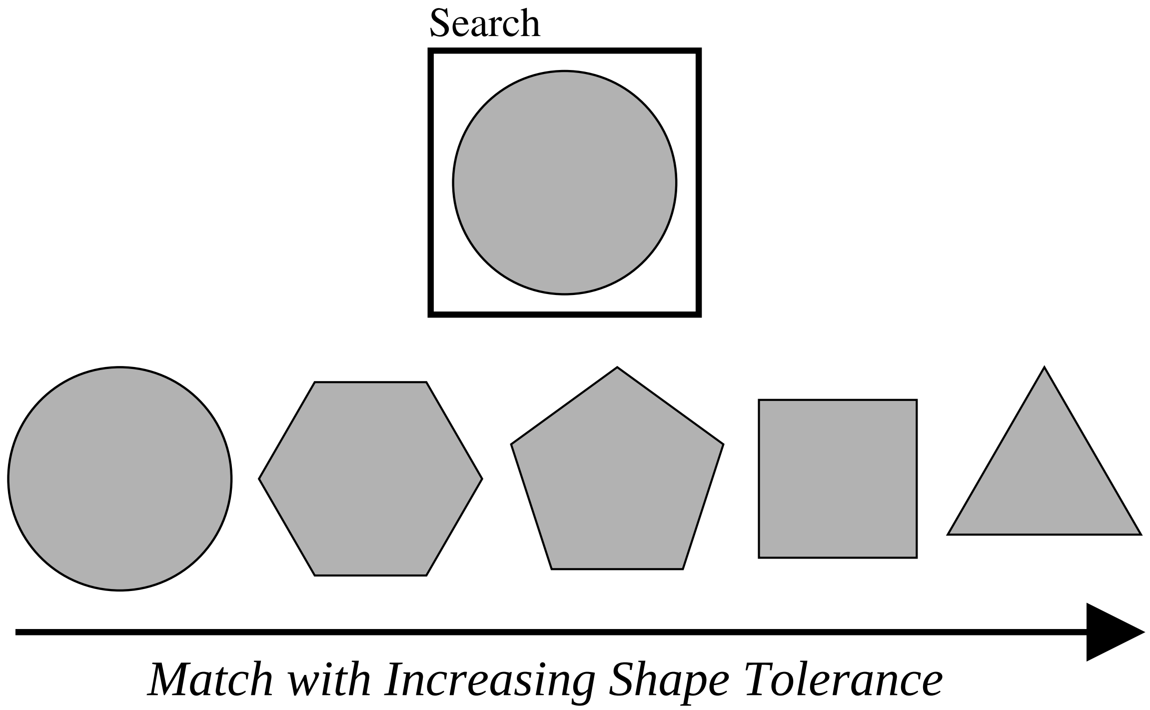


Figure 3.6 The shape tolerance parameter. Increasing the shape tolerance allows a circle to match increasingly poor approximations to it.

perform a shape match on a circle. The editor scene has several approximations to the circle, beginning with an identical circle on the left, and concluding with an equilateral triangle on the right. The shapes following the circle are regular polygons that can all be inscribed in the circle, but have decreasing numbers of segments, and therefore form poorer and poorer approximations to it. With the shape tolerance slider set to zero, only the identical circle matches. As we increase the tolerance, shapes further and further to the right match as well.

Ideally, MatchTool's shape matching metric would agree with that of its users, so shapes that appear similar to humans would appear equally similar to the program. Visual psychologists have yet to completely determine what features make two shapes appear similar, and that is outside the scope of this research. MatchTool's shape matching algorithm is described in a later section. It produces the expected results for exact matches, but for inexact matches often produces unintuitive results. While inexact matches are vital to applications like handwriting recognition, they appear to be far less useful for graphical editing. Still, we would prefer to have a more intuitive approximate match, and Section 3.6 describes some experiments towards this goal.

3.3.3.3 Rotation and scale invariance

Two additional parameters control whether differences in orientation and scaling are ignored during matching. When *rotation invariance* is turned on, the shape matching metric is rotation invariant, and objects in the search pane match identical shapes at any rotation. Similarly, if *scale invariance* is turned on, the search pattern matches identical shapes at different scales. These parameters can be controlled independently. In converting oak leaves into maple leaves, rotation and scale invariance were both turned on so that search would find objects with the same shape as the search pane objects regardless of their orientation and size. When shape is selected in both the search and replace columns, the same transformation that maps the search pane objects on to the match is applied to the replace pane objects in copying them to the scene.

3.3.3.4 Polarity

The *polarity* parameter, like granularity, relates to the use of existing scene structure in performing matches. Typically curves in graphical editors have an invisible direction associated with them. In drawing a line, for example, there is a first vertex and a second vertex. When rotation and scale invariance are turned on, a line will match another at two different orientations, 180° apart. In one case the first vertices of both will be matched together, in the second, the first vertex of one will be matched to the second vertex of the other. When polarity is turned on, two curves can only match with their directions (*polarities*) aligned.

The next figure illustrates how a curve's polarity can affect the search and replace process. Figure 3.7a shows the search and replace panes that take an equilateral triangle and turn it into a fractal snowflake. A traditional fractal snowflake, the leftmost shape of Figure 3.7b, results when the polarities are configured to make the center of each replacement point out from the interior of the snowflake. Alternatively, the replacements could always point in towards the interior, as shown in the second snowflake. (Interestingly, after the first replacement, this shape forms the logo of an automobile manufacturer. Figure out which!). The third illustration has the replacements alternating first out, then in, then out, then in. The fourth alternates in, then out, then in, then out.

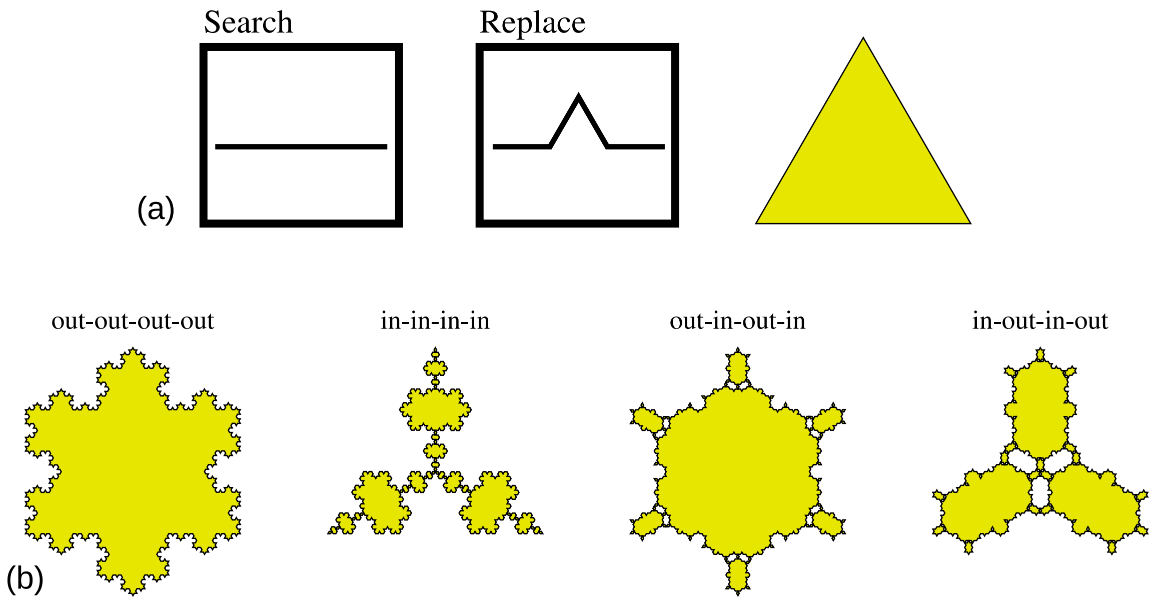


Figure 3.7 Polarity and the fractal snowflake. (a) The search pane, replace pane, and initial scene that create a fractal snowflake; (b) After four complete replacements, some of the possible results produced by varying the polarity of the match.

We have also experimented with a second interpretation of the polarity checkbox—when the checkbox is turned off, the system reverses the polarity of the objects in the search pane. Both interpretations can be useful.

3.3.3.5 Context sensitivity

When the *context sensitivity* parameter is turned on, MatchTool replaces only a subset of its match; otherwise it replaces the complete match. The user specifies which subset to replace by using the editor's standard selection mechanism to select some of the objects in the search pane. The objects that match the selected portion of a context sensitive search pattern will be altered by a subsequent replacement.

We can use context sensitive searches to avoid ambiguous replacement specifications. For example, we would like to find all triangles with circles nested inside, as shown in Figure 3.8a, and replace them with squares containing stars, as shown in Figure 3.8b. However, it is important to us that the squares get their fill colors from the triangles they replace, and that the stars get their fill colors from the circles. We might copy a triangle/circle combination into the search pane, a square/star combination into the replace pane, and try to search and replace on shape, while leaving the fill colors alone. However, this would be an *ambiguous* search and replace specification, since there would be no means to determine whether the stars and squares being added to the scene get their colors from the triangles or the circles that matched. When users attempt ambiguous replacements, MatchTool notifies them of the problem.

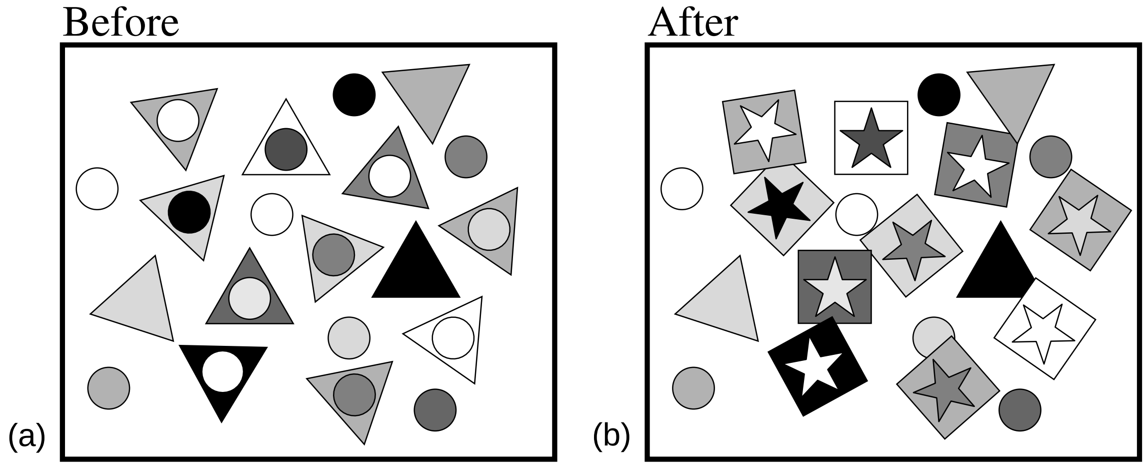


Figure 3.8 A search and replace task yielding an ambiguous specification. (a) the initial scene; (b) the desired result.

One way to avoid this ambiguity is to use two different context sensitive search and replace patterns. Both will search and replace only on object class, curve type, and shape, leaving other graphical properties, such as fill color, unchanged. The first pattern, shown in Figure 3.9a, looks for circles in the presence of triangles, and changes them into stars. Only matching circles will change, since only the circle is selected in the search pane. One reason to include the triangle in the search pane is to prevent matching circles not enclosed by triangles, of which there are several in the illustration.

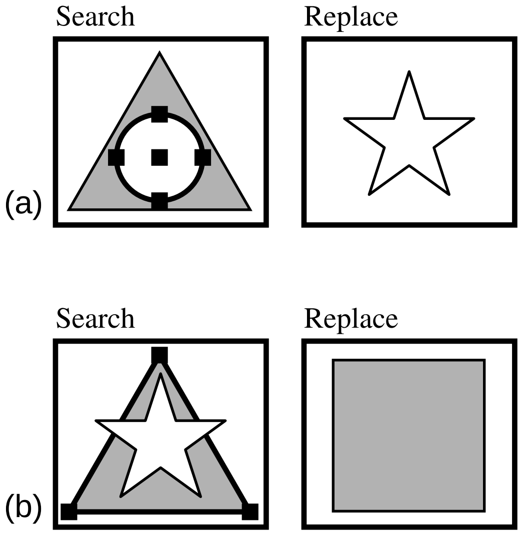


Figure 3.9 Two context sensitive search and replace patterns to achieve the transformation of Figure 3.8. (a) the first pattern; (b) the second pattern.

The second replacement, shown in Figure 3.9b, finds triangles in the presence of stars, and changes the triangles into squares. Again, the additional object in the search pane (the star in this case) prevents triangles that never contained circles from matching. The star also constrains the direction of the replacement. Two stars can match at five different orientations. Searching for the triangle and star together constrains the orientation of the square and star replacements to be the same as the triangles and circles that were formerly in the scene. Neither pattern is ambiguous, since replaced objects can inherit properties only from objects in the replace pane and scene objects that match selected items in the search pane. These patterns contain only one of each, there is no ambiguity since the replace column indicates which properties will be acquired from each.

Another method for disambiguating replacement specifications would be to have the user create a mapping between objects in the search and replace panes. The mapping could disambiguate the replacement, by indicating which objects in the replace pane get their attributes from objects matching which search pane objects. Chapter 4 describes such a mapping mechanism implemented in MatchTool for a slightly different purpose, but it could easily be adapted for disambiguating the source of replacement properties.

3.3.4 Command Invocation

Forward searches in MatchTool move from the position of the software cursor (the caret) generally towards the bottom of the screen. Objects at the same height are retrieved from left to right. This convention conforms to the standard search direction of text editors. Reverse searches work in the opposite direction. MatchTool has a set of buttons and menus to control the invocation of graphical search and replace. The menu button labeled “Search” initiates searches for the next match. Pressing it with the left mouse button brings up a menu that allows forward and backward searches, as well as searches from the top of the scene. Pressing it with the right button invokes a forward search, the default. MatchTool highlights matched objects using the editor’s selection mechanism. It also moves the caret to the position of the match.

After a match has been found, the user can press the “Yes” button to perform a replace, and the “No” button to leave a match as is. Both of these buttons invoke another search in the same direction for the next match. To perform a replace throughout the scene, rather than stepping through the matches, one by one, the user can press the “ChangeAll” menu button. Like the “Search” button, it brings up a menu to control the direction of the matching process, and it defaults to a forward search.

3.4 Algorithm

This section explains how MatchTool performs graphical search and replace. It begins by describing the way MatchTool compares the shape of one contiguous run of curves against another, and then discusses how it matches collections of these runs. Finally, it outlines how graphical replace works.

3.4.1 Matching Runs of Curves

A run of curves is a piecewise continuous collection of two dimensional curves that are represented together internally. Some MatchTool object types, such as polycurves and boxes, contain these runs. Two separate polycurves that connect at an endpoint do not form a single run of curves, since these curves are not represented internally as contiguous.

The first steps in matching one run's shape against another factor out those differences that we wish to ignore. The initial step converts both runs to a polyline representation, to effectively isolate their shape from their curve parameterizations. Figure 3.10 shows parameterizations of two different runs with the same shape. The run on the left of Figure 3.10a consists of two arcs, a straight line, and two Beziers, while the run in Figure 3.10b consists of a single arc, two lines, and a cardinal spline. The polyline approximations of both are

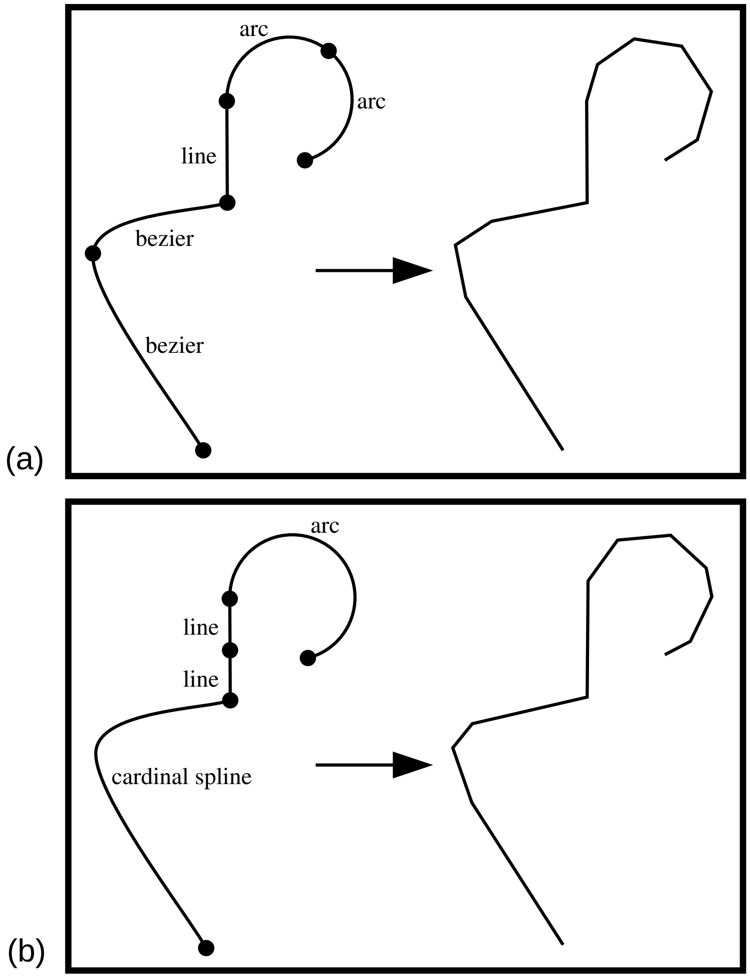


Figure 3.10 Converting curve runs to polylines. Two runs of curves with different mathematical representations (in boxes a and b) yield similar, though not necessarily identical polylines.

similar since their shapes are the same, though the precise approximations can differ. These polyline approximations are adaptive—more line segments represent regions of high curvature than low curvature. This illustration exaggerates the roughness of MatchTool's polyline approximations to make it clear that they are not smooth curves.

Next MatchTool puts the polylines in a canonical form that factors out differences caused by the classes of transformations we wish to ignore. This process contains multiple steps. First, the system finds a starting point for each polyline, and the polyline's center of mass if it were constructed from wire of uniform density. The starting point is simply the first vertex if the polyline is open, or the vertex furthest from the center of mass if the polyline is closed. Some closed polylines will have multiple starting points, and they have multiple canonical forms. Figure 3.11a shows a polyline, its starting point and center of mass.

Since all shape matches ignore differences in translation, the canonical form of two polylines should ignore this as well. Hence the system next translates the polylines so that the first vertex lies at the origin. This is shown in Figure 3.11b. When the MatchTool performs a rotation invariant search, then differences due to rotation should not be consid-

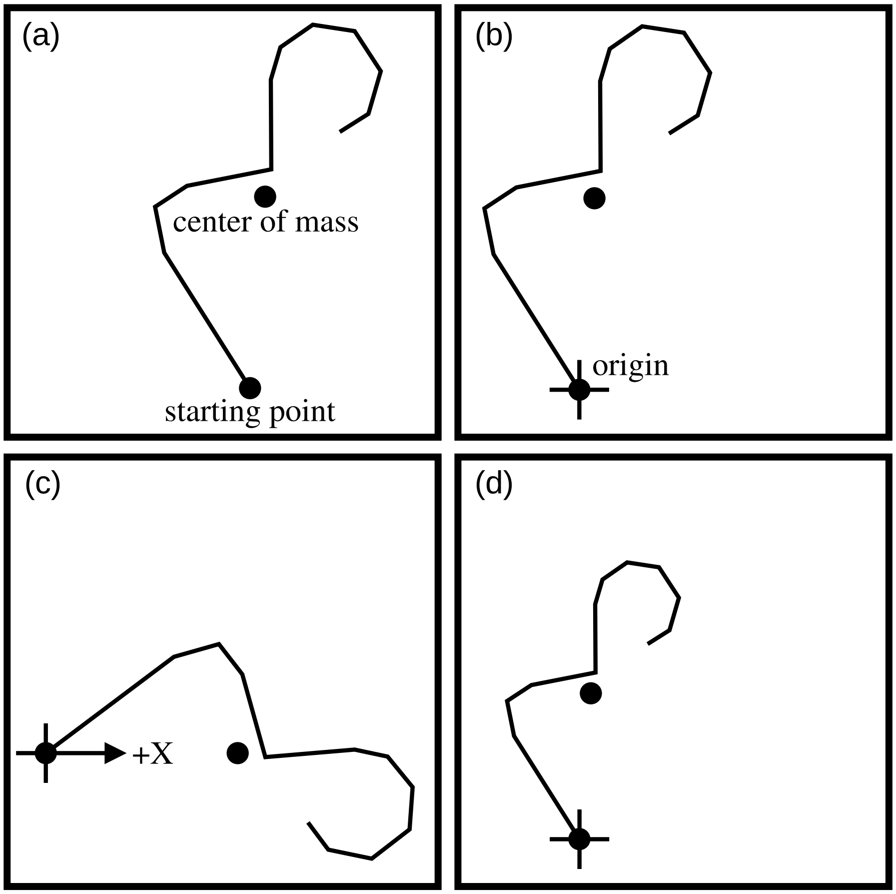


Figure 3.11 Canonical forms. (a) important points; (b) translation invariant form; (c) rotation invariant form; (d) scale invariant form.

ered either, so it rotates the polylines such that their center of mass lies along the positive *X* axis. Figure 3.11c illustrates a polyline that has had its rotation factored out by this step. If the search is scale invariant, differences due to size are factored out by scaling the polylines to a constant arc length. Figure 3.11d represents this final step. Note that the canonical form will not be unique if the starting point and the center of mass coincide. In this case, we can have the system use a substitute for the center of mass, such as the point furthest from the starting point, or the point halfway along the curve from the starting point.

Next, the MatchTool performs a set of quick-reject tests to quickly determine whether two polylines definitely do not match. The arc lengths of the two polylines are compared. If they differ by more than a certain threshold (the current shape tolerance plus a little extra to take into account possible differences in the polyline approximations and floating point fuzz), then MatchTool immediately rejects the match. Figure 3.12a shows this quick reject test. The second test compares the maximum distances between the polylines and their centers of mass. If they differ by more than this threshold, then the shapes also do not match. Figure 3.12b illustrates this test. The final quick-reject test compares the distance between the two centers of mass after the polylines have been canonicalized. As shown in Figure 3.12c, if this distance is greater than the current threshold, then MatchTool also rejects the match outright.

If the polylines pass the quick reject tests, then there is still a chance that they might have the same shape, and the MatchTool performs a more comprehensive shape comparison. For each vertex of each polyline in canonical form, MatchTool considers the distance to the point at the same proportional position along the other polyline. Since this point

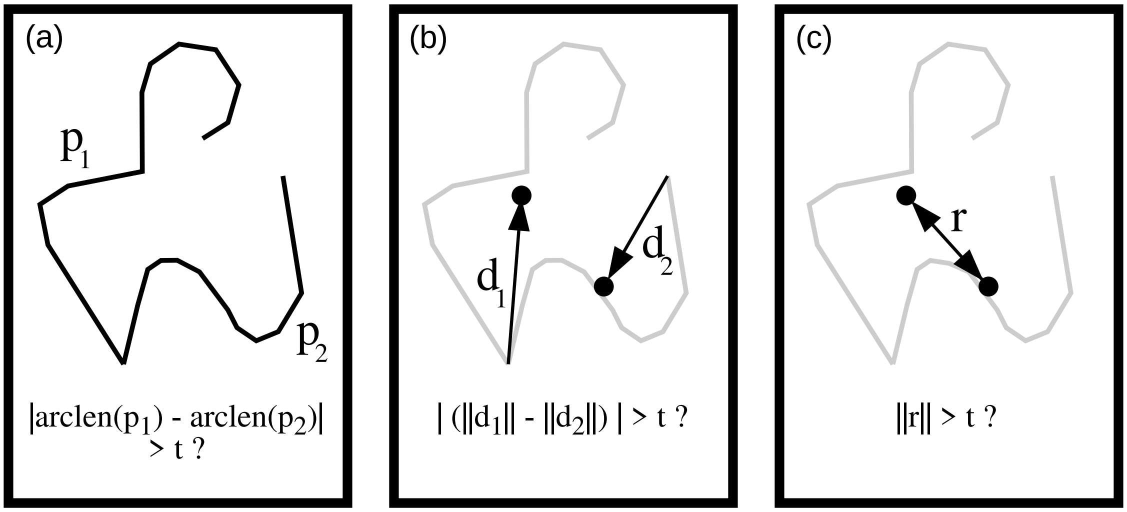


Figure 3.12 Quick reject tests. (a) comparison of the arc lengths; (b) comparison of maximum distances to centers of mass; (c) distance between centers of mass. In these figures, t is the threshold.

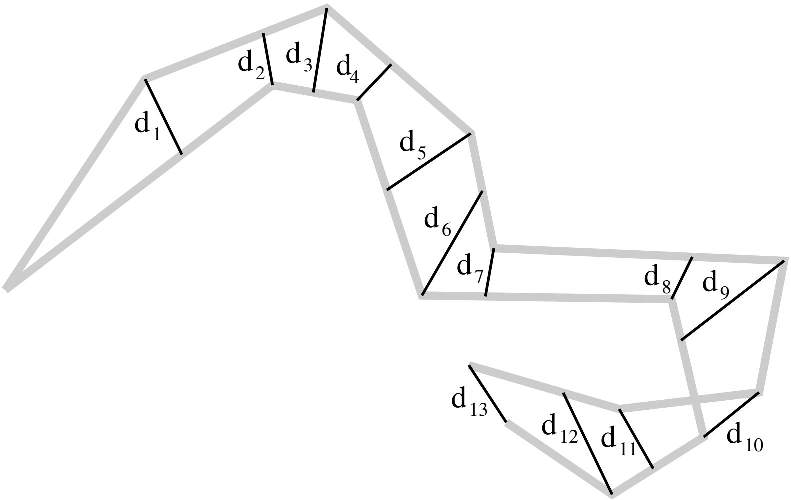


Figure 3.13 Full polyline test. If $||d_i|| \le t$, for all i, then the curves match.

typically does not coincide with a vertex, it is usually determined through linear interpolation of one of the polyline segments. Figure 3.13 shows the distances tested in this example. If any of these distances exceed the threshold, then the match is rejected, otherwise it is accepted. To speed up the calculation, MatchTool actually uses the Manhattan distance (L-1 norm) rather than the Euclidean distance, and the tolerance is scaled up by accordingly.

As mentioned earlier, some closed polylines do not have a unique canonical form. Also, if matching ignores polarity, then open polylines will have two canonical forms with a starting point at each end. When comparing two polylines with multiple canonical forms, it is important to try matching every canonical form of one of the polylines to a single canonical form of the other. Being independent of the chosen starting point, the quick reject tests of Figure 3.12a and b can be applied to a single canonical form of each polyline. However, the quick reject test of Figure 3.12c must be applied to every canonical form of one polyline matched against a single canonical form of another.

The technique just described for matching runs of curves is flexible in its ability to compare different curve types. By converting all curves into a secondary representation (polylines), and comparing these representations, the technique avoids having order n^2 different methods for comparing curves. However, there is some overhead in converting runs of curves into their polyline approximations. If the user is looking for runs of curves that match structurally, that is, have the same curve types and control point placements, then complete *polyline-based curve matching* is overkill.

MatchTool can also perform *structural curve matching*. In this technique, the system collects the end points and control points of a curve run into a polyline, and then matches one polyline against another using the polyline-based method. The polylines are annotated with curve-type information that also must match. This method typically runs faster, since the polylines tend to have far fewer segments. Structural curve matching's disadvantage lies in its inability to compare shapes with different representations. The two curve runs of Figure 3.10 cannot be compared since they have different structures, and hence they automatically fail to match. Even in comparing two Bezier curves, this technique can fail, since Beziers of the same shape might have different control point positions. However, usually in matching shape, people care only about structural similarity, so structural curve matching provides a useful, efficient alternative.

3.4.2 Matching Sets of Objects

Now that it has been explained how MatchTool compares the shapes of individual runs of curves, this section describes the technique that MatchTool employs to find matches for multiple objects in the search pane.

The initial step forms a list of scene elements to be searched. This list is called the *scene list*. If granularity is set to *object* or *anywhere* this list contains objects, otherwise it contains *clusters*. The scene list forms a snapshot of the scene at the beginning of the search. It contains supplementary information indicating which parts of scene elements still can be matched. By working from this initial snapshot, MatchTool avoids getting caught in endless cycles, replacing parts of replacements. MatchTool takes an updated snapshot whenever beginning a new search, but not when continuing an old one. The user indicates that the next search will be new by moving the caret in the scene being searched, editing this scene, changing the search pattern, modifying the search direction, or searching in an entirely new scene.

In the case of a forward search, the search list contains all scene elements located below the caret, sorted decreasingly by the y-coordinate of the upper left corners of their bounding boxes. If two bounding boxes have the same y-coordinate, they are sorted according to increasing x-coordinate. This allows forward graphical searches to proceed in roughly the same direction as forward textual searches. The *roughly* will be explained later in this section.

The search order brings up an interesting dilemma. Ideally reverse graphical searches would have the following two characteristics. First, they would be ordered according to the bottom right corner of each object in the scene, since the reverse scan direction would be bottom to top, right to left. Second, a reverse search should match objects in the exact opposite order as a forward search. This second characteristic allows the user to reverse the search at any time, and examine the previous match again. Unfortunately these characteristics conflict in graphical search. Imagine a small circle centered in a larger one. Searching from the top or bottom of the scene will find the larger circle first. Yet, the reverse search must find the smaller circle first to match objects in the opposite order as the forward search. In MatchTool the second characteristic was given priority. Scene lists for reverse searches include all objects above the caret, sorted by increasing y and decreasing x, still according to the coordinate of the upper left corner of their bounding boxes.

Next MatchTool forms a second list, the *search list*, containing all the objects in the search pane. The order of the objects in this list is not critical, though this section later describes how to choose an order to accelerate the search. The first object in the search list is called the *leading object*. MatchTool sequentially compares the leading object to all members of the scene list to find a match. After finding a match for the leading object, MatchTool needs to find matches for all of the other members of the search list. Fortunately, if we are performing shape matches, the leading match provides information on where to look for the other objects in the scene. The same transformation that maps the leading object onto its match must map the other search list objects as well.

For example, consider Figure 3.14. Here the leading object is the highlighted Bezier curve at the bottom of the search pane (Figure 3.14b). If the MatchTool seeks shape matches for the search pattern in the scene (Figure 3.14a), then finding a preliminary match for the leading object also provides locations for potential matches of the other search list objects. For example, if the Bezier at the bottom of Figure 3.14b matches the Bezier at the lower left of Figure 3.14a, then the other matches must be situated as outlined by the grey dashed lines.

After finding a match for the leading object, the technique searches for matches of the remaining objects, but only in that region of the scene where they might be located. For each remaining object on the scene list, the system transforms several of its points by the transformation that maps the leading object to its match. If a scene object's bounding box,

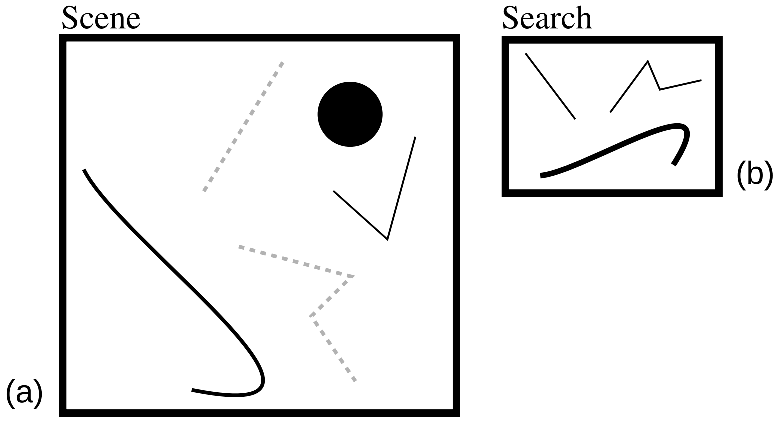


Figure 3.14 Using the leading object match to help find the other matches. (a) a graphical scene; (b) a search pattern with the leading object highlighted. If the leading object matches the lower left object in the scene, this determines the location of the other potential matches (drawn grey and dashed).

after being increased in width and height by twice the tolerance threshold, does not contain all of these points, then the scene object is immediately rejected as a match for this search list object.

If MatchTool cannot find corresponding scene elements for the remaining members of the search list, it tries other matching orientations of the leading object against its match. Before rejecting a leading object match, MatchTool must fail to match all elements of the search list, for every matching orientation of the leading object against its match. This makes it advantageous to choose a leading object with as few canonical forms, and hence potential matching orientations, as possible.

For example, Figure 3.15 contains a search pattern for a graphical representation of 1-methylcyclohexene. If the hexagon is chosen to lead the search, as in Figure 3.15a, then it can match hexagons in the scene at six different orientations. Matches may need to be sought for the remaining search list objects at each of these orientations. Figure 3.15b shows that two different matching orientations are possible if we select the double-bond line as leading object. The text string makes the best leading object, since this text string matches itself at only a single orientation. This is demonstrated by Figure 3.15c. In

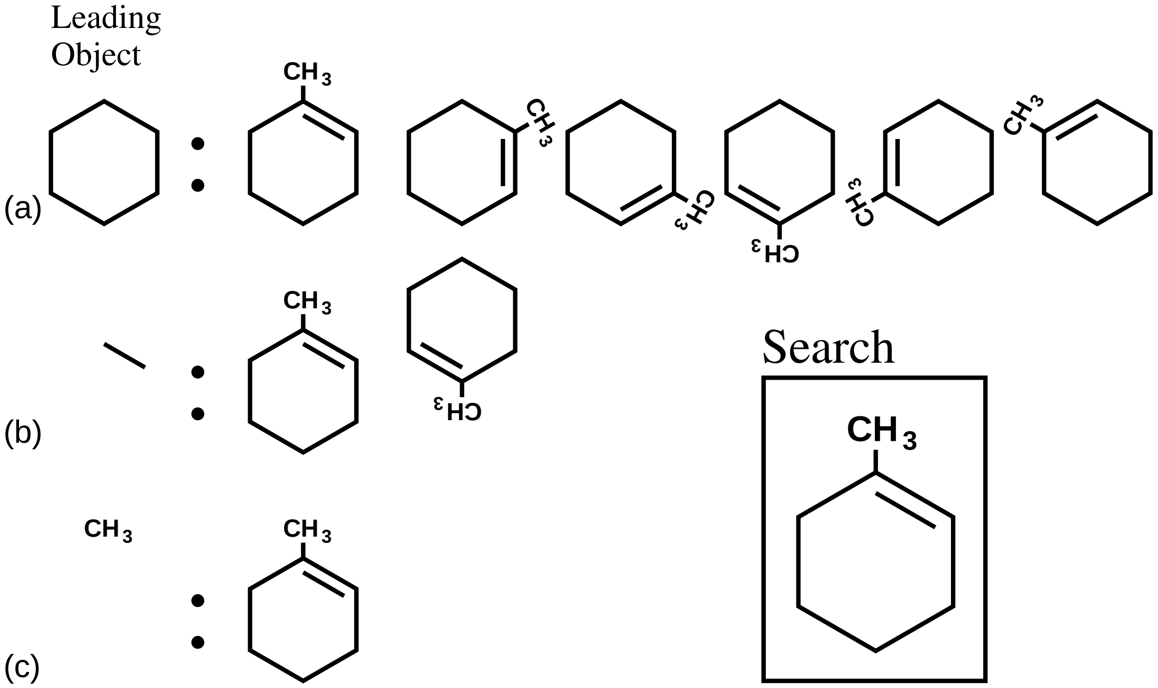


Figure 3.15 Choosing a leading object. Three possible choices for the leading object are shown; (a) the hexagon might match at six different orientations; (b) the line might match at two orientations; (c) the text string can match at only a single orientation.

general, symmetric objects such as circles and hexagons make poor leading objects, since multiple orientations may need to be tested. Common objects such as lines also make poor leading objects, since they increase the likelihood of leading object matches. For example, if the double-bond line acts as a leading object in a rotation and scale invariant search, it might also match against the line representing the CH_3 bond, as well as other lines in the scene, creating extra work. MatchTool chooses the object in the search list with the least number of canonical forms as leading object. When multiple objects fit this description, MatchTool chooses the object composed of the most segments, working on the assumption that simple shapes (such as single lines) tend to be more common in graphical scenes.

When it becomes clear that the search list does not match scene objects for any orientation of the leading object, we test the leading object against the next element of the search list or its components. The search process, in fact, finds matches ordered according to the upper left hand corner of the leading object's match. In the case of rotation invariant searches, this may not lead to matches in strictly decreasing y order, though these searches also tend to move down the screen. When searching *within* an object, MatchTool searches for matches using the object's intrinsic component ordering rather than height.

When MatchTool finds matches for the complete scene list, it visually presents the objects it found by selecting them and scrolling the editor canvas to make them visible if they lie off-screen. MatchTool moves the caret to the upper left hand corner of the match's bounding box to reflect the change in the current position of the search. It also updates the snapshot to disqualify elements of the match from participating in continuations of the same search. When searching at the *anywhere* granularity level, the snapshot keeps track of which parts of objects have already matched, as well as the next object part to be compared against the leading object. This allows searches to be continued from within the middle of objects. Also, when the user reverses a search, the new search begins from the current position in the current object.

3.4.3 Graphical Replace

The MatchTool performs replacements in one of two ways, depending on which properties the user selects in the replace column. If the user does not replace the object class, curve type, and shape properties, then the process is simple—the system extracts those properties selected in the replace column from the replace pane objects, and applies them to the matched objects. However, if the user chooses to replace object class, curve type, and shape, the process is slightly more complicated since it is harder to change these properties in existing objects.

In this case, MatchTool first copies the objects in the replace pane, and applies to the copy the properties of the match *not selected* in the replace column. The system deletes the match from the scene, then transforms the copy according to the transformation that maps the search panel objects onto their match. It then adds the transformed copy into the scene. If part of a polycurve is being replaced and a component of the replacement fits in the gap, then this replacement will be spliced into the polycurve. This feature was useful in creating the fractal snowflake examples of Figure 3.7. Since the initial scene consisted of a single polycurve, the result is also a single polycurve. This has the advantage that the resulting shape remains a closed, filled curve, and can be selected as a single unit.

During the replace process, MatchTool identifies ambiguous replace specifications and notifies the user. A replace pattern will be ambiguous if object class, curve type, and shape are not selected in the replace column, yet the replace pane contains multiple values for a property that is selected. For example, if the user tries to replace the colors of the Italian flag with the colors of the Union Jack without modifying the shape, then it will not be clear which parts of the Italian flag should receive which color. Alternatively if the user selects object class, curve type, and shape in the replace column, a replacement will be ambiguous if the match contains multiple values for an unselected replacement property. An example of this would be trying to replace the shape of an Italian flag with the shape of a Union Jack, while keeping the Italian flag's original colors. It would be unclear to MatchTool which components of the Union Jack should receive which colors of the Italian flag.

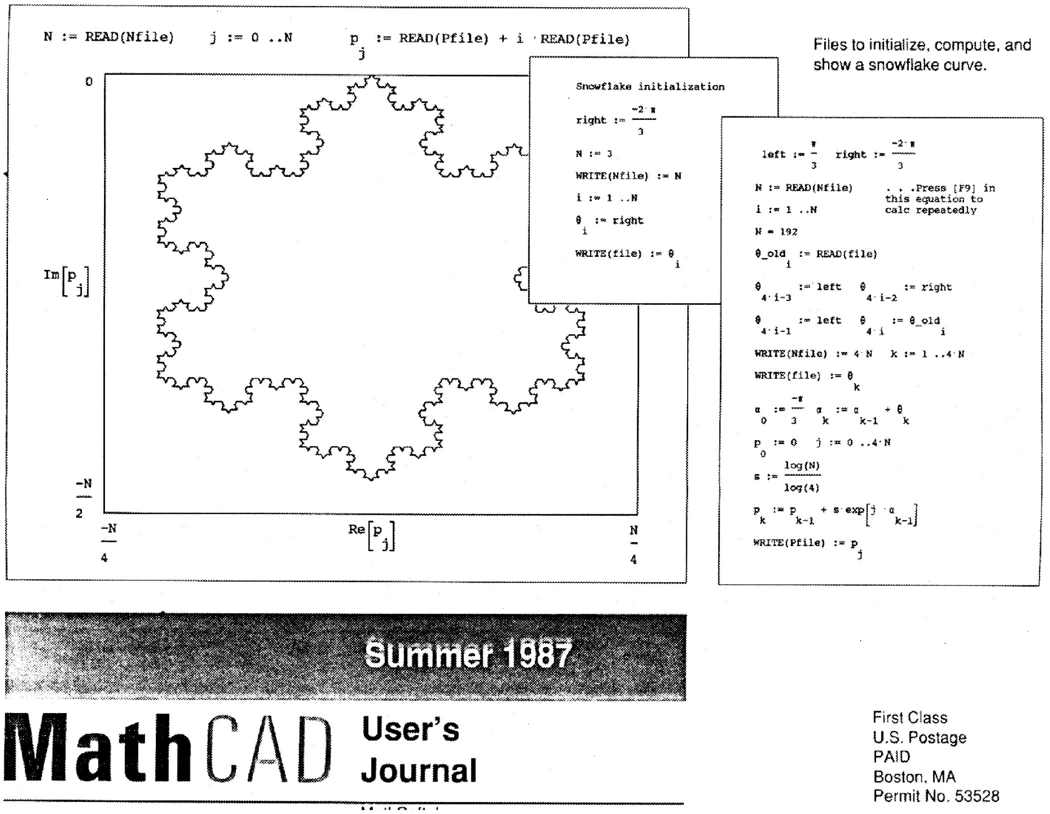
3.5 Other Applications

Aside from making commonplace changes to graphical documents, graphical search and replace facilitates several different graphical editing applications. This section describes four of these uses.

3.5.1 Graphical Grammars

A graphical grammar is a set of production rules that match and replace sets of graphical properties. Typically these graphical properties include shape. Shape grammars, a class of graphical grammars in which shapes form all the terminals and non-terminals, have been explored by others [Gips75] [Stiny75]. Graphical search and replace provides a convenient mechanism for generating pictures formed by graphical grammars.

Examples of such illustrations are the snowflake curves presented in Figure 3.7. In his dissertation, Asente cites the traditional snowflake curve as one reason why programming subsystems are important for graphical editors—that shapes like these are inherently difficult to construct through direct manipulation, since they require so much repetition and precision [Asente87]. Using MatchTool, people can construct such shapes *without* traditional programming. Also, it is far easier to construct these shapes using graphical search and replace than general programming, since search and replace very closely approximates the abstractions used to define graphical grammars. As an example of how "easy" it is to define snowflake curves through programming, Figure 3.16 shows a page from the MathCAD Journal, describing how to draw a snowflake in their programming language.



$N := \text{READ}(\text{Nfile}) \quad j := 0 \,..\, N \quad p_j := \text{READ}(\text{Pfile}) + i \cdot \text{READ}(\text{Pfile})$

Snowflake initialization

$\text{right} := \frac{-2 \cdot \pi}{3}$

$N := 3$

$\text{WRITE}(\text{Nfile}) := N$

$i := 1 \,..\, N$

$\theta_i := \text{right}$

$\text{WRITE}(\text{file}) := \theta_i$

Files to initialize, compute, and show a snowflake curve.

$\text{left} := \frac{\pi}{3} \quad \text{right} := \frac{-2 \cdot \pi}{3}$

$N := \text{READ}(\text{Nfile})$ . . .Press [F9] in this equation to calc repeatedly

$i := 1 \,..\, N$

$N = 192$

$\theta\_\text{old}_i := \text{READ}(\text{file})$

$\theta_{4 \cdot i-3} := \text{left} \quad \theta_{4 \cdot i-2} := \text{right}$

$\theta_{4 \cdot i-1} := \text{left} \quad \theta_{4 \cdot i} := \theta\_\text{old}_i$

$\text{WRITE}(\text{Nfile}) := 4 \cdot N \quad k := 1 \,..\, 4 \cdot N$

$\text{WRITE}(\text{file}) := \theta_k$

$\alpha_0 := \frac{-\pi}{3} \quad \alpha_k := \alpha_{k-1} + \theta_k$

$p_0 := 0 \quad j := 0 \,..\, 4 \cdot N$

$s := \frac{\log(N)}{\log(4)}$

$p_k := p_{k-1} + s \cdot \exp\left[j \cdot \alpha_{k-1}\right]$

$\text{WRITE}(\text{Pfile}) := p_j$

Summer 1987

MathCAD User's Journal

First Class
U.S. Postage
PAID
Boston, MA
Permit No. 53528

Figure 3.16 A page from the MathCAD Journal, listing a program to construct a snowflake curve.

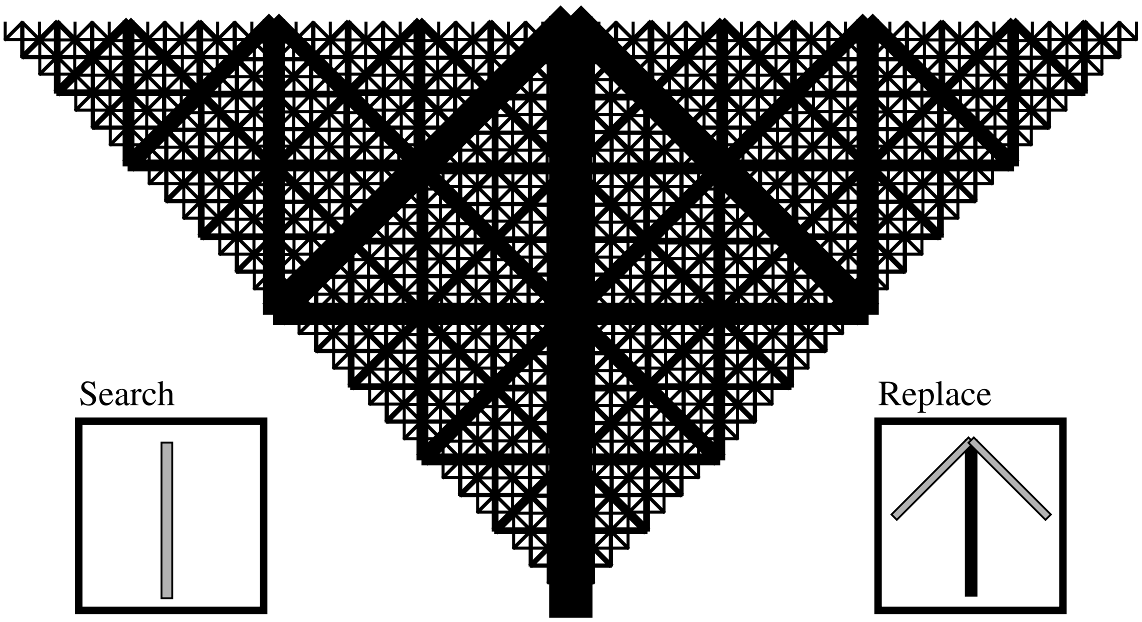


Figure 3.17 A fractal arrow.

Another example of a graphical grammar is given in Figure 3.17. Here, we construct a fractal arrow by starting with a grey box, and replacing each grey box with a black box and two grey boxes, as shown in the search and replace panes. The two additional grey rectangles in the replace pane are scaled down versions of the original, so with each generation of replacements the new rectangles get smaller. Here we search on color, in addition to shape, to avoid matching on rectangles that are not leaves of this binary tree. The fractal arrow in the figure results after repeating 10 generations of replacements, and then making the remaining grey rectangles black. The *PostScript Language Tutorial and Cookbook* describes how to create this shape through recursive PostScript programming [Adobe85]. The program is relatively simple, containing 21 lines of code, and is intended for instruc-

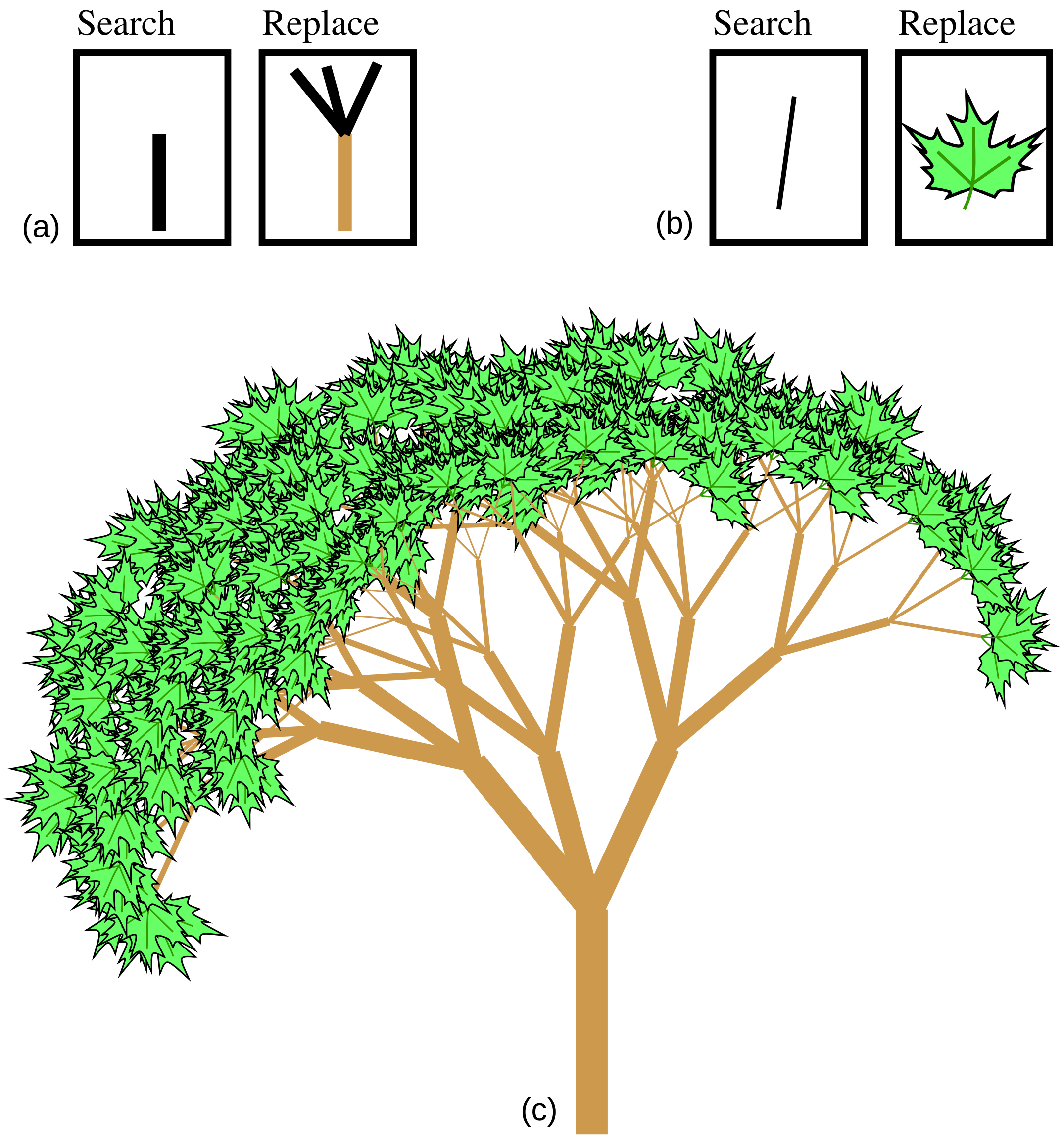


Figure 3.18 A graphical grammar-built tree. (a) the production that built the branches; (b) the production that added the leaves; (c) the maple tree.

tional purposes. However, the graphical search and replace specification is even simpler, and can be constructed by non-programmers using their graphical editing skills.

A second kind of tree appears in Figure 3.18. In this example, we form a simple maple tree by two replacement specifications. Initially the scene contains a single black vertical line. As shown in Figure 3.18a, we replace the black line with a brown line and three black lines. (Since this is a grey scale rendering, brown lines appear grey). We repeat this replacement for several generations. Next, we search for the black lines, which form the "leaves" of the tree, and replace them with maple leaves, as represented by the search and replace pattern of Figure 3.18b. The result appears in Figure 3.18c. Though graphical grammars were not used to build the maple tree of Figure 3.3, they could have been used for this purpose. Smith studied the use of grammars (or "graftals") to generate plants [Smith84], and Bloomenthal developed an interesting approach to modeling 3D maple trees without grammars [Bloomenthal85]. The next section will describe another application of graphical search and replace that did in fact contribute towards building the tree of Figure 3.3 with relatively few interaction steps.

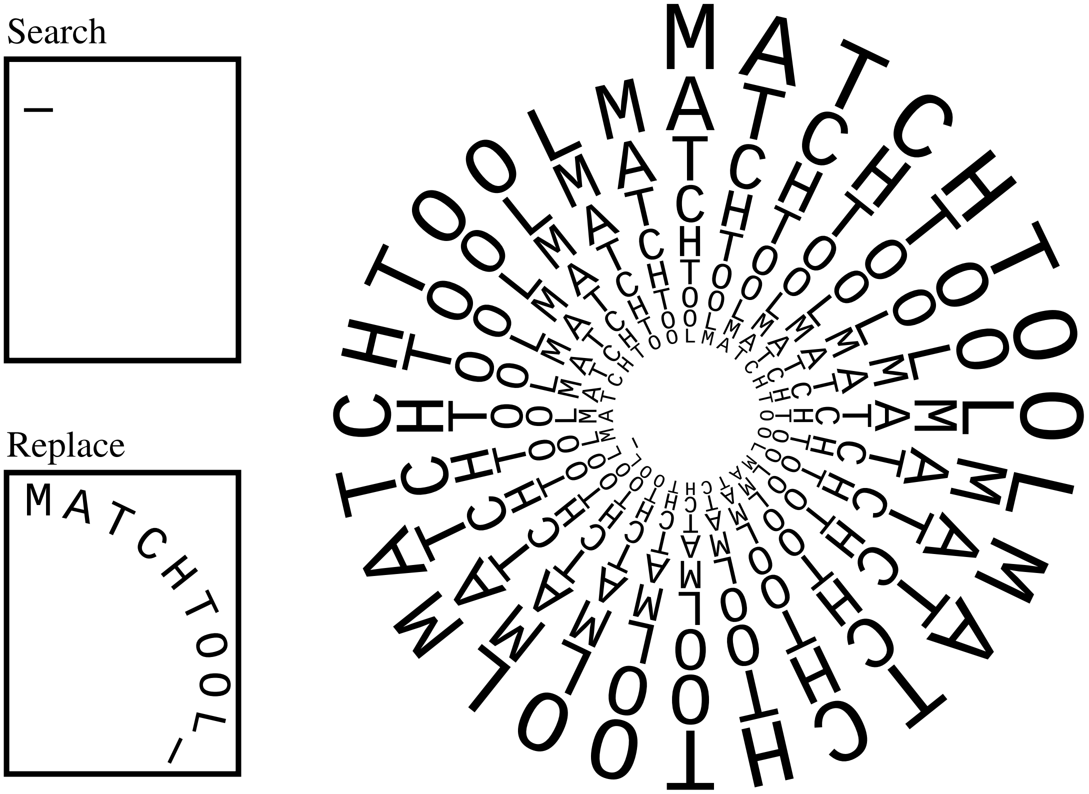


Figure 3.19 The MatchTool Spiral, and the rule that generates it.

Figure 3.19 presents a grammar to create a spiral. This spiral spells “MATCHTOOL” in two different directions: both along the arm of the spiral, and radially towards the inside. The spiral was formed by starting with a straight horizontal line, searching for the line, and replacing it with the word “MATCHTOOL”, along with another scaled and rotated line. Each time MatchTool finds a line and replaces it with new text and a line, the replacement gets its transformation from the match, so the new text spirals inward from the old. This production creates a spiral using a set of nine characters as the building block. However, the characters in the replace pane are also oriented in a spiral arc, so we created this spiral in a similar fashion, using individual characters as the building block. A simple calculation determined the relative rotation and scale that was applied to the line in the replacement pane to ensure that successive windings of the spiral align in the desired pattern, with the chosen spacing between windings.

Here is another example of a spiral grammar. This spiral, called the *Golden Spiral*, appears in Figure 3.20 along with the rule that generates it. The spiral is formed by starting with a rectangle whose dimensions form the golden ratio. The search and replace pattern finds such rectangles, and subdivides them into two rectangles. One of these rectangles is a square; the other is a new smaller rectangle with dimensions once again forming the golden ratio. The square contains a circular arc. As the replacement repeats, these new circular arcs join and spiral towards the center.

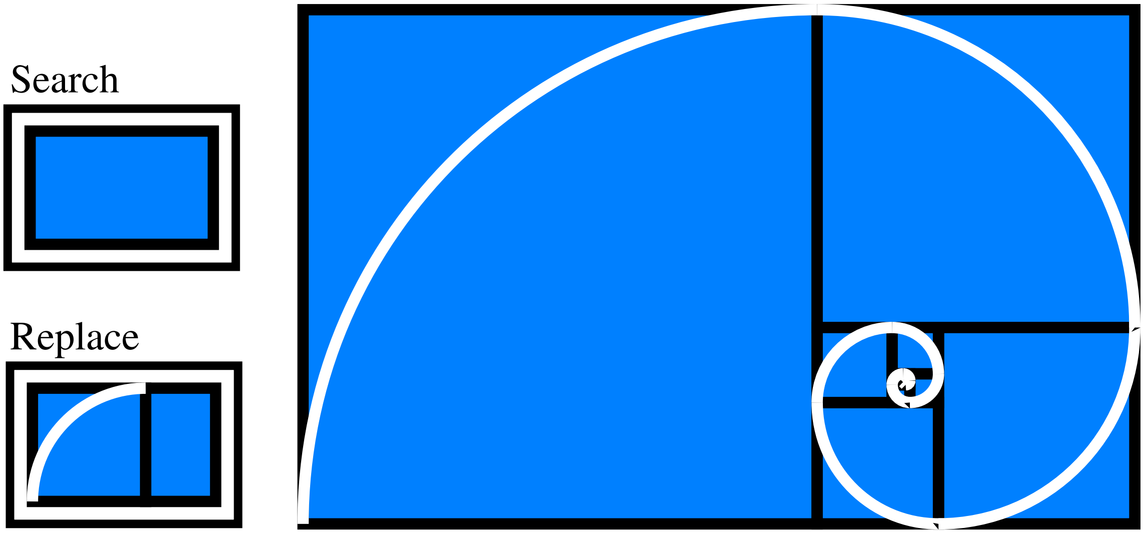


Figure 3.20 The Golden Spiral, and the rule that generates it.

Aside from providing hours of amusement, graphical grammars allow a large class of repetitive shapes to be defined simply, and declaratively. Graphical search and replace serves as a convenient technique for experimenting with graphical grammars, and generating complex objects according to example-based rules. The next section introduces another technique for creating scene complexity using graphical search and replace.

3.5.2 Graphical Templates

As described in Section 2.1.2.3, traditional templates are partially-completed textual documents with place-holders for the missing elements. *Graphical templates* extend this technique to the graphical domain. Any graphical object with a unique set of properties can act as a good placeholder. Graphical search and replace can be used to fill out graphical templates in a semi-automated fashion.

Since complex objects tend to be more difficult to create and position than simple objects, it is often easier to create a graphical template with simple objects acting as place holders, and replace these objects with more complex objects, than to draw the complex illustration from scratch. An earlier illustration, Figure 3.1, shows a yellow jay in an oak tree. The leaves could have been placed on the branches by selecting and copying an existing leaf, translating it so that it connects to a branch, rotating it to have the proper orientation, and scaling it to have the desired size. There are many leaves in the illustration, so this would quickly become tedious. The artist saved time by first drawing a skeletal version of the tree, shown in Figure 3.21, with black lines placed everywhere a leaf should appear. Other lines in the illustration, including those in the bird and other branches, all have a different color. Then graphical search and replace was used to replace every black line with an oak leaf. This requires fewer interaction steps, since drawing a line requires specifying only two endpoints, yet these endpoints completely determine the rotation, scaling, and positioning parameters. In addition to saving interaction steps, positioning simple place holders rather than complex objects allows for more rapid screen refresh during direct manipulation.

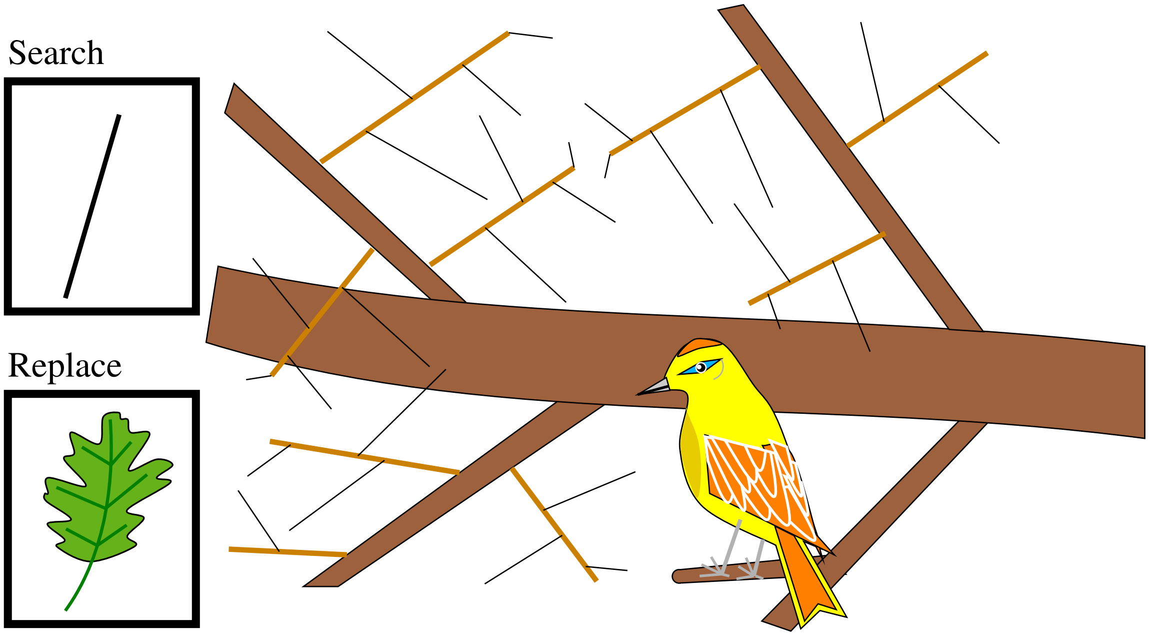


Figure 3.21 The graphical template used to generate the oak tree of Figure 3.1.

This particular use of graphical search and replace actually resembles abbreviation expansion (see Section 2.1.2.1), since easily specified input expands to much more complex document elements. However, usually in abbreviation expansion the input expands immediately, while here it expands in a subsequent pass. Chapter 4 describes a dynamic search and replace mechanism that expands graphical abbreviations immediately.

A second use for graphical templates is for saving complex graphical positioning relationships that can be re-used multiple times. For example, we can save libraries of lattices or spirals or frames with repeated place holders. These place holders need only share a single unique property to make the search and replace easy—they can vary in other graphical properties in useful ways. For example, we might build a template with place holders in a spiral, alternating through the colors of the rainbow. We could then fill in the template with new shapes, leaving the original colors alone, so that the filled in picture also forms a rainbow. Alternatively, we could specify that the new spiral should get its colors from the replacement objects.

The templates just described derive their entire benefit from archiving position and property information in the place holders. A third use of these templates is for archiving entire scenes, in which only some of the objects might vary from use to use. For example, clip-art libraries could be sold containing place holders embedded in the art. Accordingly, a company might buy one of these libraries with a commercial theme, and replace all uses of the generic company logo place holder with its own logo.

3.5.3 Graphical Macros

Graphical search provides a powerful iteration mechanism for graphical macros. Chapter 7 discusses Chimera's macro mechanism, and describes how MatchTool can add iteration to these macros. This section, however, addresses how graphical search supports an earlier graphical macro facility, called *mouseclick macros*.

Users define mouseclick macros within the MatchTool viewer itself. First users specify the search pattern in the normal fashion, and then they copy the search pane objects into the replace pane, and perform editing operations on them. These operations are recorded in a log, and can be played back on each match instead of replacing it. One twist is that during playback the coordinates of all mouse-based operations (*mouseclicks*) are first transformed into the coordinate system of the match. So if during the demonstration sequence, the user clicks on an object to select it in the replace pane, then when this operation is played back, the click is transformed to the location of the matched object.

A mouseclick macro could be used, for example, to perform the substitution of Figure 3.8 in a single iteration, without an ambiguous specification. Recall that the task is to replace triangles and circles, with squares and stars, giving each square the color of the triangle it replaces, and each star the color of the circle. The user first copies a triangle and circle into the search and replace panes. The user then starts recording the macro, and adds a square and star to the existing shapes in the replace pane. With a simple sequence of graphical editor commands, the color of the triangle is copied directly to the square, as is the color of the circle to the star. This avoids the correspondence ambiguity, since the mapping is implicit in the macro. Then the user deletes the original shapes, stops recording the macro, and invokes the search and macro operations on the scene.

Simply playing back the transformed operations might do the wrong thing if the scene is cluttered. For example in performing a selection, the scene object nearest to the mouse in the replace pane may not correspond to the object nearest to the transformed mouse position in the scene. To avoid this problem, we could limit the macro's effects to objects matched by the search or generated by the current invocation of the macro. A more robust search-based macro system would compose an explicit mapping between the search and replace pane objects, and the search pane objects and the match, to generalize the recorded object references. The next chapter discusses such mappings in more detail.

3.5.4 Graphical Grep

Graphical search also serves as the basis of a *graphical grep* facility. The UNIX™ grep program finds occurrences of a specified text pattern within a set of files. Similarly, graphical grep takes a list of editor scene files, possibly including wildcards, and determines which contain instances of the given graphical pattern. The pattern is specified using MatchTool, as in ordinary graphical search. When MatchTool finds the pattern in a file, it opens up an editor viewer on the file, selects the first match, and scrolls to make the match visible. Using MatchTool's command controls, the user can then cycle through all of the matches, either viewing them one by one or globally replacing a set of their graphical properties.

The traditional grep program allows regular expression search patterns. It is not clear what the graphical analogue to regular expressions is, and whether it would be useful in graphical search specifications. In a sense, MatchTool's search patterns always contain wildcards, since components of a single pattern can match objects located anywhere in the scene.

3.6 Conclusions and Future Work

Graphical search and replace provides a new mechanism for making repetitive, coherent changes to graphical documents. Unlike instancing and grouping, two other methods for making such changes, graphical search and replace has the advantage that no extra scene structure must be added in advance to support these changes. To experiment with this technique, we have built MatchTool, a direct manipulation, graphical interface for specifying graphical search and replace queries. Users construct these queries by supplying example search and replace objects, indicating the significant graphical properties, and setting a collection of parameters. These parameters tell MatchTool, for example, whether to ignore differences in rotation and scale, and how much tolerance to allow for shape matches. In addition to facilitating common, coherent changes, graphical search and replace can also construct shapes defined by graphical grammars, fill in graphical templates, provide flow of control for graphical macros, and serve as the basis for a graphical grep utility.

MatchTool's performance is very reasonable on a SparcStation 1+, a 1989-1990 class workstation (15.8 MIPS, 1.7 MFLOPS). In most of these examples, the searches were at least as fast as the replacements. Replacing black lines with leaves in Figure 3.21 took under 5 seconds (on the average, about 0.01 seconds for each match, 0.16 seconds for each replacement). Replacing oak leaves with maple leaves to convert Figure 3.1 to Figure 3.3 took about 9.5 seconds (on the average, about 0.17 seconds for each match, 0.17 seconds for each replacement). Replacements could be sped up significantly by improving the efficiency of Chimera's object copying code.

The original MatchTool was built during the summer of 1987, and released into the Cedar environment so that other people at PARC would benefit from it. Since the author completed the project just before leaving PARC, there was no one to champion its use, give hints, or answer questions on site. Merely putting a program on a server, and announcing its availability is a poor strategy for gaining a user population. A better approach would be to set up classes or personalized training sessions. Though occasionally people mention that MatchTool has helped them accomplish a task, it has not been used extensively. People often resist trying new things, particularly without encouragement. When faced with a task, users often approach it as they have done in the past, rather than taking a moment to wonder if it can be solved in a new or better way.

Perhaps one factor limiting MatchTool's use is that Cedar environment had a limited system resource (gfi's - global frame indices) that was partially consumed with every new application launched. Once this resource ran out, the Dorado workstations would hang and need to be rebooted. This made people very conservative about launching applications that they absolutely did not need. In most cases graphical search and replace is not a necessity, but a convenience—the user can repeat the same changes by hand to objects located throughout the scene.

Eric Bier also observed that the interface may be too heavyweight for many tasks. Traditional textual search and replace usually can be conducted within the text editor application itself, instead of a secondary utility. Ideally there would be a mechanism for invoking the most common classes of graphical search and replace tasks directly from the graphical editor. Less common tasks would continue to require the full MatchTool interface. Potentially this might help solve a related problem: the number of options and widgets sometimes confuses even people that have a good understanding of MatchTool's full capabilities. It is easy to forget to set one parameter, have the search fail, and be puzzled for a minute before noting the rogue widget.

Graphical editing tasks that can be accelerated by graphical search and replace arise all the time. For example, after printing an initial version of Figure 3.3 on a grey scale device, we found that the colors of leaves (which were drawn on a color display) mapped to the same grey value as their veins and stems, making them indistinguishable. Graphical search and replace made what would otherwise be a tedious editing repair job almost trivial. After preparing an initial version of Figure 3.15, we realized that the CH_3 component of the molecule was drawn much too large and the font was too light. Graphical search and replace made it easy to change the size and family of the text throughout the picture, while keeping each occurrence centered above its bond. If graphical search and replace were not available, we would not have undertaken the chore by hand, so in this case the technique resulted in a better picture rather than a time savings.

Graphical search and replace can be extended and improved in a number of ways. In MatchTool, the search and replace properties are fixed. A better, object-centered approach would determine the significant properties of the objects in the panes, and present these and only these as options. For example if the user places a checkbox in the search pane, MatchTool can search for objects of the same class. However, MatchTool cannot search for objects with the same *value* (on or off), since this property does not apply to the graphical domain for which MatchTool was originally designed. MatchTool 2 was coded using object-oriented techniques, and it would not be difficult to extend it in this way.

A current limitation of the MatchTool interface is that different sets of search and replace properties cannot be specified for objects in the same pane. For example, it is impossible to use the interface to a search for squares and red circles together, since the color property in the search column must be either set or unset for all objects in the search pane. This is a limitation of the interface rather than the search and replace code, since the code can be readily extended to perform such tasks. Perhaps the user should be able to select a subset of objects in the panes, and specify their property vector independently.

Though MatchTool's shape matching metric works well for exact matches, it is unintuitive for inexact matches. MatchTool would benefit from an inexact shape matching metric that much more closely matches that of the human visual system. As an experiment in improving this component of MatchTool, we built a system for matching runs of curves using the minimal cost string-edit algorithm [Sankoff83]. A run of curves can be discretized into a one dimensional string, with characters representing directions. The similarity of two runs can be compared by calculating the cost of editing one string into the other. Burr used a similar approach for handdrawn character recognition [Burr79]. This appears to be much more intuitive than MatchTool's current metric for inexact matches, but we have yet to extend the technique to matching multiple runs of curves.

As mentioned in the last section, the number of widgets in the MatchTool window can be daunting. It would be better to provide a simplified interface for the most common searches. Perhaps this interface should be incorporated into the graphical editor window to make it more convenient. Developing a regular-expression-like capability for MatchTool would be interesting. First it would be necessary to develop an analogue to regular expressions in the graphical domain, and then determine whether people really want to search for them!

Now that we have a graphical grep, perhaps we should build a graphical *diff*. Diff is a UNIX™ program for comparing two text files. It would be handy to have a similar program that compared two graphical editor scene files, graphically illustrating how they differ. This program could also help with performing regression tests on graphical editors.

Graphical search and replace could also work cooperatively with an instancing system. It could be used to find all similar objects in a graphical scene, and subject to the user's approval, turn them into instantiations of the same master object. This could make terser scene files, and add extra structure after the fact to make subsequent scene changes easier.

The original MatchTool had no means of searching for geometric relationships. For example, we would like to have been able to search for text in boxes, regardless of the precise positioning of the text, or nearly touching lines, ignoring their length or orientation. This "future work" has since been implemented, and is the subject of the next chapter.

> *Every house worth considering as a work of art must have a grammar of its own. "Grammar," in this sense, means the same thing in any construction—whether it be of words or of stone or wood. It is the shape-relationship between the various elements that enter into the constitution of the thing. The "grammar" of the house is its manifest articulation of all its parts. This will be the "speech" it uses. To be achieved, construction must be grammatical.*
>
> — Frank Lloyd Wright, *The Natural House*

Chapter 4

Constraint-Based Search and Replace

4.1 Introduction

When repetitive changes must be made to a document, there are several approaches to consider. The changes can be performed by hand, which is tedious if there are many modifications to make or if they are complex to perform. Custom programs can be written to perform the changes automatically, but this requires programming skill, and familiarity with either the editor's programming interface or file format. Some editors, particularly text editors, allow macros to be defined by demonstration. These macros do not however extend easily to domains, such as graphical editing, where it is difficult to assign an unambiguous meaning to each interaction.

As most users of text editors are keenly aware, another approach to making repetitive changes involves the use of search and replace. The last chapter described how we extended this technique to the 2D graphical editing domain by building a utility called MatchTool. Using MatchTool, we could search for all objects matching a set of graphical properties, such as a particular fill color, line style, or shape, and change either these or a different set of properties. However, there was a large class of search and replace operations that MatchTool could not perform. There was no way to search for a particular geometric relationship, because shape-based searches matched on the complete shape of the pattern. For example, MatchTool could search for triangles of a particular shape, but not all right triangles. Similarly, shape-based replacements would substitute the complete shape of the pattern without any way of preserving particular geometric relationships in the match.

By adding constraints to the search and replace specification we now have better control of which features are sought and modified. Many complex geometric transformations can be expressed using constraint-based search and replace. For example, the user can now search for all pairs of nearly connected segments in the scene, and make them connected, or search for connected Bezier curves and enforce tangent continuity between them. All text located in boxes can be automatically centered, or boxes can be automatically created and centered around existing text. Lines parallel within a chosen tolerance can be made parallel. These object transformations can be used, for example, to beautify roughly drawn scenes, and enforce other design decisions.

Several other systems use automatic constraint generation for scene beautification. Pavlidis and Van Wyk's illustration beautifier searches for certain relationships, such as nearly aligned lines or nearly coincident vertices, and enforces these relationships precisely [Pavlidis85]. Myers' Peridot uses a rule set to find particular relationships between pairs of scene objects and establish new constraints among them [Myers88]. The system described here differs from these in that the constraint rules can be defined by the system's users, thereby providing a powerful new form of editor extensibility. Rules are defined by direct manipulation [Shneiderman83], using the same techniques that are used for editing ordinary scenes. Furthermore, users can view the constraint rules graphically, in the same visual language as the rest of the editor interface. As will be described later in this paper, simple demonstrational techniques help in defining these rules.

The ability to create custom rules is particularly important, now that methods have been developed to allow the user to define new types of constraints with little or no programming. For example, Borning's ThingLab allows new constraint classes to be defined using graphical techniques [Borning86], and several recent systems allow new constraints to be entered via spreadsheets [Lewis90] [Hudson90b] [Myers91a]. If the system designers cannot foresee every constraint that may be necessary, they clearly cannot provide for every transformation rule based on these constraints.

Search and replace is also a particularly easy way to add constraints to similar sets of objects. Sutherland's Sketchpad [Sutherland63b] introduced instancing to facilitate the same task. Instancing, though an extremely useful technique with its own benefits, requires that the user know, prior to object instantiation, the types of constraints that will be used. Also, many instancing systems place objects in an explicit hierarchy, not allowing one object to be a member of two unrelated instances. Constraint-based search and replace has neither of these limitations.

Nelson's Juno allows constraints to be added either in a WYSIWYG view or a program view, resulting in a procedure that can be parameterized and applied to other objects [Nelson85]. Constraint-based search and replace rules are implicit procedures that are specified through a direct manipulation interface. The procedures are parameterized through the search portion of the rule, which also specifies the criteria that an object must match to be a valid argument. When a replacement rule is to be applied many times, the search mechanism reduces the burden by finding appropriate argument sets automatically.

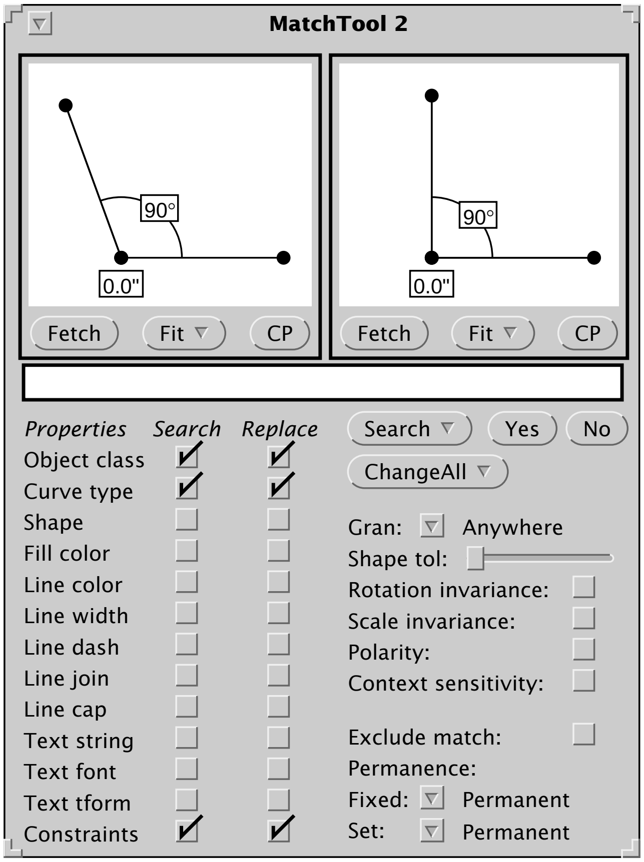
Constraint rules have been used by several researchers in human-computer interaction. Weitzman's Designer was supplied with a set of rules to enforce design goals [Weitzman86]. Peridot's rules, written in Interlisp, infer geometric constraints among objects in an interface editor. Maulsby's Metamouse graphical editor infers graphical procedures by demonstration [Maulsby89a]. Each program step is associated with a set of preconditions and postconditions to be met, which can include "touch" constraints. Vander Zanden developed a method of specifying graphical applications using constraint grammars to isolate the visual display from the data structures [Vander Zanden89].

The second implementation of MatchTool, MatchTool 2, has a constraint-based search and replace feature that, like the other techniques described in these chapters, works in conjunction with the graphics and interface editing modes of the Chimera editor. This chapter describes the motivation, interface, and implementation of constraint-based search and replace. The next sections introduces the capabilities of MatchTool 2 and its interface through a series of examples. Later sections discuss the algorithm and implementation. The last section summarizes this work and presents planned future work.

4.2 Example 1: Making A Nearly Right Angle Right

Suppose that we would like to specify that all pairs of connected lines *nearly* 90° apart should be *precisely* 90° apart. Figure 4.1 shows MatchTool 2's interface for doing this.

When the "Constraints" box is checked in the search column, MatchTool 2 searches the scene for *relationships* expressed by the constraints in the search pane. The scene may contain constraints too, but the search ignores these. For example, there are two constraints in the search pane of Figure 4.1. The zero length distance constraint connects an endpoint of each line. It indicates that MatchTool 2 should look for two segments that touch at their endpoints. The 90° angle constraint between the two lines specifies that the lines matched must meet within a given tolerance of a right angle. Constraints are shown in these figures as they appear in the editor, and the display of individual constraints can be turned on and off from a constraint browser. Appendix B documents the constraints that Chimera currently supports, and Appendix C describes Chimera's visual representation for these constraints and its interface for creating and viewing them.



Properties	*Search*	*Replace*
Object class	☑	☑
Curve type	☑	☑
Shape	☐	☐
Fill color	☐	☐
Line color	☐	☐
Line width	☐	☐
Line dash	☐	☐
Line join	☐	☐
Line cap	☐	☐
Text string	☐	☐
Text font	☐	☐
Text tform	☐	☐
Constraints	☑	☑

Gran: Anywhere
Shape tol:
Rotation invariance: ☐
Scale invariance: ☐
Polarity: ☐
Context sensitivity: ☐

Exclude match: ☐
Permanence:
Fixed: Permanent
Set: Permanent

Figure 4.1 A constraint-based search and replace rule.

4.3 Tolerances By Example

We intend that the pattern should match all angles that are roughly 90°, so we need a way to specify the tolerance of the search. We use a simple demonstrational technique. When constraints are turned off, objects can be moved from their constrained positions. The user shows how far off a particular relation can be by demonstrating it on the search pattern. The search pattern above matches angles of 90° ± 20°, since the angle drawn is 110°, and the constraint specifies 90°. To represent an asymmetric range (e.g., 90° +0° –20°), we can simply convert it into a symmetric range about a different value (e.g., 80° ± 10°). In the search specified by Figure 4.1, the distance constraint must be satisfied exactly since the endpoints coincide in the search pane. We also provide an option that lets the user arrange the search pattern into several configurations, and takes the maximum deviation.

When we were first developing the system, we used MatchTool 2’s shape tolerance slider for specifying constraint tolerances. The results were quite unsatisfactory because this control adjusted the tolerances of all the constraints simultaneously. Also, there was no visual clue relating how far off a constraint could be for a given position of the slider.

4.4 Additional Search And Replace Parameters

In the lower right hand corner of Figure 4.1, underneath the ChangeAll button, is a set of controls that affect how the search and replace is performed. Most of the parameters described in the last chapter are still useful, but others have been added as well. The new controls appear at the bottom of Figure 4.1, starting with *Exclude match*.

4.4.1 Match Exclusion

In the original MatchTool, scene objects that match the pattern are excluded from future matches. When dealing with constraints, this behavior is usually undesirable. For example, in our search for angles of nearly 90°, we do not want to rule out other potential matches involving some of the same objects, since segments can participate in multiple angle relationships. When *Exclude match* is selected, scene objects can match at most one time in a single search. When it is not selected, scene objects can match multiple times; however to insure that the search will halt, no permutation of objects can match more than once.

4.4.2 Constraint Permanence

Just as constraints in the search pane indicate relationships to be sought, constraints in the replace pane specify relationships to be established in each match. To establish the relationship, MatchTool 2 applies copies of the replacement constraints to the match, and solves the system. Whether or not constraints remain in the scene after the match is a user option. The user has three choices: the constraints can be removed from the match immediately upon solving the replacement system, they can be removed after *all* the replacements have been made, or they can be made permanent. In the current example, it is important to choose one of the latter two options. A segment can participate in multiple angles, and if we delete the angle constraint immediately upon making a replacement, a

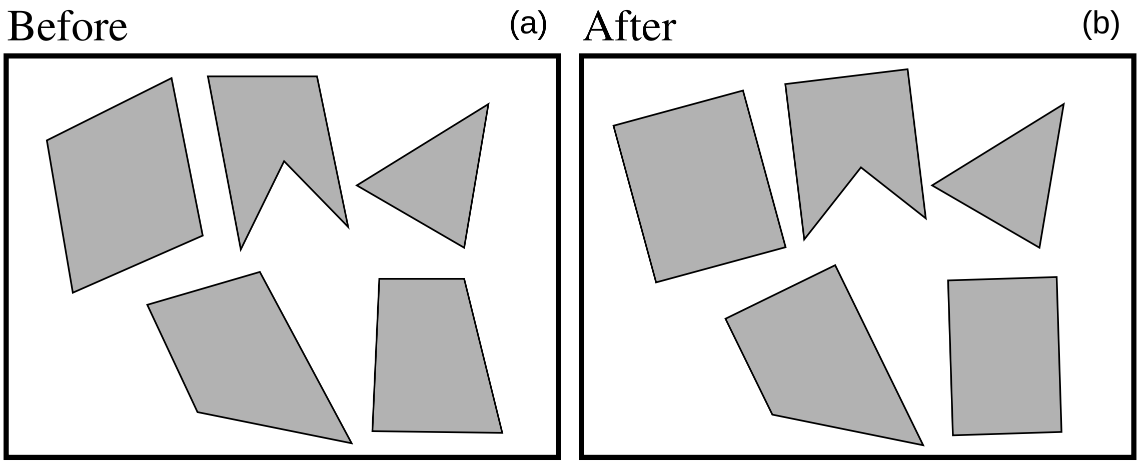


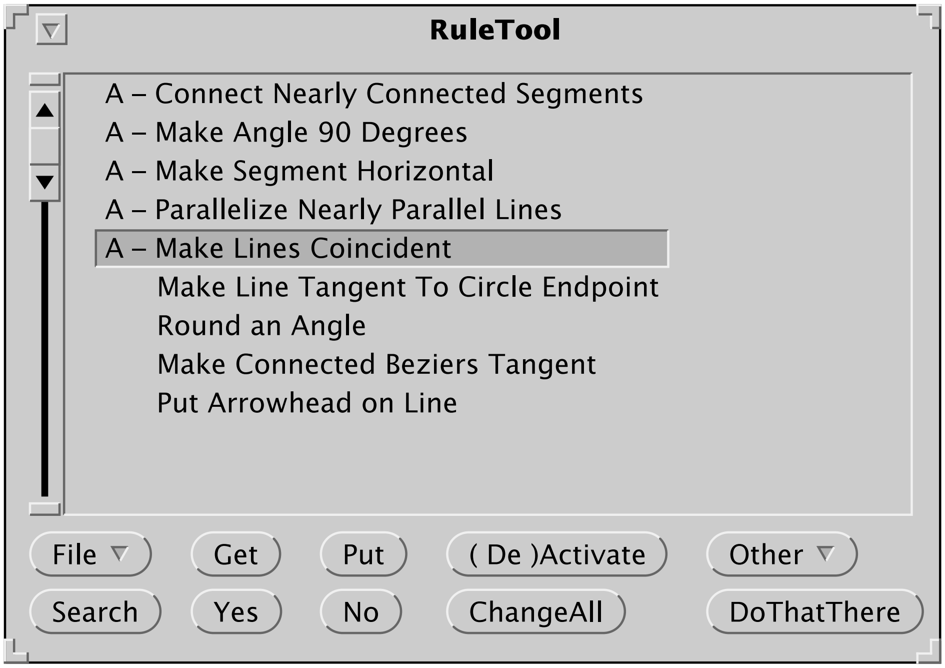
Figure 4.2 Application of the rule to make right angles out of nearly right angles. (a) The initial editor scene; (b) The modified scene.

subsequent application of constraints might destroy the established relationship. As will be discussed later, there are two different classes of constraints: *fixed constraints* and *set constraints*. The *permanence* of each class is controlled independently.

With these parameters now set, we are ready to begin the search. First we place the software cursor in the editor scene shown in Figure 4.2a, to indicate the window in which the search should occur. Then we press the ChangeAll button in the MatchTool 2 window. All of the 90° ± 20° angles become true right angles, as shown in Figure 4.2b.

4.5 Rule Sets

Many constraint-based rules have wide applicability, so an archiving facility is important. Once search and replace rules have been defined, they can be saved in libraries, or *rule sets*. We have built a utility for manipulating rule sets, called the RuleTool, and its interface is shown in Figure 4.3.



A – Connect Nearly Connected Segments
A – Make Angle 90 Degrees
A – Make Segment Horizontal
A – Parallelize Nearly Parallel Lines
A – Make Lines Coincident
Make Line Tangent To Circle Endpoint
Round an Angle
Make Connected Beziers Tangent
Put Arrowhead on Line

Figure 4.3 The RuleTool interface, a utility for building libraries of rules.

The RuleTool contains a scrolling list, in which rules are catalogued. Rules are initially defined through the MatchTool 2 interface, but can be loaded into the RuleTool. Once in the RuleTool, a rule can be executed directly without using the MatchTool 2 window at all. A single rule can be selected and executed, or a set of activated rules can be executed in sequence in the order listed (activated rules are preceded with an “A”). The user can execute rules in the RuleTool as a post-process, after the illustration has been completed, or dynamically as objects are added or modified, in the manner of Peridot. We refer to the latter mode as *dynamic search and replace*. When a match is found, the match is highlighted, and the name of the rule is displayed. To invoke the rule, the user hits the “Yes” button, otherwise “No”. Though in some cases it may be ambiguous which selected object corresponds to which object in the rule, rule executions can easily be undone if they have unexpected results.

Figure 4.4 shows the results of executing the activated rules of Figure 4.3 on a rough drawing of a house. Figure 4.4a is the initial drawing, and Figure 4.4b is the neatened version. The user explicitly accepted or rejected each of the matches. The results have the flavor of a simple version of Pavlidis and Wyk’s drawing beautifier. In contrast, however, all of the rules used to neaten the drawing of the house were specified interactively, by the user, with constraint-based search and replace. These rules are simple rules and might have realistically been pre-coded by the implementer. The next two examples demonstrate more sophisticated tasks that introduce other capabilities of the system.

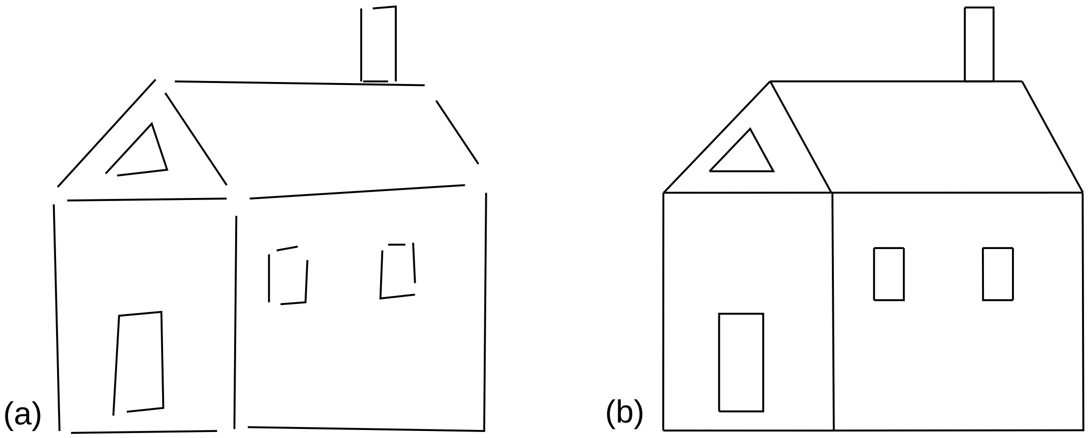


Figure 4.4 Renovating a house. (a) The original house; (b) The house after application of the activated rules in the rule set of Figure 4.3.

4.6 Example 2: Making A Line Tangent to a Circle

As the Gargoyle graphical editor was being developed at Xerox PARC, several people proved their drafting prowess by constructing letters from the Roman alphabet, following the instructions described by Goines [Goines82]. An example of this is included in [Bier86]. These constructions consist of a small number of tasks that are repeated many times, some of which are difficult to perform or require some geometric knowledge. At the time, we tried our hand at one of these constructions, and felt that a macro facility would be extremely helpful, particularly since others had drawn letters before us and presumably would have written the macros. One particular task that we found difficult was making a line through a specified point be tangent to a circle. Here we show how this task can be encapsulated in a constraint-based search and replace rule.

We would like to find lines with one endpoint nearly tangent to a circle, and make that endpoint precisely tangent. The search pattern is given in Figure 4.5a. Since our system currently has no tangency constraints, the user expresses the relationship with two constraints. The distance between the center of the circle and its other control point is constrained to be the same as that between the center of the circle and the near endpoint. This expresses that the endpoint lie on the circle. As shown in Figure 4.5, this distance constraint is represented in Chimera by a “D” connecting the two equi-length vectors. Also, the angle formed by the line’s far endpoint, its near endpoint, and the center of the circle is constrained to be 90°. (Actually, there are two lines tangent to a circle through a point, and we should be looking for –90° angles as well. After defining our rule we can easily copy it and modify it to catch the second case). Since we would like to match objects that nearly fulfill the given constraints, we manipulate the objects to show how much tolerance should be assigned to each constraint. Next, the objects in the search pane are selected and copied into the replace pane, shown in Figure 4.5b.

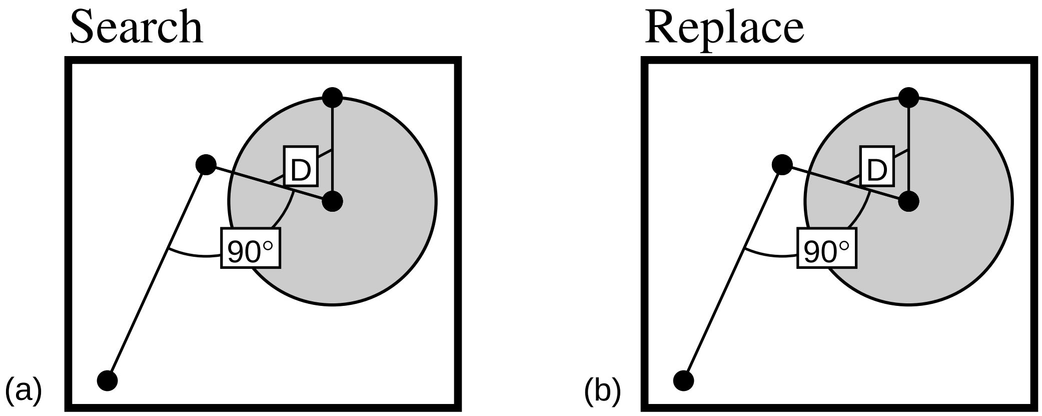


Figure 4.5 Line-endpoint and circle tangency rule. (a) search pane; (b) initial replace pane.

One helpful test to find mistakes in the replacement specification is to invoke the constraint solver on the replacement pattern directly, and confirm that the objects adopt the desired configuration. The reason why this works is that typically the search pane is copied to the replace pane as an initial step, and the search pane contains a valid match. If the constraints already on these objects or subsequently added to them bring the match into the desired configuration, then they are fulfilling their job. The result of invoking the constraint solver is shown in Figure 4.6.

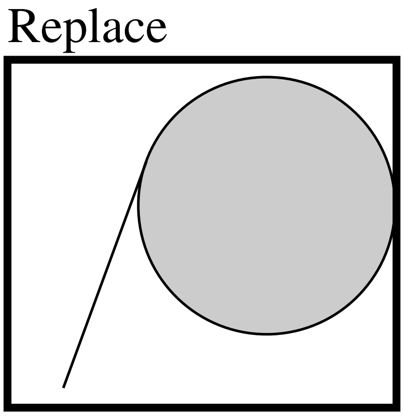


Figure 4.6 A test reveals a problem in the replacement specification.

Though the line has indeed become tangent to the circle at one endpoint, the result is not exactly what we had in mind. The circle expanded to meet the line, and both endpoints of the line moved as well. We would like to refine our specification to allow only the near endpoint of the line to move. To do this, we undo the last command (using Chimera's undo facility), putting the system back into its prior configuration, and specify additional constraints, as explained next.

4.7 Fixed Constraints And Set Constraints

In this example there is a fundamental difference between the constraints already specified and those still to be added. The existing constraints specify relationships that must be changed in the match—two distances must be made equal and an angle must be set to 90°. Additional constraints are needed to fix geometric relationships of the match at their *original* values. When we match a circle and line in the scene, we want to fix both the circle and the far endpoint of the line at their locations in the match, not their locations in the replace pane. We refer to the first type of constraint that *sets* geometric relationships to their values in the replace pane, as *set constraints*. Constraints that *fix* geometric attributes of the match at their original values are called *fixed constraints*.

At first thought it may not be clear why fixed constraints are necessary at all. One might think that only the geometric relationships explicitly expressed by set constraints in the replace pane should be changed in the match, and all other relationships should remain invariant. However, this is not possible—changing some relationships automatically results in others being changed as well. In the general case, it is impossible to make an endpoint of a line tangent to a circle, keeping the center of the circle, its radius, the other endpoint of the line, and the line's slope and length fixed. Given that some of these relationships must change, it is important to allow the user a choice of which.

Fundamentally, the difference between set and fixed constraints is the difference between checking and not checking an attribute in the replace column of MatchTool. During replacements, checked attributes come from the replace pane, and unchecked attributes come from the match. We considered adding entries to the replace column for constraints specified in the replace pane, and using checkboxes to specify the source of the constraint value. However this would clutter the interface. Instead, we have developed a simple demonstrational heuristic for determining whether the constraint should come from the match or the replace pane, without the user having to reason about it.

The heuristic requires that the search pane contents initially be copied into the replace pane, as is a common first step in specifying the replacement pattern. As a result, the replace pane contains a sample match. The user then adds constraints to this sample match, transforming it into a valid replacement. Conceptually, the user demonstrates the constraints to be added to all matches by adding constraints to this example.

We have two different menus for specifying constraints, either of which may be used in a MatchTool 2 pane or in an arbitrary scene. Users can select commands from the FIX menu to fix relationships at their current value, or alternatively they can specify a new value by using the SET menu and typing the new value into a text input widget. It is always easier to use FIX to make an existing relationship invariant, and it is usually simpler to use SET rather than manually enforcing the relationship and invoking FIX. This interface for specifying constraints is described in Appendix C. MatchTool 2 keeps track of whether the user chooses FIX or SET and creates fixed or set constraints accordingly.

This heuristic works well, but occasionally the user does not take the path of least resistance or a relationship that we would like to enforce is coincidentally already satisfied in the sample. In these cases, the user may choose the wrong interface or forget to instantiate the constraint at all. To further test the constraint specification, constraint enforcement can be temporarily turned off, the replace pane objects manipulated into another sample match, and a Verification command executed. This command resets all fixed constraints to their values in the new configuration, and the constraint system is re-solved. If this second example reconfigures correctly, it is a good indication that the specification is correct. Fixed and set constraints are displayed differently in the replace pane (fixed constraints are marked by an asterisk), and the interface contains a command that converts between types.

We now return to the task of making the near endpoint of a line tangent to a circle, while moving only this endpoint. After selecting both control points of the circle and the far

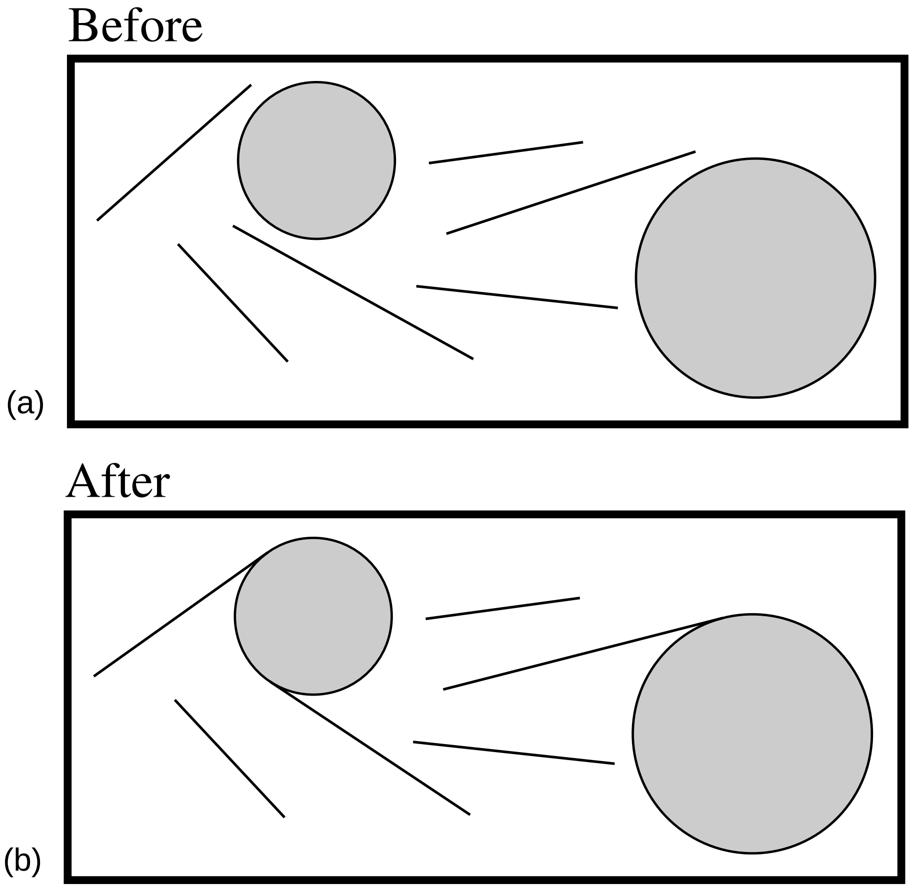


Figure 4.7 Applying the tangency rule. (a) a scene containing lines and circles; (b) nearly tangent lines made tangent.

endpoint of the line, we select “Fix Location” from a menu. We now solve the system in the replace pane, and it reconfigures in the desired way, indicating the replacement specification is probably correct. When Chimera is given the sample scene of Figure 4.7a, our rule makes all line endpoints that are nearly tangent to circles, truly tangent, resulting in the scene shown in Figure 4.7b.

4.8 Do-That-There

While for certain editing tasks it is useful to keep rules such as line-circle tangency active, they can interfere at other times by matching too frequently if the tolerances are low, and by slowing down the editor. An extension to the current system that would allow tolerances to be scaled up without editing the rules would be helpful in the first case. Another approach is to keep all but the most necessary rules inactive, and require that the user explicitly invoke other rules on an as-needed basis.

We provide two new facilities for this. In both versions of MatchTool, forward searches proceed from the position of the software cursor towards the bottom of the scene. A new "Do-That-There" command orders the search roughly radially outward from the cursor. The search is invoked using the rule selected in the RuleTool, and the MatchTool interface is circumvented entirely. As is the case with regular MatchTool searches, the matches can be accepted or rejected with "Yes" and "No" buttons. Another option invokes a chosen rule only on selected objects, so for example, selected quadrilaterals can be transformed into rectangles by choosing the "Make Angle 90 Degrees" rule from the RuleTool, and invoking the RuleTool's "ChangeAll" button.

4.9 Example 3: Rounding Corners

As we were developing the first MatchTool, we accumulated a list of editing tasks that would be facilitated by an ideal graphical search and replace utility, and evaluated our implementation by determining which tasks it could actually solve. A task suggested by Eric Bier was to "round" 90° corners, that is, splice an arc of a given radius into the angles while maintaining tangent continuity between the arc and the lines. We perceived this as difficult, because we thought it would be necessary to match on the shape of pieces of segments. Though neither MatchTool implementation can perform this kind of matching, in our third example we show how constraint-based search and replace can perform the rounding task not only for 90° angles, but for arbitrary ones. In this example, the replacement rule adds a new, constrained object to the scene, which is a type of replacement beyond the capabilities of other existing scene beautifiers.

The search pattern shown in Figure 4.8a, matches on two segments, meeting at 90°. Since the lines in the search pane are part of a single polyline, and we have chosen to search on object class, curve type, and constraints, MatchTool 2 will match only pairs of joined line segments that are part of the same polyline. We copy the search pane contents into the replace pane, and delete the angle constraint. The replace pane now contains a representa-

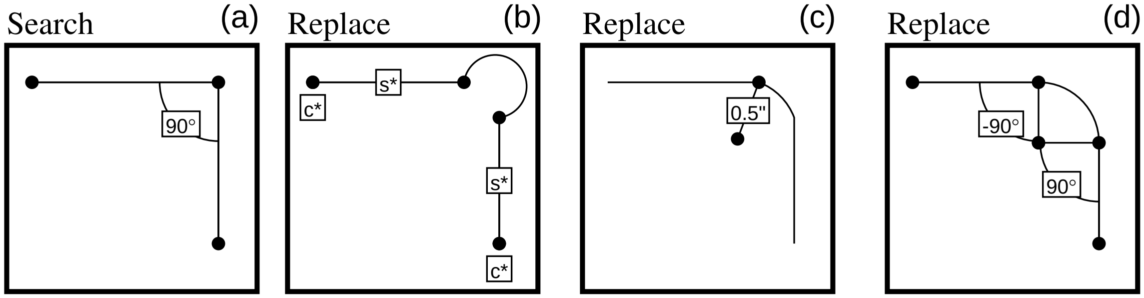


Figure 4.8 Rounding corners. (a) search pane; (b) replace pane after splicing in an arc, and adding fixed location and fixed slope constraints (labeled c* and s*, respectively); (c) replace pane after adding a 1/2 inch distance constraint on the arc's radius (the figure has been reduced); (d) replace pane after adding two more angle constraints.

tive match that we will, step by step, transform into its replacement. Figure 4.8b-4.8d shows the steps in this sequence.

First we fix the far endpoints at their current locations, since they should not move, and we fix the slopes of the line segments as well. We shorten the segments a bit, and splice in an arc, producing the pane shown in Figure 4.8b.

A few additional constraints still must be added. Though the arc implicitly constrains both of its endpoints to be the same distance from its center, we still need to constrain its radius. Eventually we plan to allow numerical parameters for replacement rules, but in this example we set the radius to a constant of one half inch. This constraint is shown in Figure 4.8c. Finally, we add two additional constraints to ensure tangency. The angle formed by the arc's center, the near endpoint of the top line, and the far endpoint, is set to –90°, and the corresponding angle formed with the other line is set to 90°. These final constraints are shown in Figure 4.8d (Note that for each of these figures we have turned off the display of constraints not added by the step.).

The representative match has now been completely transformed into the desired replacement. After applying the rule to the "F" in Figure 4.9a, the corners are correctly rounded, producing Figure 4.9b.

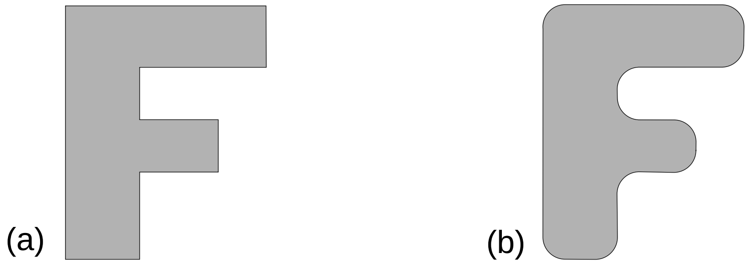


Figure 4.9 Rounding right angles in an F. (a) The unrounded version; (b) After application of our rounding rule. (This figure has been reduced.)

The rule can easily be generalized to round all angles. In fact, the constraints added to the replace pane will already round any angle between 0° and 180°, provided it can be rounded. We need only add a 90° tolerance to the search pattern, making the entire rule work for angles between 0° and 180°. Given two lines that meet at an angle ABC, either this angle or its reverse, angle CBA, is between 0° and 180°. Thus this search pattern will match any pair of connected lines, and the convex angles of the search pattern and the match will be aligned, which is important for the replacement. Applied to the "N" of Figure 4.10a, the rule rounds all the angles, producing Figure 4.10b.

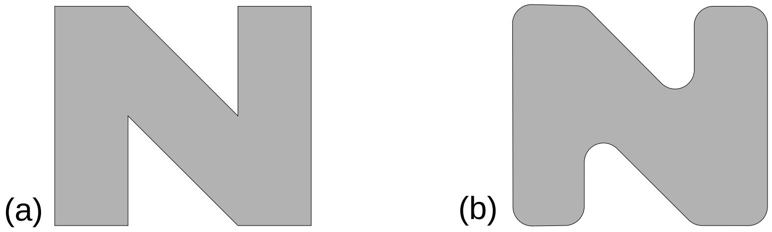


Figure 4.10 Rounding arbitrary angles in an N. (a) The unrounded version; (b) After application of the generalized rule. (This figure has been reduced.)

4.10 Example 4: Wrapping Rounded Rectangles Around Text

This section presents one approach to automating a task posed by Richard Potter and David Maulsby in their Programming by Demonstration Test Suite [Potter93b]. While using MacDraw II, a member of Richard's lab found it a chore to wrap rounded rectangles around the many text fields in a graphical document. Each rectangle had to be centered around the text, a fixed amount larger than the string it enclosed. The Test Suite describes one solution to this problem using Triggers, and two different solutions using Chimera. A solution using constraint-based search and replace is explained here, and Section 7.4 presents an alternative approach based on Chimera's macro by example facility.

Initially we type a sample piece of text (in this case, just the string "Text") in the search pane, shown in Figure 4.11a. Since we are interested in matching any text in the scene, we set only the object class attribute in the search column. After copying the text from the search pane into the replace pane, we add fixed location constraints to the text's vertices. This ensures that the text string will not be moved by constraints to be added later. A rounded rectangle is added to this pane, and moved behind the existing text. We set the distance between each corner vertex of the text, and its corresponding vertex of the rounded rectangle to be a fixed distance (here, 0.5"), and we also add angle constraints between each vertex of the rounded rectangle and two vertices of the text. For example, one of these constraints might set the angle between the upper right vertex of the rounded rectangle, the upper right vertex of the text, and the upper left vertex of the text to be 135°. The replace pane of Figure 4.11b displays this constraint, and the other two constraints relating to the upper right vertices of the rounded rectangle and text. Symmetric constraints have also been added to the other three vertices, making 12 constraints in all, but to reduce clutter they do not appear in the figure.

The replace pane now contains constraints to specify the size and orientation of the rounded rectangle relative to the text. These same relationships could have been expressed with other types of constraints available in MatchTool 2. By checking constraints in the replace column, we indicate that only the geometry of objects matching the search pattern can change, as well as objects, such as the rectangle, that must be added to the scene since they have no counterpart in the search pane.

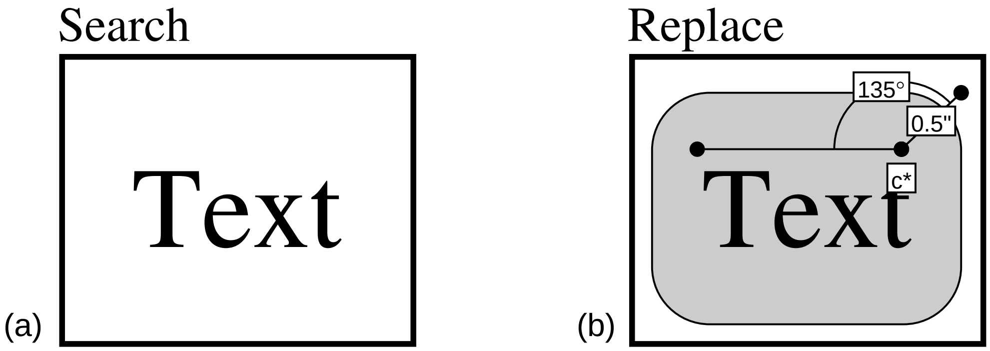


Figure 4.11 Wrapping a rounded rectangle about text. (a) The search pane; (b) The replace pane. To reduce clutter, only constraints on the upper right portion of the illustration are shown.

Now that this replacement is defined, we can apply it to the scene in a variety of ways. We can add rounded rectangles around all strings, or only those strings for which we confirm the replacement. We can move the caret to a particular string and execute "Do-That-There". This replacement specification even allows us to place rounded rectangles around rotated text, oriented in the same direction as the text. Figure 4.12a shows a scene containing several text strings of different sizes, positions, and orientations. After applying the replacement rule to all of the text in the scene, Figure 4.12b results.

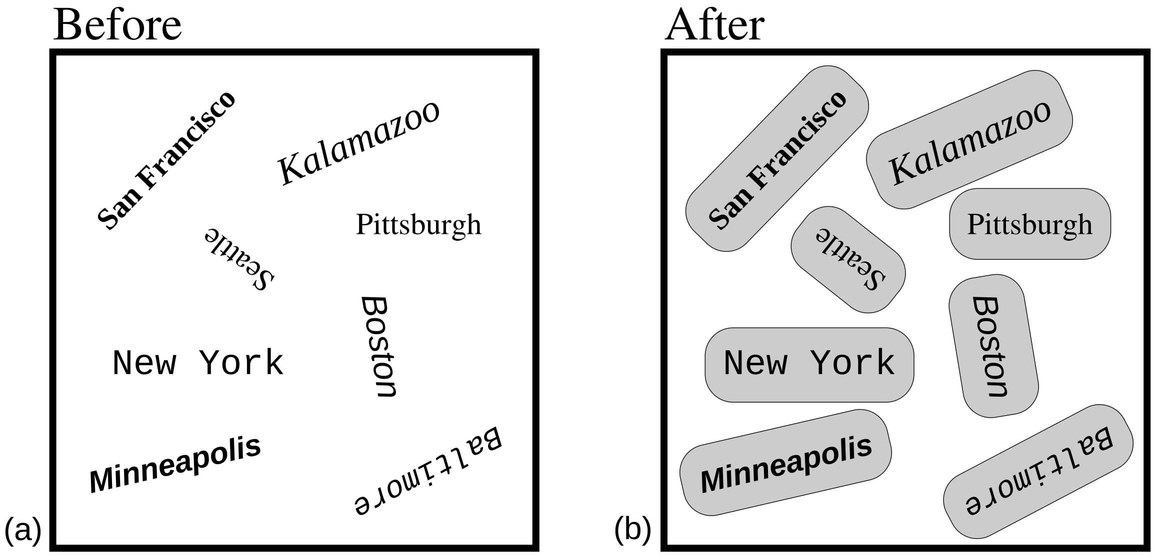


Figure 4.12 Applying the rounded rectangle rule. (a) The initial scene; (b) The scene afterwards. (This figure has been reduced).

4.11 Algorithm

Objects in both the scene and the search pane can be viewed as nodes in a graph, with constraints linking the nodes. Therefore, matching the search pattern to the scene requires finding occurrences of one graph within another. This is known as the subgraph isomorphism problem, and the bad news is that it is NP-complete for general graphs [Garey79]. The good news is that for a search pattern of fixed size the matching can be done in polynomial time, and for typical replacements the exponent is very small. To match the n elements of the search pattern to the m elements of the scene, the cost is $O(cm^n)$, where c is the number of constraints in the search pattern. Each of the examples presented here has a search pattern of two objects (examples 1 and 2) or less (polylines in search patterns, as in example 3, count as single objects), so the costs are $O(m^2)$ and $O(m)$, respectively. For dynamic search and replace, when a single object is created or modified, the search proceeds even faster, since we know this object must participate in the match. In this case the exponent is reduced by one, and the search costs for the examples in this paper are linear or constant time.

Initially, objects in the search pane are placed in a list, and one by one each is matched against scene objects. If a match is found for an object, the search proceeds to the next element of the list. If no match can be found for a given object, the search backtracks and a different match is sought for the previous object. When matching a search pane object against scene objects, MatchTool 2 first verifies that the graphical attributes selected in the search column correspond, and then proceeds to examine relationships expressed by constraints. Only those constraints that reference the search pane object currently being matched, with no references to other unmatched objects, need to be considered at this step. As an optimization, constraints are pre-sorted according to when in the search process they can be tested.

Another technique accelerates the search by using the search pane constraints to isolate where in the scene matches might be found. For example, if the search pane contains two objects with a distance constraint between them, and a match has been found for the first object, then we can determine how far away the second match must be located in the scene (to the accuracy of the constraint tolerance). When matching the second object, we can immediately rule out objects whose bounding box does not intersect this region. Similarly, slope and angle constraints also narrow down the match’s location, but the intersection calculations are more costly, so we currently do not use this information.

When a match is found that the user chooses to replace, the constraints of the objects in the replace pane are copied and applied to the match. This operation is somewhat tricky, since it requires a mapping between objects in the replace pane and the matched objects of the scene. We do this as a two step process: first we map the replace pane objects to those in the search pane, and then we map the search pane objects to the match. The second mapping, between the search pane and the match, is generated automatically by the matching process. The first mapping must be created through other means and is done only once, prior to searching. We have two mechanisms for doing this. When objects are copied from the search pane to the replace pane, they are automatically mapped by the system. This mapping can be overridden or supplemented through auxiliary commands. In addition to copying constraints from the replace pane to the match, MatchTool 2 also copies objects in the replace pane that are not mapped to objects in the search pane. This allows constraint-based replacements to add objects to the scene.

4.12 Implementation

Chimera uses Levenberg-Marquadt iteration to solve systems of constraints [Press88]. This method uses gradient descent when far from a solution, but switches to the inverse Hessian method to converge quadratically when a solution is near. Levenberg-Marquadt is a least-squares method. Each constraint is implemented as an error function, and the algorithm finds the best solution to a set of error functions according to a least-squares evaluation, provided it does not fall into a local minimum. The functions are not limited to be linear, or even algebraic. If the constraint solver cannot find an acceptable solution the user is notified of this, and they then have the option of undoing the operation, or trying to coax the system out of a local minimum by further manipulating the graphical objects. We would eventually like to add multiple constraint solvers, so that when one fails to find a solution, another can be invoked. Chimera can currently enforce the geometric relationships listed in Appendix B, but theoretically constraint-based search and replace could work with nearly any constraint set, provided the system has a means of testing and solving these constraints.

The implementation of MatchTool 2 was greatly facilitated by the use of *generators*, which are objects that return the values of a sequence on an as-needed basis. Both the function and data for producing the next value are stored in the generator. The first MatchTool used a single generator for producing all matching orientations of one shape against another. It worked so well that in MatchTool 2 we use generators pervasively. MatchTool 2 has generators for matching sets of segments within an object, matching a single search pane object, matching the entire search pattern, and matching the activated rules of a ruleset. The abstractions provided by these generators made the program elegant and much easier to write. The code runs reasonably quickly. For example, the search and replace operations in Figure 4.7 take about 0.2 seconds on a Sun SparcStation 1+, including the time spent solving constraints.

4.13 Conclusions and Future Work

The power of graphical search and replace is significantly enhanced by the addition of constraints. Constraints allow specific geometric relations to be sought and enforced, making the rules applicable in many situations not addressed by search and replace on complete shapes. Constraint-based search and replace is a convenient interface for defining and understanding rules that transform drawings, including illustration beautification rules. Other systems have pre-coded beautification rules, so that existing rules can be understood only by reading documentation or using the system, and new rules can be defined only as programming enhancements. Constraint-based search and replace allows new rules to be defined by the user, making an additional kind of editor extensibility possible. Constraint-based search and replace can add new objects to a scene, constrained against existing objects, which extends the applications of the technique beyond simple beautification.

We are pleased with the interface for constraint-based search and replace specifications, though several important user options need to be added. In Peridot the search process is terminated when all of an object's degrees of freedom have been constrained. This is a very useful feature that we would like to add. However, the constraints that we use are multi-directional, and often non-unique (i.e., they may constrain a degree of freedom without uniquely determining it), so it is harder in Chimera to determine when this is the case. For example, in beautifying the house shown in Figure 4.4, our implementation prompts the user to accept or reject replacements, even if they add only *redundant constraints* (i.e. constraints already implied by other constraints in the scene). We are currently investigating methods for determining when a degree of freedom is already constrained.

These methods might also indicate if existing constraints conflict with another that we would like to add, and if so which. If there is a constraint conflict, MatchTool 2 should allow the option of either removing conflicting constraints and applying the new ones, or not performing the replacement at all. Currently we can determine only whether or not the augmented system can be solved by our constraint technology, not whether the system is solvable or which particular constraints conflict. If MatchTool 2 cannot solve the system, it prints a message and the user can either undo the replacement or manually remove the unwanted constraints.

Recent research has dealt with merging rule-based techniques into direct manipulation systems [Hudson91] [Karsenty92]. Constraint-based search and replace is a direct manipulation technique for defining rules that govern the geometry and placement of graphical objects. Since direct manipulation interfaces represent data in terms of such objects, dynamic constraint-based search and replace might be useful for defining rules to control the behavior of these interfaces. For example, certain types of semantic snapping could be defined with this technique. We are interested in modifying constraint-based search and replace to make it useful for this task.

Rules in our system could be enhanced in a number of ways. Currently there is no mechanism for expressing the value of a replacement attribute relative to the match. For example, our system currently cannot search for all angles in a given range, and add 5° to each. In addition, we would like to be able to control the permanence of constraints in the replacement pattern individually, and to assign them individual priorities if necessary. To improve the visual representation of rules we should display the tolerance of the search pane constraints, textually, in their labels. Also, sometimes it is more convenient to type in tolerances textually, and MatchTool 2 should allow this.

Finally, we would like to improve our tools for archiving and merging multiple rule sets, add new kinds of constraints to our system, and allow numerical parameters to the search and replace rules.

> *As to Holmes, I observed that he sat frequently for half an hour on end, with knitted brows and an abstracted air, but he swept the matter away with a wave of his hand when I mentioned it. "Data! data! data!" he cried impatiently. "I can't make bricks without clay."*
>
> — Sir Arthur Conan Doyle,
> "The Adventure of the Copper Beeches"

Chapter 5

Constraints from Multiple Snapshots

5.1 Introduction

The technique described in the last chapter infers the intended presence of many constraints from a single, static illustration, according to user-definable rules. However, single illustrations often do not provide enough data to infer all of the constraints in a scene. Since geometric constraints typically specify how some scene objects reconfigure when other scene objects move, it can help to look at multiple valid configurations (or *snapshots*) of a scene to provide another source of information for inferring constraints implicit in an illustration. This chapter describes a new technique for inferring constraints from multiple valid configurations of a scene.

Geometric constraints are used extensively in computer graphics in the specification of relationships between graphical objects [Sutherland63a] [Borning79] [Myers88] [Olsen90]. They are useful during object construction to position components relative to one another precisely, as well as during subsequent manipulation of the components. Several graphical techniques, such as grids, snap-dragging [Bier86] and automatic beautification [Pavlidis85] were developed to make the initial construction phase easier, since specifying constraints explicitly can be a complex task. However, when objects are to be manipulated frequently, permanent constraints have an advantage over these other techniques in that they need not be reapplied. Permanent constraints can be particularly useful when subsequent editing of a scene is required, in constructing parameterized shapes that can be added to a library, in specifying how the components of a window should change when the window is resized, or in building user-interface widgets by demonstration.

We introduce a technique for inferring geometric constraints from multiple examples, replacing the traditional constraint specification process with another that is often simpler and more intuitive. Initially, the designer draws a configuration in which all constraints are satisfied, and presses a button to take a snapshot. A large number of possible constraints are inferred automatically. Subsequently, if the scene is modified and other snapshots taken, previously inferred constraints are generalized or eliminated so that each snapshot is a valid solution of the constraint system. For example, we can define two objects to be squares, constrained to maintain the same proportional sizes, by taking a snapshot of two squares, scaling them by equal amounts, and taking another snapshot. Then, if the length of one of the square's sides is changed, the lengths of its other sides and the sides of the second square are updated automatically. The designer need not have a mental model of all the constraints that must hold, and can test the results by manipulating the scene objects.

Furthermore, the designer may take snapshots at any time. If after one or more snapshots a set of graphical objects does not transform as expected or if the constraint solver cannot reconcile all inferred constraints simultaneously, the graphical primitives can be manipulated into a new configuration with constraints turned off, and a new snapshot taken. The incorrectly inferred constraint set is automatically modified so that the new snapshot is a valid constraint solution.

There are a number of problems with traditional constraint specification that this new technique attempts to address:

- *Often many constraints must be specified.*

Complex geometric scenes contain many degrees of freedom, and often most of these need to be constrained. It can be tedious to explicitly express large numbers of constraints.

- *Geometric constraints can be difficult to determine and articulate.*

Using constraints requires specialized geometric skills and the ability to articulate about geometric relationships. For example, people asked to define a square often describe it as a rectangle with four equal sides. This definition is incomplete, since it neglects the 90° angle constraint. Yet ask them to draw a square and they typically get it right. Traditional constraint-based drawing systems may not use the appropriate language or abstractions for expressing geometric relationships.

- *Debugging, editing, and refining constraint networks are complex tasks.*

When incorrect or contradictory constraints are specified, the designer needs to debug the constraint network, which can be a cumbersome process. To support the debugging task, a visual representation is usually provided for constraints. WYSIWYG editors need a special mode for displaying constraints, or support for multiple views. When constraints and graphical objects are presented together, the scene becomes cluttered if more than a few constraints are displayed simultaneously.

Many approaches have been taken to solve these limitations. The first problem was addressed by Lee, who built a system to construct a set of constraint equations automatically for a database of geometric shapes [Lee83]. In doing so, he worked with a restricted class of mutually orthogonal constraints, and required that the geometric shapes be aligned with the coordinate axes. Lee's problem domain and assumptions restricted the set of constraints such that there was never any ambiguity about which to select. In our domain the initial ambiguity is unavoidable, and we rely on multiple examples to converge to the desired constraint set.

Systems like Sketchpad [Sutherland63a] [Sutherland63b] and ThingLab [Borning79] make it easier to add large numbers of constraints to a scene, by allowing users to define new *classes* of objects that include the constraints that operate on them as part of the definition. When users create instances of a new class, the system automatically generates the associated constraints. However, people defining a new object class must still instantiate all the constraints to include in their class definition or prototype. Constraints from multiple snapshots can help with this task.

One of the innovations of Myers's Peridot [Myers86] [Myers88] is a component that infers constraints automatically as objects are added to the scene. A rule base determines which relationships are sought, and when a match is found the user is asked to confirm or deny the constraint explicitly. This reduces much of the difficulty inherent in choosing constraints—the designer is prompted with likely choices. Peridot's geometric inferencing component is limited to objects that can be represented geometrically as boxes aligned with the coordinate axes. The previous chapter describes Chimera's constraint-based search and replace mechanism, which also infers geometric constraints from static scenes, but according to user-defined rules. However, a single example often contains insufficient information to infer many desired constraints. This chapter describes another component of Chimera that uses multiple examples to support the constraint inferencing process.

Maulsby's Metamouse [Maulsby89a] induces graphical procedures by example, and infers constraints to be solved at every program step. To make the task more tractable, he considers only touch constraints in the vicinity of an iconic turtle that the user teaches to perform the desired task. These constraints are treated as preconditions and postconditions for learned procedural steps, and not as permanent scene constraints. Complex relationships between scene objects can be expressed through procedural constructions, but the relationships between objects in these constructions tend to be unidirectional, and procedures for every dependency need to be demonstrated.

The difficulty inherent in understanding interactions among multiple constraints and debugging large constraint networks has been addressed by the snap-dragging interaction technique [Bier86] [Bier88] and by an automatic illustration beautifier [Pavlidis85]. In snap-dragging, individual constraint solutions are isolated temporally from one another, so that their interaction cannot confuse the artist. The automatic beautifier infers a set of constraints sufficient to neaten a drawing, but the constraints are solved once and discarded—they are isolated temporally from subsequent user-interaction. In the approach described here, constraints can interfere with one another when a new solution is computed. However, the conflicting constraints can be removed by taking additional snapshots.

A number of systems provide visual representations of constraints to facilitate debugging. Sutherland's Sketchpad connected constrained vertices together with lines accompanied by a symbol indicating the constraint. Nelson's Juno, a two-view graphical editor, provided a program view of constraints [Nelson85]. Peridot communicated constraints as English language fragments during confirmation, and Metamouse used buttons for confirming and prioritizing constraints. The OPUS interface editor represented constraints between interface components as arrows connecting hierarchical frames or drafting lines [Hudson90a]. Our technique never requires that its users work with individual, low-level constraints. In both the specification and debugging stages, they can think entirely in terms of acceptable configurations of the illustration. The inferred constraints can be tested by manipulating scene objects, and the constraint set refined through additional snapshots. For those that prefer a more direct interface for verifying the inferred constraint set, we provide a browser that displays constraints in a Sketchpad-like fashion. This is described in Appendix C. Because our technique is particularly useful in heavily constrained systems, we allow constraints in the browser to be filtered by type or object reference.

One of Borning's ThingLab implementations allowed new types of constraints to be defined and viewed graphically [Borning86]. Several systems permit users to define new classes of constraints by filling in cell equations in a spreadsheet [Lewis90] [Hudson90b] [Myers91a]. The technique introduced here infers constraints from a fixed set of classes that have proven useful for graphical editing. The inference mechanism determines constants in the constraint equations, but it does not synthesize new classes of equations.

Our technique is an application of learning from multiple examples, also known as *empirical learning*. Several empirical learning systems are discussed in [Cohen82]. In contrast, generalizing from a single example is called *explanation-based learning* and is surveyed in [Ellman89]. Explanation-based learning requires a potentially large amount of domain knowledge to determine why one explanation is particularly likely. As we illustrate in subsequent examples, there are often few or no contextual clues in a static picture indicating that one set of constraints is more likely than the next, so we felt the empirical approach was warranted. Empirical learning algorithms have been extensively studied by the AI community, but we developed our own to take advantage of certain features of the problem domain and to make learning from multiple examples a feasible approach to geometric constraint specification.

The technique described in this chapter, like the others in this thesis, has been incorporated in the graphics and interface editing modes of Chimera. Our initial experience suggests that the snapshot approach, like declarative constraint specification, has its own set of strengths and weaknesses. These will be discussed later in the chapter.

In Section 5.2, we illustrate the user's view of constraint specification with a number of examples. We provide a detailed description of our algorithm in Section 5.3. In Section 5.4 we discuss implementation details. Finally we mention limitations of the approach, present our conclusions, and discuss future work in Section 5.5.

5.2 Examples

In this section we show three examples of how constraints are inferred from multiple examples within the graphics and interface editing modes of Chimera. To facilitate the initial construction of the scene, Chimera provides both grids and snap-dragging alignment lines. Chimera has fixed square grids that can be turned off if they interfere with the drawing process. Alignment lines facilitate establishing geometric relationships that cannot be expressed with these grids.

5.2.1 Rhombus and Line

Suppose that we would like to add permanent constraints to the rhombus in Figure 5.1a, so that during subsequent graphical editing it will remain a rhombus, its horizontally aligned vertices will be fixed in space, and the nearby line will remain horizontal, of fixed length,

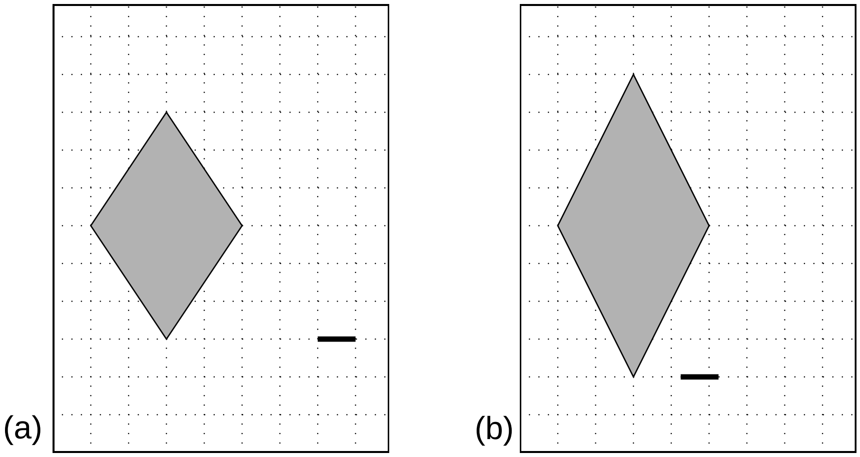


Figure 5.1 Two snapshots of a rhombus and line.

to the right and at the same *Y* position as the bottom vertex of the rhombus. After the initial scene is constructed in Figure 5.1a, the user presses the *snapshot* button in the editor's control panel. Next, the user translates the top and bottom vertices of the rhombus to make it taller, and translates the horizontal line to the same *Y* coordinate as the rhombus's bottom vertex, but to a different *X* so that its position will not be absolutely constrained in *X* with respect to the bottom vertex. The user presses the snapshot button once more. Figure 5.1b shows the second snapshot.

Initially constraints were turned off. Now when the user turns them on from the control panel and edits the scene, the constraints inferred from the snapshot are maintained by the editor. In Figure 5.2a, the user has selected the horizontal line and moved it upwards. The top and bottom vertices of the rhombus automatically move so that the demonstrated constraints are maintained. When, in Figure 5.2b, the top joint of the rhombus is selected and translated to a higher grid location, the bottom rhombus vertex and the horizontal line both move appropriately.

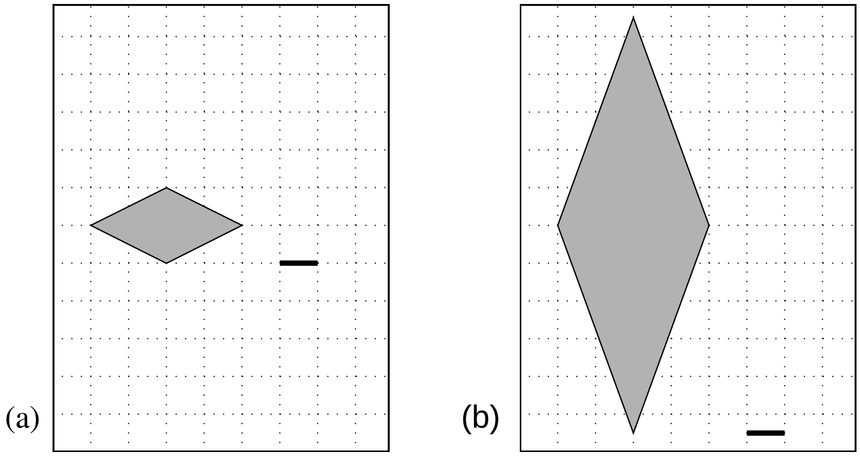


Figure 5.2 Two constrained solutions to the snapshots in Figure 5.1.

The abovementioned constraints were all specified implicitly, without the user having to express intent in low-level geometric terms. Inferring this information from a single example would be problematic, since it is not clear how to distinguish between those parameters that should be fixed (such as two of the rhombus's vertices, and the length and slope of the horizontal line) and those that should be allowed to vary (such as the length of the rhombus's sides, and the locations of the horizontal line's vertices).

One might expect that people need a sophisticated understanding of the inferencing mechanism to provide the right set of snapshots, but this is untrue. In the second snapshot, the user foresaw the need to move the horizontal line in *X*, relative to the bottom vertex of the rhombus, to allow it to move this way during subsequent interaction. However if the user neglected to think of this, the constraint solver would disallow such configurations during later manipulations of the scene. The user could then turn off constraints, and provide as an extra example the configuration he or she tried to achieve, but could not. This third snapshot would automatically remove constraints that originally prohibited this configuration, without the user explicitly naming them.

In part because this is a highly constrained illustration, few editor operations were necessary to establish the required constraints using snapshots. In traditional constraint specification, the user starts with a clean slate, and adds all of the intended constraints to the illustration. The snapshots technique takes a very different approach. It initially assumes that all constraints are present in the initial snapshot, and with additional snapshots the user prunes away undesired constraints. This approach is subtractive rather than additive, and it works best for heavily constrained scenes in which few constraints must be removed. In contrast, traditional declarative specification typically becomes more difficult as more constraints must be added to a scene. The two techniques complement one another. When only a few constraints must be instantiated, it is typically easier to use the traditional declarative approach. Having both forms of specification available allows each technique to be used in cases where it works best, so Chimera's interface supports both.

5.2.2 Resizing a Window

Constraints are useful in constructing user-interfaces because they allow the attributes of one interface object to be defined in terms of the attributes of others. For example, when a window is resized, the position and size of the contents may change. Figure 5.3a shows a window that we have constructed in Chimera's interface editing facility, containing an application canvas (the darkly-shaded rectangle), a scrollbar, and three buttons that invoke menus. After positioning these widgets within the parent window, the user presses the snapshot button. The components of the window are then shifted into another configuration, shown in Figure 5.3b, and a second snapshot is taken. Precise positioning in these snapshots was achieved by using a combination of grids and snap-dragging.

The user intends that the buttons be a fixed distance above the bottom of the window, that the left side of the Basics button be a constant distance from the window's left, that the right side of the File Ops button be a constant distance from the window's right, and that the Transformations button be evenly spaced between the inner sides of the two other buttons. The scrollbar's dimensions are intended to be fixed by the top and right sides of the window, the top of the buttons, and by its constant width. The application canvas should be fixed relative to the left and top of the window, the top of the buttons, and the left side of the scrollbar. Now, when we turn on constraints and select the upper right corner of the window (while the lower left corner is fixed), the window and its contents reshape as shown in Figure 5.3c.

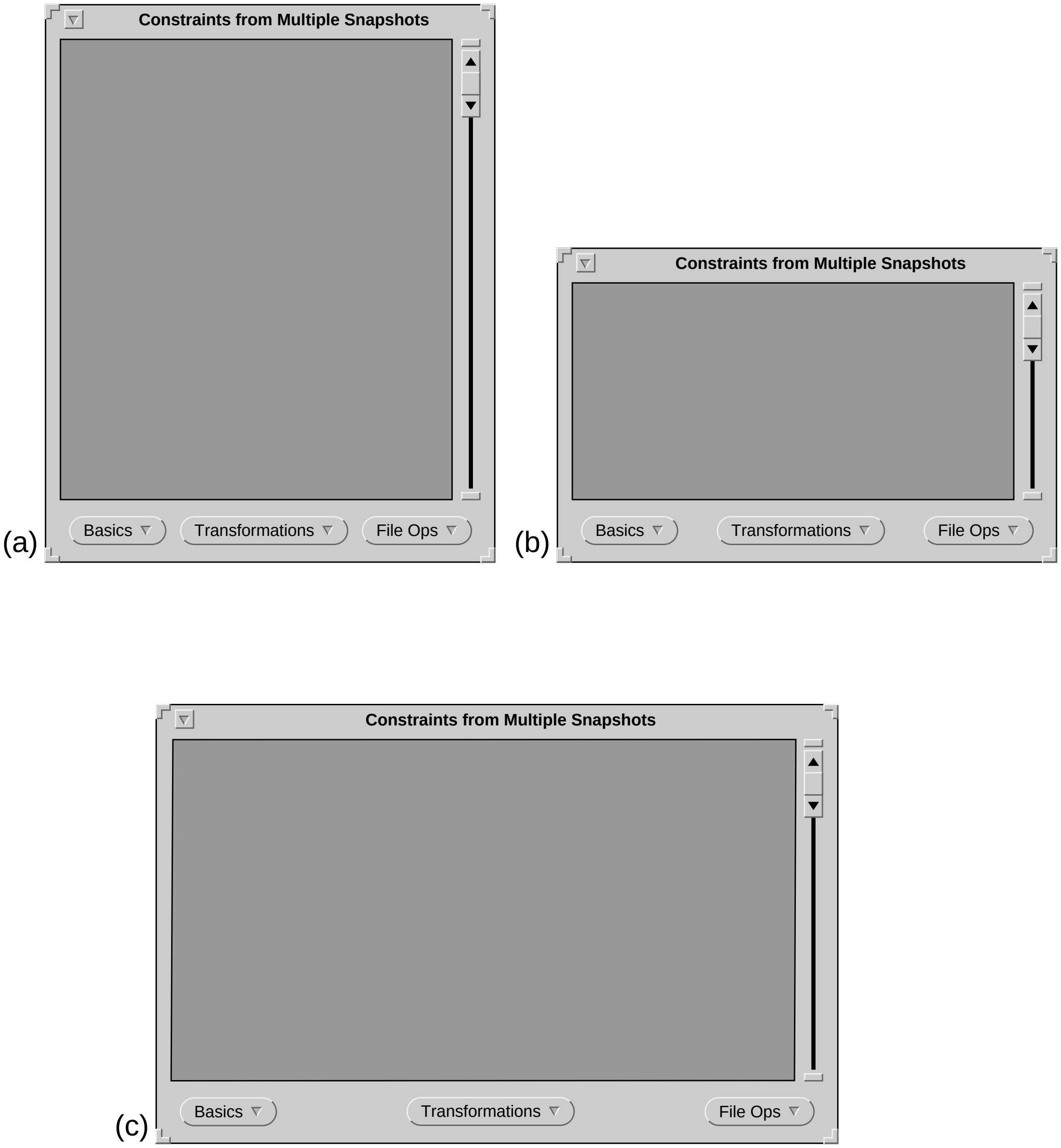


Figure 5.3 Specifying window resizing constraints. (a) and (b) are the two snapshots, (c) was produced by dragging the upper right corner of the window.

5.2.3 Constraining a Luxo™ Lamp

This final example applies to both graphical editing and user-interface construction. We would like to constrain a 2D illustration of a Luxo lamp, so that it behaves like a Luxo lamp. In particular, we want the various pieces to remain connected, the base to be fixed at its initial location, and the arms of the lamp to remain a constant length. Other constraints are important as well, but instead of determining which are significant ourselves, we would prefer to edit the lamp into a number of valid configurations and take snapshots. To control the direction of the lamp's beam, we have built a simple dial widget out of a circle and line, and we specify the behavior of the dial relative to the Luxo lamp by demonstration as well. Figures 5.4a and 5.4b show the initial two snapshots of valid configurations of our illustration. Note that the constraints inferred from these two snapshots are independent of the particular editing operations chosen, as explained in Section 5.3.

After taking the first two snapshots, we turn on constraints and try to manipulate the Luxo lamp, but the constraint solver indicates that it cannot solve the system. The source of the problem is an *incidental constraint,* that is, a constraint that was evident in the first two illustrations, yet was not an intended relationship. When incidental constraints interfere with a desired configuration they can be removed by manipulating the scene into the new configuration with constraints turned off, and taking an additional snapshot. We could determine which incidental constraint(s) occur in the scene by cycling through the visual representation of all constraints, and then explicitly delete the undesired ones. However this can be time consuming when a scene contains many constraints, and it requires that the end user understand the constraint composition of the scene. Fortunately, there is never a need to specifically identify and cull unwanted relationships. While manipulating the scene, if users find that unwanted constraints prohibit a desired, valid configuration, they can turn off constraints, set up this configuration by hand, and take another snapshot. This additional snapshot removes all constraints prohibiting the new configuration. People need not be clever about conveying only the desired constraints in the first two snapshots. Refining a constraint set using snapshots, as with declarative specification, can be an incremental process.

Without identifying the incidental constraint, we set up a configuration that this constraint forbids. This additional snapshot appears in Figure 5.4c. Now the various components of the lamp move as we had intended. In Figure 5.5, we manipulate the lamp into three configurations by moving its top joint and adjusting the beam direction dial. Note that these two controls are not independent—when the dial is rotated, the arms of the lamp can move during the solution of the constraint equations, since it does not uniquely determine a lamp configuration. We can temporarily place a declarative constraint on the top joint if we want to change only the beam direction while keeping the arm fixed.

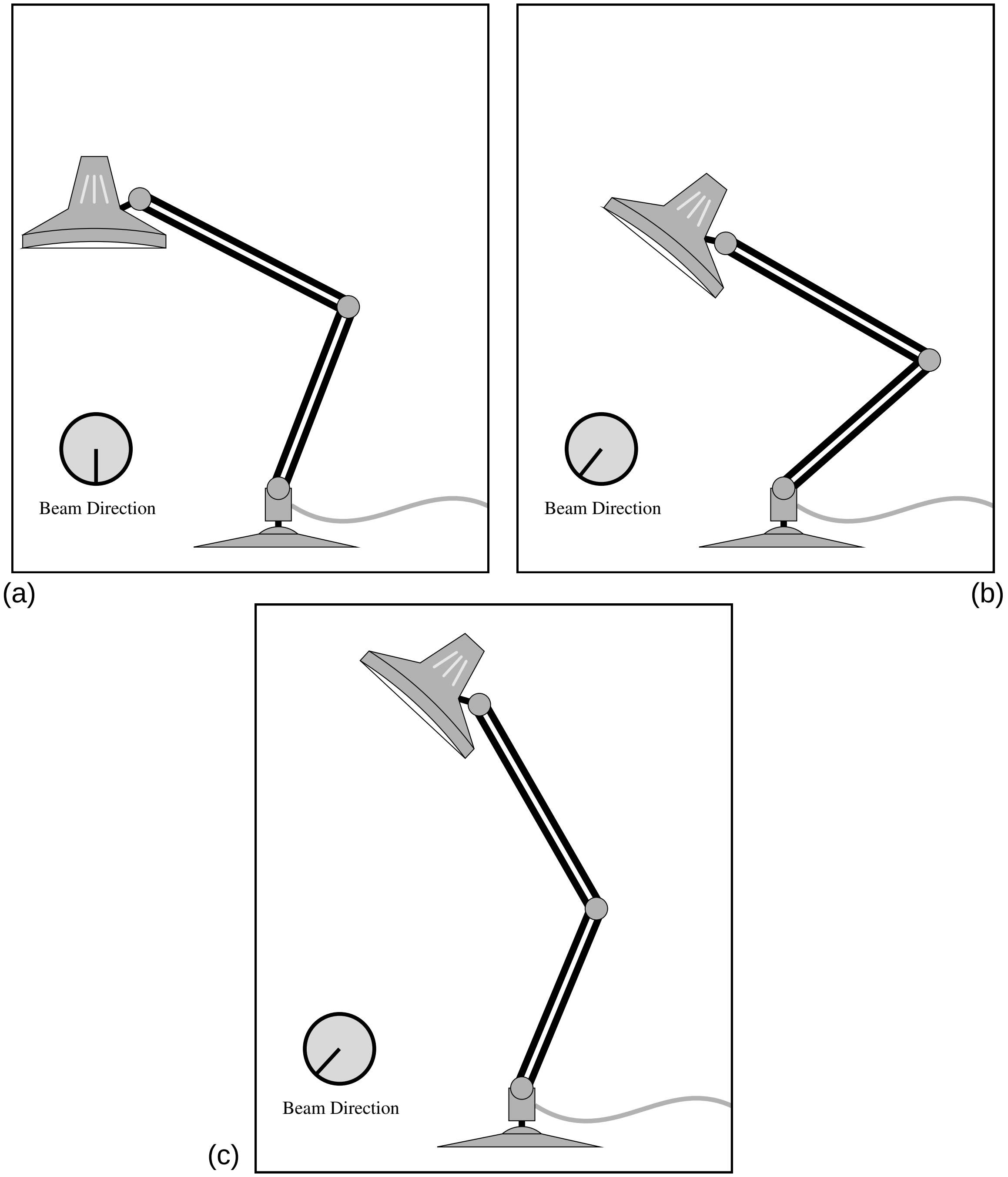


Figure 5.4 Teaching Luxo constraints. Three snapshots of valid configurations, provided as input.

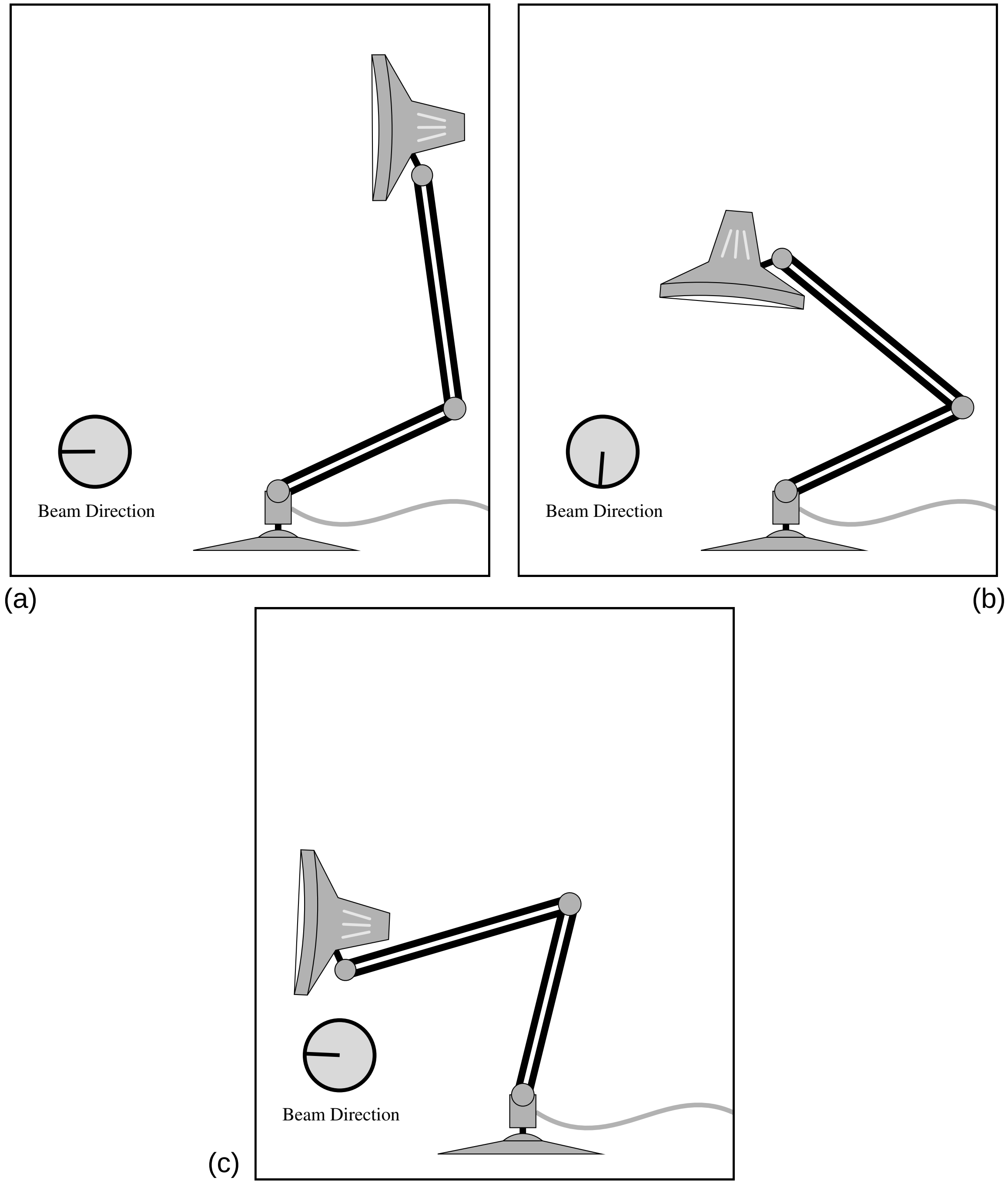


Figure 5.5 Luxo on his own. Configurations created by manipulating the uppermost joint and Beam Direction dial.

5.3 Algorithm

In this section we discuss the set of constraints that our system infers. Then we present an efficient algorithm for inferring these constraints, and demonstrate the algorithm on a simple example. Next we analyze the algorithm, and discuss how parameters can be inferred.

5.3.1 The Constraint Set

All objects in the Chimera editor are defined geometrically in terms of vertices, and constraints fix the relationships between these vertices. Based upon a finite set of example scenes, an infinite number of arbitrary constraints can be inferred. Hence we have chosen to infer a fixed set of geometric relationships that have proven particularly useful in graphical editors. Appendix B describes the set of constraints that Chimera can infer from multiple snapshots, and this is the same set that can be specified through Chimera's declarative interface described in Appendix C. Since understanding this constraint set is important to understanding the algorithm, these constraints are repeated below:

Absolute Constraints	**Relative Constraints**
Vertex location	Coincident vertices
Distance between two vertices	Relative distance between two pairs of vertices
Distance between parallel lines	Relative distance between two pairs of parallel lines
Slope between two vertices	Relative slope between two pairs of two vertices
Angle defined by three vertices	Equal angles defined by two sets of three vertices

The algorithm described here infers both absolute and relative geometric constraints. *Absolute constraints* fix geometric relations to constant values. *Relative constraints* associate geometric relations with one another. For example, an absolute constraint might fix a vertex to be at a particular location, or a distance to be a constant scalar. A relative constraint might fix two distances or slopes to be the same. Each relative constraint on the right corresponds to an absolute constraint on the left.

The algorithm discussed in this section finds all of the constraints of the form listed above that hold over a sequence of snapshots. The relative slope constraint sets one slope to be a constant offset of another, when measured in degrees not y/x ratio. As pointed out in Appendix B, two of these constraints subsume two others. The absolute distance constraint between vertices subsumes the coincident vertices constraint, so the constraint inferencing component needs no special support for the latter. The relative slope constraint subsumes the absolute angle constraint, but the algorithm still monitors absolute angle relationships, because they lead to the identification of relative angle relationships.

Many higher-level constraints can be formed by the composition of these lower-level constraints, and thus are also inferred by the algorithm. For example, the constraint that one box be centered within another is captured by two relative distance constraints between parallel lines.

5.3.2 Algorithm Description

In the rest of this section, we describe the algorithm that infers these constraints. An overview of its steps is given in Figure 5.6. It may be helpful to refer back to this figure during the subsequent discussion.

```
IF first snapshot THEN
    add vertices to initial transformational group
ELSE BEGIN
    split transformational groups to form new child groups
    identify intra-group constraints of child transformational groups
    identify inter-group group-to-group constraints due to splitting transformational
      groups
    identify inter-group vertex-to-vertex constraints
    break previously instantiated constraints that have been violated
    form delta-value groups from broken absolute constraints
    make a copy of the constraints with redundancies filtered out for the solver
END;
```

Figure 5.6 Steps of the inferencing algorithm

With the first snapshot, the scene is entirely constrained, and each subsequent snapshot acts to reduce or generalize the constraints on the system. If we were to represent explicitly each constraint that could hold at any one time, the space and time costs would be prohibitive. Instead, we economically represent similar constraints over sets of vertices as *groups*. For example, after the initial snapshot, all vertices are constrained to a set location, and the distance and slope between each pair of vertices is fixed, as is the angle between each set of three vertices. Although we could instantiate each of these constraints explicitly, it is far more efficient to represent the vertices as a group with a tag indicating the relationships that hold among all of its members. As will be discussed later, groups can also accelerate the process of determining which constraints hold over a series of snapshots, and can ultimately reduce the number of constraint equations that are passed to the solver.

5.3.2.1 Transformational groups

The most important type of group in our inferencing mechanism is the *transformational group*. A transformational group contains a set of vertices that has always been transformed together since the first snapshot. At the first snapshot, the algorithm places all the vertices into a fixed vertex location transformational group, since their positions are initially constrained to be fixed. As vertices are transformed, our undo mechanism keeps track of the sets of vertices selected and the transformation applied, and this information is used by the inferencing mechanism to fragment existing transformational groups into smaller ones. The transformations that can be applied in our system currently include translations, rotations, and isotropic scales, although we plan to extend this algorithm to work with any affine transformation.

5.3.2.2 Intra-group constraints

We can very efficiently determine *intra-group constraints*, that is, constraints that hold within a given transformational group. Figure 5.7, shows various affine transformations and the geometric relationships that they preserve.[1] The transformation listed in the top half of each box maintains the relationships listed in the lower half of the box, and those relationships in the boxes above it. For example, if a transformational group has only been scaled and translated, the slopes, angles, ratios of distances, and parallel relationships are all maintained. By tracking the transformations that have been applied to a transformational group, we determine which constraints must hold within the group without examining its individual vertices.

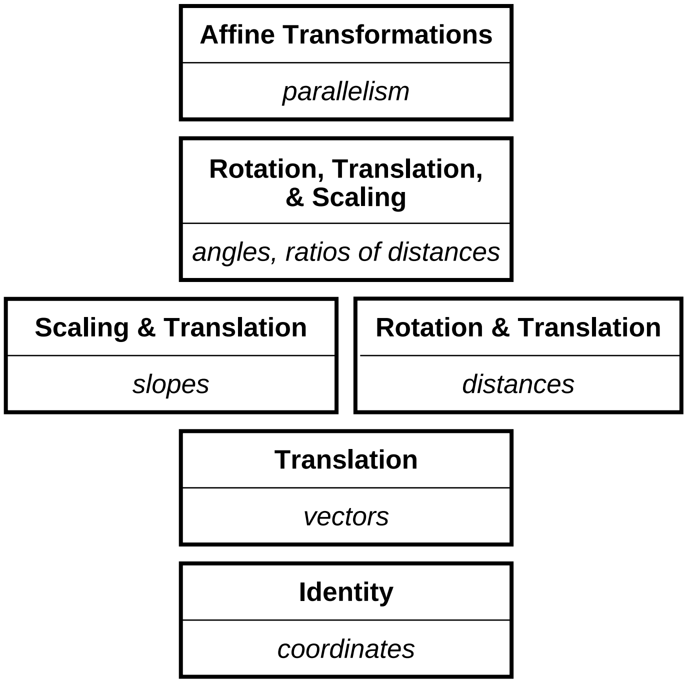


Figure 5.7 Transformations and the geometric relationships that they maintain. Reprinted with permission from [Bier86].

---

[1] Note that scale in this diagram refers to isotropic scale, and the vector relationship is the combination of slope and distance constraints.

We next determine which constraints *cannot* hold within the transformational group. Again, this is easily done by examining the transformations that have been applied to the group. If a group has been translated, all of its fixed vertex location constraints are broken. Fixed vertex location constraints are also broken among vertices during rotations and scales if the vertices are not at the center of the transformation. Scales break all absolute distance relationships within a transformational group, and rotations break all absolute slope relationships within a transformational group.

After determining which relationships *must* hold within a group, and which *cannot* hold, we must consider the relationships that *might* hold. For each of these constraint relationships, we must examine the vertices in the group, looking for invariant relationships. Fortunately this expensive task need not be done for the most common transformations, translations, rotations, and isotropic scales, since all relationships in our constraint set can immediately be classified as either definitely present or definitely not present. If we were to extend this algorithm to other less common affine transformations, then the vertices would need to be examined explicitly.

For every snapshot after the initial one, the first step fragments existing transformational groups into new ones, accounting for the transformations that have occurred since the last snapshot. Each new child group has all the constraints of its parent, except those broken by the transformations performed since the last snapshot. Since we are only interested in effective transformations at the snapshot granularity level, we factor the composition of transformations applied since the last snapshot into scale, rotation, and translation components, and use these, as described above, in determining which intra-group constraints were broken. This allows us to ignore transformations that have been undone by subsequent operations between the two snapshots. For example, if a set of vertices is translated away from its original location, and then back again between snapshots, then those translations are effectively ignored.

To illustrate transformational groups, and several other algorithmic details discussed later, consider the two simple snapshots given in Figure 5.8. Two boxes were captured in the first snapshot (Figure 5.8a). Initially, all vertices were in the same transformational group, and constrained to have fixed locations. After this, but prior to the next snapshot (Figure 5.8b) the boxes were both scaled by a factor of 2, and the right box was translated to the left, one large grid unit from the left box. Taking the second snapshot caused the system to split the original transformational group into two children, each containing the vertices of one of the boxes. The second snapshot broke the fixed location constraints for all vertices except the bottom left of the left rectangle, since this vertex's effective transformation had no translational component, and its location was at the center of the scale. The intra-group absolute distance constraints were broken for each group because there was a net scale, but isotropic scales maintain proportional distances, so an implicit relative distance constraint was added to each group. It is important to note that transformational groups are dependent upon the transformations performed, but they have no impact on the constraint

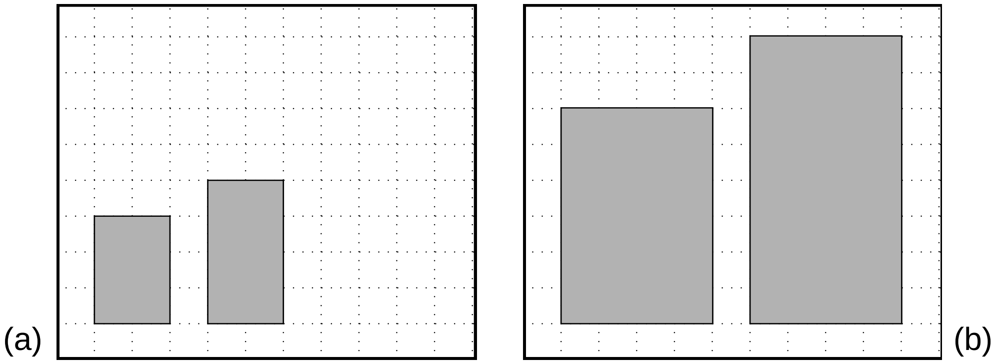


Figure 5.8 Two snapshots of a simple scene.

set that will eventually be inferred. They accelerate the search process by pruning the search space.

5.3.2.3 Inter-group constraints

The next step is to compute *inter-group constraints*—constraints between different transformational groups or their vertices. These constraints are generated in several ways. They can be formed from a relative intra-group constraint when a transformational groups is split by a transformation that preserves the relation. Consider a transformational group with a relative slope constraint among all of its vertices. If the group is split in two by a translation or scale, then we must add a relative slope constraint between the two groups, relating the slopes contained within one group to the slopes within the other. Similarly, if a transformational group has a relative distance constraint among all of its vertices, and the group is split by a translation or rotation, then we need to add a relative distance constraint between the two groups, specifying that the distances within one group will remain proportional to the distances in the other.

We have just described *group-to-group* inter-group constraints—constraints that make entire groups rotate or scale with another. There are also *vertex-to-vertex* inter-group constraints, which express relationships between a small number of vertices. Finding these is the most costly step in our algorithm, but the cost is reduced by the observation that we only need to compute inter-group constraints between a child transformational group, its parent, and its siblings (other child groups of the same parent spawned during the same snapshot). Inter-group constraints between the child and other groups were already formed when their ancestors were split.

For small sets of vertices containing members both from the newly created child group and from its parent or siblings, we look for relationships that have not changed and generate absolute constraints for these when found. For example, we compute the slope and distance between such pairs of vertices at the current snapshot, and the previous snapshot.

If either of these values are unchanged, we create an absolute constraint between the two vertices. Similarly, for each pair of lines constrained to have the same slope, that were contained in a single transformational group during the last snapshot, but are now split among groups, we identify absolute parallel distance constraints.

In our rectangle example, an inter-group vertex-to-vertex constraint inferred at the second snapshot declares that the inner segments be one large grid unit apart. This constraint was implicit after the first snapshot, when both rectangles were members of the same transformational group, but must be made explicit after the second snapshot since the relationship still holds after the transformational group was split.

5.3.2.4 Delta-value groups

Existing constraints between groups or vertices transformed since the last snapshot are now considered, and those that no longer hold are broken. Broken *relative* constraints, constraints relating geometric measures (such as slope) of more than one object, are split if possible into constraints that are still satisfied among fewer objects. Absolute constraints that have been broken during the current snapshot are matched, as is now described, to form new relative constraints.

We have already described how absolute inter-group constraints are found by locating relationships that do not change. One type of relative inter-group constraint is found by locating relationships that change together. If two pairs of vertices are constrained to have constant slopes, then there is no need for a relative constraint between the two, since the individual values are fixed. However, if these slopes now change by the same amount, it becomes necessary to create a relative constraint between them. Collections of relations that were absolutely constrained in a previous snapshot, but have broken by equal amounts in the current snapshot, are bundled together into *delta-value groups*.

Delta-value groups, like transformational groups, allow us to represent similar constraints among many objects compactly, but otherwise they are unrelated. Delta-value groups are simply relative constraints between arbitrary numbers of relations. For example, a delta-value group might constrain n distances to be proportional. However, when passing the delta-value group to the solver, it need only be expanded to $n-1$ binary constraints when solving the system (relating the first element to each subsequent element) rather than n^2 constraints relating each pair of elements.

Every absolute constraint broken in the current snapshot must be considered for inclusion in a delta-value group. There are three steps in our algorithm where broken absolute constraints are identified:

- During the fragmentation of transformational groups
- During the identification of vertex-to-vertex inter-group constraints
- During the breaking of constraints instantiated during previous snapshots

We place together in delta-value groups distance relations that change by the same proportion, and angle and slope relations that change by the same number of degrees. Since typically many absolute constraints break during the same snapshot, it is important to find matches efficiently. We employ hashing to match constraints that break by similar amounts, so this step is performed in linear time with respect to the number of broken constraints identified.

Returning to the example of Figure 5.8, the two rectangles are in separate transformational groups after the second snapshot. This snapshot broke absolute distance constraints for both of these groups, since they were scaled differently than their parent, which had an implicit absolute distance constraint among all of its vertices. Both of these absolute distance constraints broke by a factor of two. As a result, they were added to the same delta-value group, maintaining that distances in the two groups be proportional.

5.3.2.5 Redundant constraints

We have now computed all of the constraints that are invariant among snapshots. When objects are transformed with constraints turned on, the inferred constraints are passed to our solver. Typically our constraint set contains a large number of redundant constraints—constraints derivable from others through geometric tautologies. The algorithm finds all constraints from our set that occur in the snapshots, not just the minimal set, though some of the constraints are represented implicitly and efficiently in groups. To accelerate the process of finding a solution to the constraint set, we try to remove redundant constraints. There are two ways that this can be done, both of which involve looking for simple geometric relationships. The first looks for these relationships as a post-process after the constraint inferencing has been performed, and filters out extra constraints known to hold in those circumstances. This is the only method currently implemented in Chimera for filtering redundant constraints, and it works well for those relationships that generate a constant number of redundant constraints. However, certain relationships yield a polynomial number of such constraints, and it would more efficient never to generate them.

These redundancies could be avoided by building additional kinds of groups during the constraint inferencing process. As discussed earlier, transformational groups and delta-value groups allow large numbers of graphical relationships to be represented tersely. By identifying relationships that lead to redundant constraints, and classifying them as special groups, we could pass only the essential constraints to the solver. Figure 5.9 illustrates two relationships that would be particularly useful to express as groups since they are common and yield many redundant constraints if fully expanded. In Figure 5.9a, vertices p and q are constrained to be coincident. If each other vertex r_i in the figure were part of a separate transformational group, our algorithm would instantiate the constraints distance(p, r_i) = distance(q, r_i), and slope(p, r_i) = slope(q, r_i) for all r_i. These redundant constraints could be avoided by building *coincident vertex groups* for sets of vertices currently constrained to be coincident. These groups could be used in lieu of their actual vertices while computing inter-group vertex-to-vertex constraints. If vertices in the group are not coincident in a

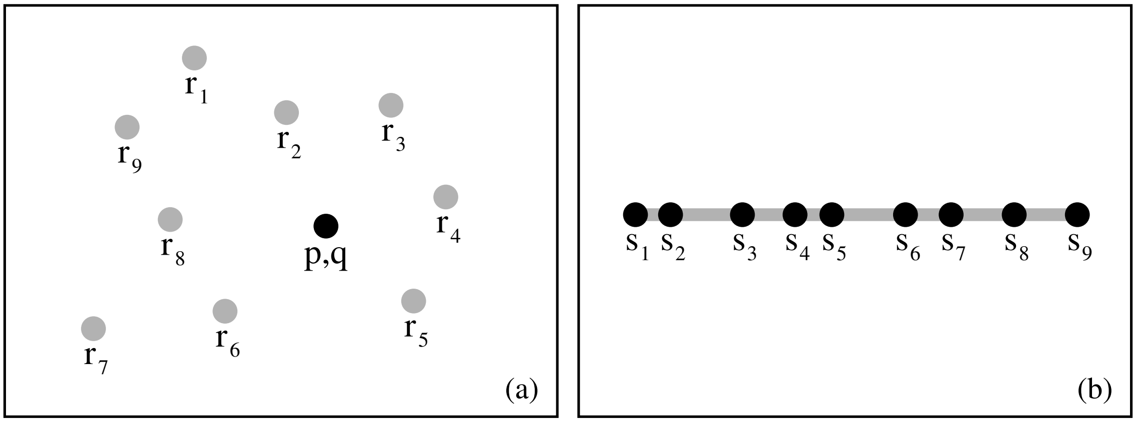


Figure 5.9 Two geometric relationships that lead to redundant constraints.

subsequent snapshot, the group would be broken, and the formerly redundant constraints that still hold would then be instantiated.

Another common geometric relationship leading to redundant constraints is shown in Figure 5.9b. Here, snapshots have resulted in a set of collinear vertices s_i, such that each vertex is in a separate transformational group, and the slope between each pair of vertices is fixed. Here, only n-1 constraints are necessary to represent the slope constraints between the n vertices, but the algorithm identifies constant slope constraints between each pair of vertices, s_i and s_j such that $i < j$. By identifying this relationship as a group during the inferencing process, we could avoid generating these redundant constraints.

Currently we look for only a few classes of redundant constraints, which we filter as a postprocess, and often a large number eludes us. We are working on improving this component of our system.

5.3.2.6 Solving the constraint system

When constraints are turned on and constrained objects transformed, we compute the effects on other objects in the scene. The constraint system can be viewed as a graph, with the nodes being vertices of scene objects, and the arcs being constraints between the vertices. Changes to one disconnected subgraph cannot affect another, since there are no constraints linking them. We find the disconnected subgraphs containing the vertices actively being transformed, by performing a simple graph traversal beginning at these vertices. Constraints which are not a part of any of these subgraphs cannot affect our solution and can be safely ignored. Also, since the constraints of different subgraphs are mutually independent, they are solved independently, thereby reducing the cost of the solution.

We also reduce the solution cost by using a simple generalization of the technique many constraint-based systems use to solve for rigid bodies efficiently. All vertices of the same transformational group are constrained to transform together under a restricted class of transformations, and often we can use this information to avoid passing certain constraints and vertices to the solver. A transformational group that has only been translated has absolute slope and distance constraints between each pair of vertices, and these same constraints insure that all vertices in the group will translate together. If some vertices in the transformational group participate only in these constraints, then instead of passing them to the solver, we can explicitly apply to these vertices the translation that the solver finds for other vertices in the group. Similar approaches can be taken for isotropic scales, rotations, and compositions of these transformation classes.

As an example of this, consider Figure 5.10. The snapshots in Figure 5.10a and 5.10b constrain the hand to scale so that the lower left vertex of the wrist is fixed, and the right-

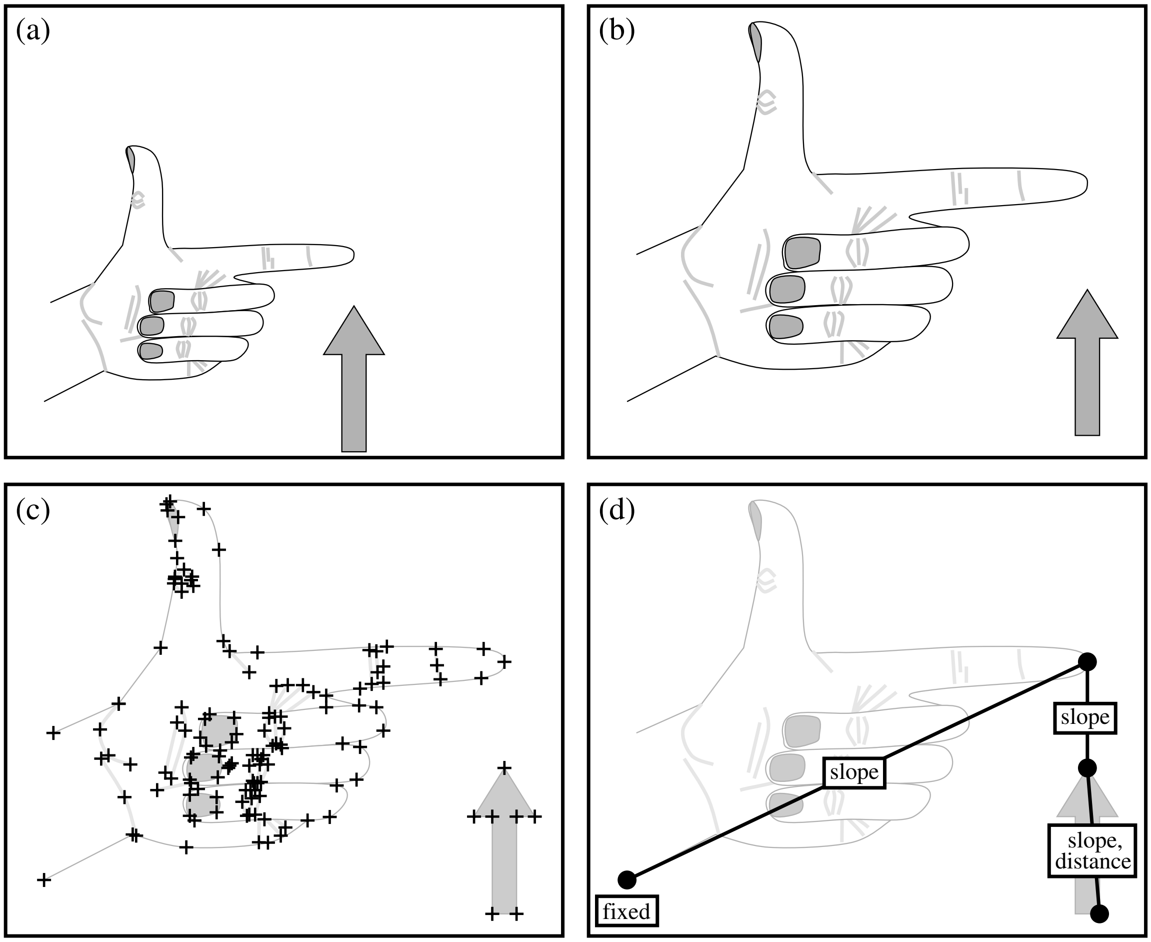


Figure 5.10 Efficient constraint formulations for transformational groups. Snapshots (a) and (b) constrain the scene. A naive approach solves constraints for all vertices marked in (c). A more efficient method solves only constraints shown in (d).

most vertex of the index finger aligns with the arrow. All vertices of the hand are part of the same scale transformational group, and those of the arrow are part of the same translation transformational group. Figure 5.10c shows all the vertices in the system that participate in the constraint solution. However, only a few of these 134 vertices must be passed to the solver.

In Figure 5.10d, we choose to translate the lower right vertex of the arrow. The system begins traversing the constraint graph at this vertex to determine which constraints and vertices must be passed to the solver. This vertex will be passed to the solver, since it is being manipulated directly by the user. The top vertex of the arrow must also be passed to the solver, since it participates in an inter-group slope constraint. These two vertices are bound together by absolute slope and distance constraints because the arrow is a single translation transformational group. The displacement of all the other vertices in this group will be determined by calculating the displacement vector that the solver finds for these points.

Similarly, the vertex at the tip of the index finger is passed to the solver, since it participates in an inter-group slope constraint with the point of the arrow. The lower left vertex of the wrist must also be passed to the solver, since it has a fixed location constraint. These two vertices of the hand are connected by an absolute slope constraint, because they are part of the same scale transformational group. The positions of all the other vertices in the hand are easily determined by the scale transformation that maps these two vertices to their new positions.

5.3.3 Analyzing the Algorithm

This section summarizes the inferencing algorithm, and presents informal arguments for its correctness. The technique described in this chapter finds all relationships of the classes listed in Section 5.3.1 that are present in a sequence of snapshots, and instantiates these into constraints. A brute force algorithm would consider each relationship applied in turn to every collection of vertices of the appropriate size, and then determine whether the relationship in fact changes over the course of the snapshots. This would be computationally expensive, so our algorithm takes a different approach. To show that it works though, it suffices to explain how it finds the same constraint set as the brute force algorithm.

The snapshot process partitions all vertices in the scene into transformational groups. The set of translations, rotations, and isotropic scales applied to a transformational group since the first snapshot automatically determines which of the relationships in our constraint set hold among its vertices, and which do not. This provides the first savings over the brute-force algorithm—it is not necessary to search through collections of vertices in the same transformational group for invariant relationships, since these relationships are completely determined by the group’s transformations. However our algorithm, like the brute-force algorithm, must consider constraints that span multiple transformational groups (inter-group constraints) as well as these constraints that lie in a single group (intra-group constraints). Together, inter-group and intra-group constraints comprise all possible constraints in a scene.

Finding inter-group constraints is more difficult. The constraints of interest to us include absolute constraints and relative constraints. Absolute constraints express a single geometric relationship to be constant, while relative constraints compare multiple geometric relationships. To find *absolute* inter-group constraints, our algorithm does the same thing as the brute-force approach—it considers all collections of vertices of the appropriate number, spanning multiple groups, and looks for absolute relationships unchanged over all the snapshots. It does this incrementally, as transformational groups are split from their parents, but the effect is the same as seeking these relationships after all the snapshots are given.

Performing an exhaustive search for *relative* inter-group constraints would be more costly, since they typically involve larger numbers of vertices than absolute constraints. Fortunately they can be found without resorting to the brute-force approach. Relative relationships are merely pairs of absolute relationships that always change the same way, for example, two distances that always remain proportional. There is no need to create a relative constraint before its absolute components change for the first time, since the absolute components already capture the relative relationships. For example, if two distances are fixed absolutely, then their proportion is implicitly defined. All the absolute constraints in our set capture their corresponding relative relationship until a snapshot breaks the absolute constraints. Then, since relative relationships are pairs of absolute relationships that always change together and in the same way, the algorithm finds relative inter-group constraints merely by identifying absolute constraints that always change identically. So instead of performing a costly exhaustive search for relative inter-group constraints, this algorithm monitors all absolute constraints, both absolute inter-group constraints and absolute constraints on entire groups, and matches those that always change by equal factors (in the case of distances) or degrees (in the case of angles and slopes). In this way, the algorithm finds all inter and intra-group relationships in the scene, and finds an equivalent set to the brute-force approach with less computation.

The algorithm discussed so far is *not* heuristic; it finds all constraints of the classes described in Appendix B that are obeyed by a sequence of snapshots. As will be discussed later, considering all these relationships often results in a number of incidental constraints—relationships in a snapshot sequence that the user did not intend. To combat this problem, we also experimented with a simple modification of the algorithm that instantiates relative inter-group slope and distance constraints only between two pairs of connected vertices, and absolute and relative inter-group angle constraints only on angles formed by three connected vertices. This removes from consideration some relationships that are usually not significant, yet often yield a large number of incidental constraints.

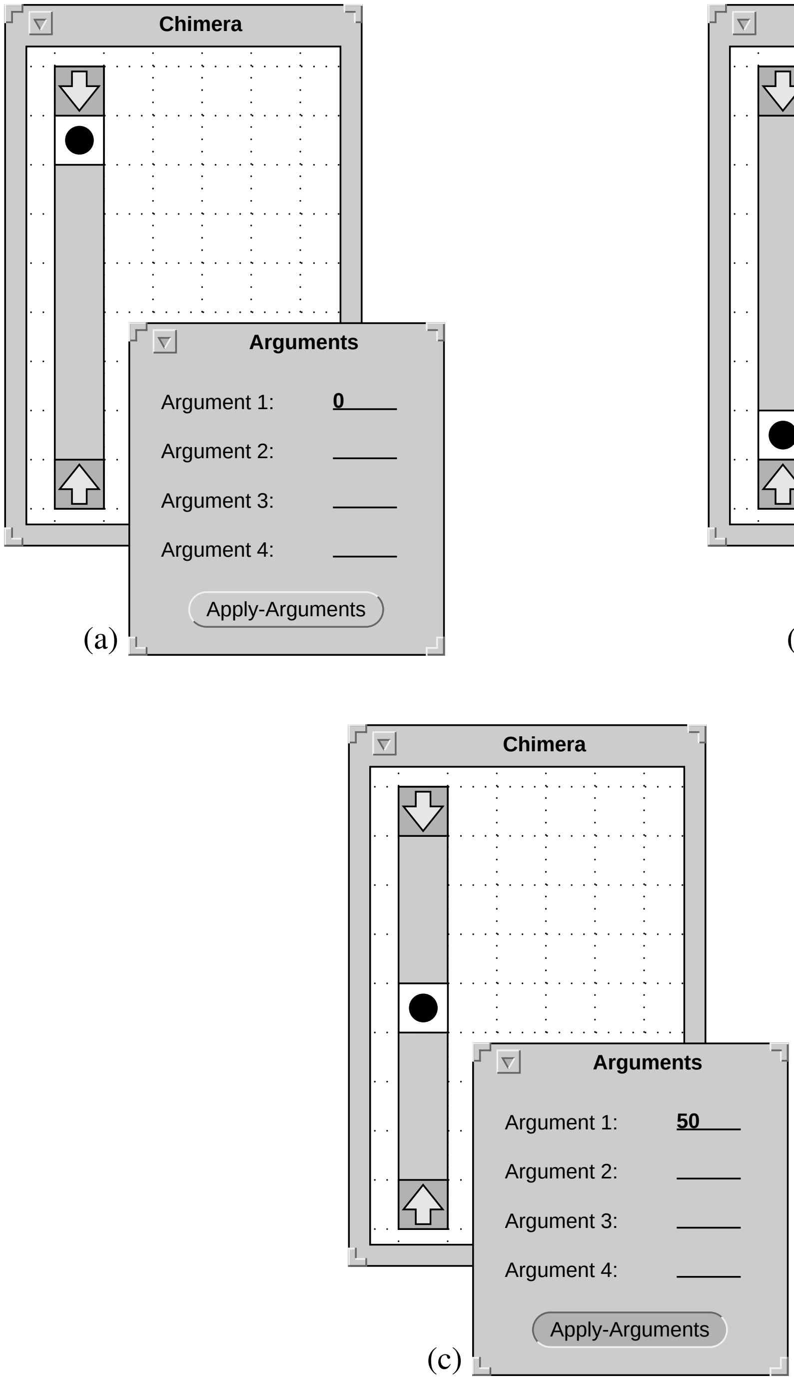
The complexity of the inferencing algorithm depends on whether we restrict the above constraints to connected vertices. If not, the most expensive step is finding inter-group absolute angle constraints, which is O(n^3) with the number of vertices. Since vertices in our system connect no more than two lines, the task of searching for these constraints between *connected* vertices is O(n). But then the cost of finding inter-group absolute distance and slope constraints between arbitrary vertices is still O(n^2), and this becomes the bottleneck of the inferencing component. At this time our system only removes a few classes of redundant constraints, and we do not know the cost of implementing a good, general redundancy filter.

5.3.4 Parameterizing an Illustration

Often it is convenient to be able to parameterize graphical illustrations. A slight modification to the algorithm described above allows simple relationships between scene objects and numeric text fields to be inferred during the snapshot process. We provide an Arguments window in which scalar values can be typed as the illustration is edited into new configurations. These values are interpreted by the algorithm as though they were distances, slopes, or angles between vertices. If one of the changing geometric relationships in the scene matches a changing numeric argument, a relative constraint is created between the two values.

In Figure 5.11 we have drawn a scrollbar in the Chimera editor, and we would like to equate the percentage typed in the Argument 1 field of the Arguments window to the *Y* position of the scrollbar's slider. We constrain the scene by providing the two snapshots depicted in Figures 5.11a and 5.11b, but in addition to presenting two valid versions of the scene's geometry, we type corresponding values in the Argument 1 text field. As shown in Figure 5.11c, after turning constraints on, we can adjust the scrollbar's slider by editing the value in this same text field, and pressing the Apply-Arguments button. Alternatively, we can adjust the slider, and the value of Argument 1 changes accordingly.

Myers presents a similar example of parameterizing scrollbar behavior in [Myers88]. His method linearly interpolates between two different constrained configurations, which is a more powerful abstraction, particularly for defining the behavior of widgets. For example, in Peridot the slider height can be parameterized with respect to the bottom and top of the scrollbar. This cannot currently be done in our system. In our example, Argument 1 is interpreted as proportional to the distance between two parallel lines—the top of the slider box and the bottom of the box containing the upper scroll arrow. So if the scrollbar is resized, the percentage parameter will no longer range from 0 to 100. Peridot's constraints were chosen for the domain of widget construction, and are specialized for this type of task. Our system provides a lower-level constraint set for the construction of general illustrations. The type of parameterization that our system provides is useful for many basic illustration tasks, such as the dimensioning of distances, slopes, and angles.

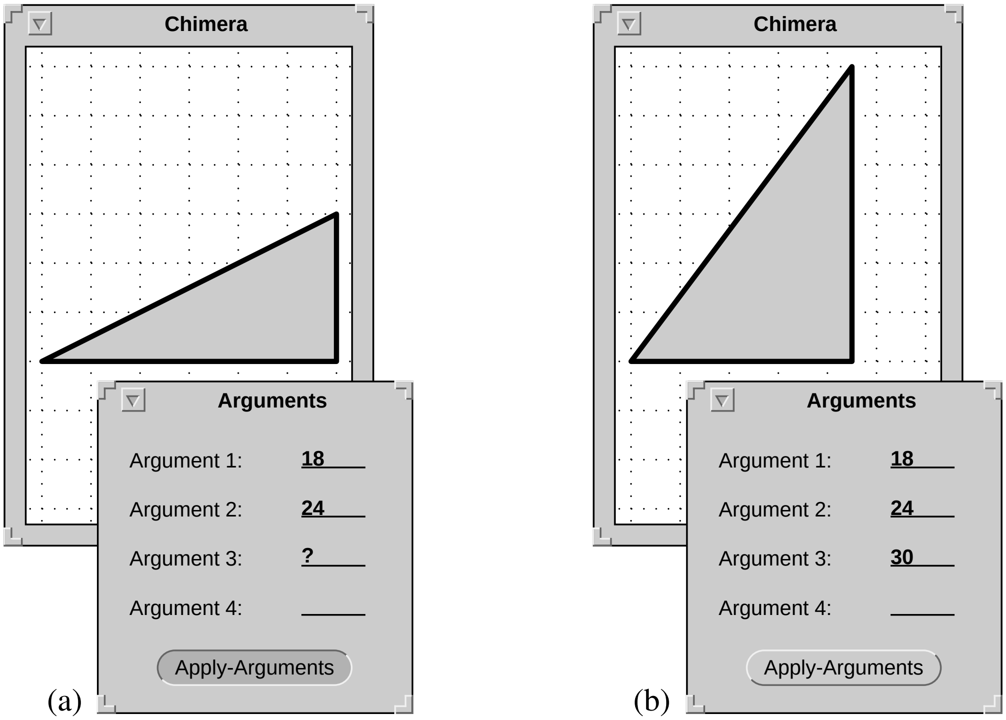


(a)
Argument 1: 0
Argument 2:
Argument 3:
Argument 4:

(b)
Argument 1: 100
Argument 2:
Argument 3:
Argument 4:

(c)
Argument 1: 50
Argument 2:
Argument 3:
Argument 4:

Figure 5.11 Dimensioning the height of a scrollbar. Initially two snapshots, (a) and (b), are specified. A new value for Argument 1 is entered in (c), and the scrollbar adjusts automatically.



(a)

Argument 1: 18
Argument 2: 24
Argument 3: ?
Argument 4:

(b)

Argument 1: 18
Argument 2: 24
Argument 3: 30
Argument 4:

Figure 5.12 Specifying a subset of the parameters. Only the first two parameters are specified in (a). The triangle resizes, and a value is computed for the third parameter in (b).

Since parameters of the illustration can be mutually dependent, the values of a subset may determine the rest. Sometimes the user may care to set only a few of the available parameters. For these reasons, we allow parameters to be either specified or unspecified. Specified parameters are constrained to their current value during the constraint solution, but unspecified parameters are allowed to vary. Figure 5.12a shows a Chimera editor scene containing a single triangle. Two previous snapshots (which are not shown) have constrained it to be a right triangle, with a fixed lower left corner, and horizontal base. They also have constrained Argument 1 to be proportional to the length of the base, Argument 2 to be proportional to the length of the vertical segment, and Argument 3 to be proportional to the hypotenuse's length. In Figure 5.12a, we type the desired lengths of each of the sides but the hypotenuse into the Arguments window. The question mark entered for Argument 3 requests that it be chosen by the constraint solver. After the Apply-Arguments button is pressed, the triangle resizes subject to its constraints, and Argument 3 is filled in with a suitable value.

5.4 Implementation

The Chimera editor is implemented mainly in Lucid Common LISP and CLOS (the Common Lisp Object System), with some C code as well. Our constraint solver is implemented in C, but the inferencing mechanism is in LISP. The code runs on Sun workstations under OpenWindows.

To solve constraint systems inferred from multiple snapshots (as well as constraints specified in the traditional declarative manner), Chimera uses the same Levenberg-Marquadt-based solver invoked by its constraint-based search and replace component. Part of the Levenberg-Marquadt method requires solving a system of equations to determine how the current solution estimate should change. If the error functions are not mutually independent (which is the case with redundant constraints), the system cannot be solved using Gaussian elimination. Instead, we use singular value decomposition [Press88] to find a solution at this step.

In looking for absolute and relative relations in the scene, it is important to build tolerances into the matching process. We use small, fixed, empirically-derived tolerances, just large enough to account for floating point inaccuracies during the construction and editing of the scene. If the tolerances were large, the number of incidental constraints would increase. Our small tolerances are on the order of fractions of degrees and millimeters. These small tolerances require that the snapshots be drawn accurately, so Chimera provides both grids and snap-dragging for this purpose.

Both the inferencing algorithm and constraint solver typically run at interactive speeds, on a Sun SparcStation 1+, for systems of the size presented in the chapter. The slowest snapshot (that of Figure 5.4b) took about 3 seconds. Constraint solutions were obtained in under a second in all cases but the window resizing example. This took somewhat longer because a large number of redundant constraints were passed to the solver by the inferencer. Further work on the inferencer should reduce the number of redundant constraints, and speed up constraint solutions.

5.5 Conclusions and Future Work

Snapshots appear to be a very intuitive way of specifying constraints, and often allow complex constraint systems to be specified with relatively few operations. However, we will not know until performing user-trials whether, and under what conditions, people prefer the technique to traditional declarative specification. Our personal experience with snapshot constraint specification in Chimera suggests that it is not a panacea. There are certain tasks for which it appears to be a simpler, more natural method of constraint specification, but for others, traditional declarative specification remains easier. The snapshot approach works well for highly constrained scenes, particularly those which easily can be manipulated into example configurations. Explicit constraint declaration is an additive technique rather than a subtractive one, and it often seems preferable for weakly constrained scenes, and those for which setting up snapshots would be difficult. There are a number of problems using snapshots that the traditional method does not have:

- Certain pictures can be difficult to edit into new configurations. In some of these cases it may be easier to specify constraints explicitly.
- Incidental, unintended relationships often occur in large scenes, necessitating extra snapshots.
- Redundant constraints are commonly passed to the solver, increasing solution costs.

When it is easier to specify constraints declaratively than by example, then the declarative technique should be used. We have built a traditional declarative constraint interface for our editor that is useful in these cases, and will allow us to better compare the two methods.

Incidental constraints can be reduced by restricting the classes of constraints that can be inferred. In our initial implementation, we inferred inter-group relative distance and slope constraints between any two pairs of vertices. This resulted in too many incidental constraints, so we restricted these constraints to pairs of two connected vertices (although there need not be a connection between the pairs). We still infer absolute distance and slope constraints between any two vertices, and intra-group relative distance and slope constraints between any two pairs of vertices. We are looking for additional restrictions that will not significantly impair the utility of the system.

Another way to reduce incidental constraints is to have the user select a set of objects prior to the beginning of a snapshot sequence, and have the inferencer look for constraints only among these objects. Partitioning the scene has the additional benefit of accelerating snapshots. In traditional constraint specification, constraints are also often added in partitions, to speed up solution. Currently we do not allow inferencing to be restricted to a subset of the scene, but this option is important for large scenes, and we plan to include it in the future.

One approach to reducing redundant constraints might involve using algorithms similar to those Chyz developed for maintaining complete and consistent constraint systems [Chyz85]. When a new constraint is added to the network, his algorithms determine which constraint must be eliminated to avoid overconstraining the system. These methods may allow us to reduce the set of constraints passed to the solver. However, we would not filter out most redundancies from our master constraint set, since after subsequent snapshots they may no longer be redundant.

There are a number of other interesting topics for future work. We would like to extend our system to handle constraints between non-geometric properties, such as color or font. Animating the constrained systems would provide an intuitive display of the set of constraints inferred, in the same visual language as the snapshot specification. We would like to provide an audit trail of snapshots by incorporating them into our graphical edit history representation, which is described in Chapter 6. This will allow individual snapshots to be eliminated and the constraint network recalculated.

It would be helpful to infer a few additional geometric relationships, such as the distance between a vertex and a line. This constraint could be easily added to the inferencing algorithm. Currently we infer constraints only among vertices present in the scene objects. There are cases when we would also like to infer relationships among implied objects, such as the center of a rotation, or the bounding box of an object. We also plan to allow constrained shapes inferred by our technique to be parameterized in more complex ways, and included as part of macros.

> *Daily, hourly since first waking on my Elysian couch, I reviewed those murals, wondrous, as faithful to my story and its several characters as if no chiseling sculptor, but Medusa herself, had rendered into veined Parian, from her perch in the great sixth panel, our flesh and blood.*
>
> — John Barth, *Chimera*

Chapter 6

Editable Graphical Histories

6.1 Introduction

Traditionally demonstrational techniques have been classified into two groups: those that work entirely on input and output examples, and those that look at the trace of the user's interaction. In fact, this is really a continuum, and the techniques presented in this thesis span the full spectrum. In graphical search and replace, the user places input and output examples in the search and replace panes and generalizes them by hand, so this technique lies completely in the input/output domain. Constraint-based search and replace works similarly, but the heuristic for determining whether to create a set or fixed constraint looks at how the constraint was instantiated. This second technique is mainly an input/output method, yet it uses the trace to choose intelligent defaults.

In the third technique, constraints from multiple snapshots, the system infers constraint sets independently of how objects have been transformed between snapshots, so it can be viewed as an input/output method. However the algorithm monitors the transformations between snapshots to do this efficiently, so it also reasonably can be classified as trace-based. With every chapter so far, the techniques described have used more and more trace information. The technique presented in this chapter follows this trend, and resides entirely in the trace-based domain.

One of the simplest trace-based techniques is *redo*: the user simply selects a set of commands from the interaction history for reexecution. Without a visual representation of these commands, the process of selecting a subset to be reexecuted can be difficult indeed. Many applications evade this problem by providing a command that redoes only the previous command. Most users can remember the last command they issued, particularly if they want to redo it! Visual representations of commands become more important as redo facilities increase in sophistication.

Visual representations of command sequences are also critical to applications with powerful undo facilities. Again, visual representations are not necessary if an application provides only the ability to undo the last command. But what about applications that allow arbitrary sequences of commands to be undone? There must be a mechanism to specify the bounds of these sequences, and it should work in conjunction with a visual aid to remind the user of his or her commands.

Allowing users to browse through the commands that they executed has additional advantages. Visual representations of commands permit users to review the operations that they performed in a session, as well as to communicate this information to others. Redo sequences are essentially simple, non-generalized macros. If users choose to archive a redo sequence or macro, they may need to review its contents at some later time. A visual representation will help here, and may also permit them and others to edit the sequence to remove bugs or add extra features.

Traditional text-based applications often visually represent command histories using the *spatial browsing* technique. In this representation, commands executed in a session appear one after another in a list. Some of these systems allow users to scroll through a list of all commands executed in a session, while others maintain a record of only the most recently issued commands. Since users of these applications invoke commands by typing text at a keyboard, this text makes a natural record of the users' interactions. Both the UNIX™ C Shell [Joy79], and the Doskey facility of MS-DOS [Microsoft91a] support this history representation.

With the advent of graphical user interfaces, application designers explored other means of representing command sequences. In these interfaces, users can invoke commands through mouse operations and keyboard accelerators, and they may be unaware of the textual command names. Furthermore, operands of these commands can be objects or properties that users specify through direct manipulation and the interface renders graphically. In these interfaces users may never construct or see a textual representation of their data, so it seems impractical to make them learn such a representation to understand the command history. To avoid casting the history in textual terms, some graphical applications allow users to view the history through *temporal reexecution*. In these systems, the application can be scrolled forwards and/or backwards in time, and the user infers the history by examining the changing state of the application canvas. Alvy Ray Smith's PAINT program had this capability [Smith78]. Applications with undo and redo commands, like GNU Emacs [Stallman87], provide a type of temporal reexecution by allowing the user to step through the history by hand, viewing the results of each command. Pygmalion uses temporal reexecution to animate visual programming steps applied to new data, though not to represent the actual programming history [Smith93].

A disadvantage of temporal reexecution is that only a single step in the history can be displayed at once. Trying to examine and modify a long history is akin to editing a text document in a line editor; the editor does not display ample context along with the data that the user needs to manipulate and understand. Accordingly, several researchers have developed *hybrid* techniques, that merge spatial browsing and temporal reexecution, yielding a history representation for graphical interfaces, while providing context.

The IGD hypermedia system, built by Feiner, Nagy, and van Dam, constructs a timeline out of screen miniatures to represent its user's navigation path through a document [Feiner82]. By selecting one of the screen miniatures, the user can return to a previously visited page. Christodoulakis and Graham developed a system that adds icons to a scrolling window at points in a multimedia presentation that the author deems important [Christodoulakis88]. In Makkuni's Chinese temple editor, architects share a design history represented as a sequence of icons. The system provides an additional record of the design process by showing component editors for the various steps, connected by threads relating their use [Makkuni87].

This chapter describes *editable graphical histories*, a new representation for histories in a graphical user interface. Editable graphical histories are based on a comic strip metaphor, depicting important events in the history of the interface within a sequence of panels. These histories use the same visual language as the interface itself. New users need not learn many new visual conventions in order to understand these histories. Editable graphical histories address some limitations of the other hybrid techniques. For example, the IGD system also represents the history as a sequence of panels, but the panels are exact miniatures of the screen. Differences between one screen and the next may be invisible if the change affects a small area. Prefabricated icons lack the expressive power to represent arbitrary graphical manipulations. A good history visualization structures the presentation to help the user better understand it. It emphasizes changes that are made at each step, rather than what remains unaltered. It allows the user to easily review past steps, and experiment with new changes. To this end, editable graphical histories incorporate five important features:

Inter-panel detail removal. If a new panel were generated for every user action, then the sheer number of panels would be daunting, and the histories would be wasteful of screen real estate. Also in many interfaces, multiple low-level physical operations can compose a single higher-level, logical operation, and the operations can be better understood as a unit. Editable graphical histories *coalesce* related operations into individual panels. Under user-control, these panels can be expanded to examine their component parts. The system automatically chooses which operations to coalesce together in a panel, based on a grammar for the command language and a user-settable default granularity level. Previous systems showed each action separately or only author-designated actions.

Intra-panel content selection. To represent a wide range of graphical actions, and to represent them clearly, this history mechanism automatically generates panels customized for this purpose. It chooses the contents of the panels and rendering styles for the objects depicted in the panels, based on heuristics and a rule-base. In contrast, previous systems used exact screen miniatures or prefabricated icons.

Interactive elaboration. With editable graphical histories, users can dynamically adjust the detail of the presentation according to their needs, and choose different detail levels for different parts of the history. Users can expand high-level panels into lower-level ones, at a variety of levels of detail, or coalesce related panels together.

Full history editing. These histories allow the user to review past commands, and undo or redo sets of them. Unlike previous displays of undo possibilities, however, editable graphical histories allow the user to view the past consequences of collections of previous commands, without having to execute undos and redos.

In-place modifications. Editable graphical histories allow changes to be made to the history sequence, in place, by edits to the panels themselves. Users can manipulate and query aspects of past state within the history, and propagate changes to the present.

The next section discusses the capabilities and interface of this history facility in the context of an example. Section 6.3 describes the rule language that governs the construction of these histories. Implementation details are described in Section 6.4. Section 6.5 describes how this graphical history representation evolved to its present form. Finally, Section 6.6 presents a summary and describes potential future directions.

6.2 Exploring Editable Graphical Histories (by Example)

To better explain the features of this history representation, this section presents a picture produced in Chimera, accompanied by sequences from its graphical history, and explains how edits to this history modify the picture. Figure 6.1 shows a diagram constructed in Chimera, illustrating the fuel subsystem of a military vehicle. A graphical history viewer appears in Figure 6.2, showing some of the commands executed while creating this diagram. The history viewer contains seven panels from a much longer sequence. Each panel includes a view of the screen displaying the effects of the editor commands it represents, and each panel also has a label. The label indicates the most important command represented by the panel, and the number of commands the panel contains.

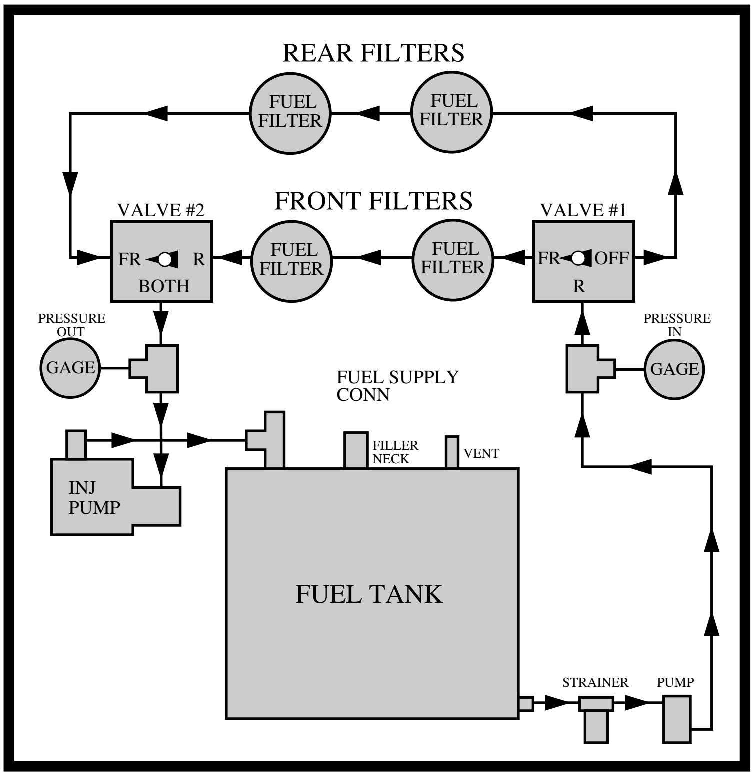


Figure 6.1 A fuel system for a military vehicle. (The military spelling of “gauge,” used in this figure, omits the “u”.)

Panels depict parts of the editor scene and control panel that are relevant to the commands that these panels represent. In the first panel of Figure 6.2, we turned on grids using the “Toggle-Grids” command. This step required only a single command that was invoked when we set the grids checkbox. The panel shows the portion of the editor control panel used to launch the command. This visual description supplements the textual label in two ways: first, it shows how this command was and can be invoked; second, it indicates the new state of the grids (in this case, turned on), which may or may not be a parameter to the command, but which is certainly important for interpreting the command’s effect.

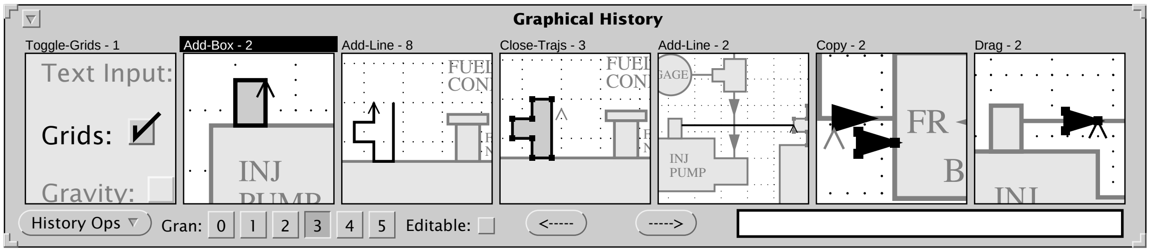


Figure 6.2 Editable graphical history that generated part of Figure 6.1.

Note that Chimera's entire control panel does not appear in the history panel—only those objects directly relevant to the operation, plus a little bit of context are included. Typically in each panel, Chimera shows those objects that act as arguments to the commands, as well as objects created and modified. The checkbox and its label are important to the operation for the reasons cited earlier, so they are included in the panel. As will be discussed in a later section, a rule set specifies to Chimera those objects to display for the various commands. In this case the rule-writer might have chosen also to show the resulting grid in the panel, but considered this unnecessary, particularly since the grid would probably be visible in subsequent panels. However, as shown later in this section, history panels can be split to show elements of both the control panel and the scene.

Other elements of the control panel appear in the first panel of the history to provide context, but the history mechanism deemphasizes these by graying them out. In editable graphical histories, objects are rendered in styles chosen according to their roles in the explanation. The history system renders those objects that participate directly in the operations normally, and it subdues objects included in the panels for context by lightening their colors. Examples of contextual objects in the first panel include the Text Input label, the Gravity label, and its checkbox.

The second panel of this history sequence shows a box being added to the scene. It is clear which box is being added, since its colors are not subdued, and the caret is attached to its corner. Users of the system know that the caret is the software cursor, representing the editor's current position, and is always attached to new objects added to the scene. In portraying a sequence of commands, editable graphical histories use the visual conventions of the interface, and do not rely on special meta-objects, such as arrows and X's to convey the meaning of the operations. The motivation underlying this decision is that all people that understand the application's interface should understand its history representation. It should not be necessary to learn the meaning behind a suite of meta-objects for the sole purpose of interpreting histories. Essentially, editable graphical histories communicate using the visual language of the interface itself. They supplement the language slightly by adding multiple rendering styles, but by keeping the number of styles low (just one in addition to normal rendering), they avoid a steep learning curve.

Another way that the history representation avoids the need for meta-objects is through the use of *before and after* panels. People can perform a mental differencing operation on these panels to determine the effects of an operation. As illustrated in Section 6.5, earlier versions of Chimera's histories broke up all editor steps into these panel pairs. Though this is no longer the case, *before* panels (called *prologues*) can usually be found in the history for *after* panels (called *epilogues*).

For example, in the second panel we want to be sure which box is being added to the scene. Though this is clear from other visual clues, we realize that Add-Box operations are really compound logical operations consisting of two basic physical operations: the first places the caret where one corner of the box should be (using the Move-Caret operation), and the second uses the physical Add-Box operation to drag the caret to where the opposite corner should be located. The label of the second panel includes a “2” to indicate that this panel really does include multiple physical operations. We can expand this panel into its component parts by selecting it, and changing its granularity value.

Panels can be selected using the mouse. One mouse button clears existing selections, and sets the new selection to be the panels swept out by the cursor. The other button extends the selection. Currently Chimera allows a single contiguous sequence of panels to be selected. Chimera highlights selected panels by rendering their labels in reverse-video (white letters on a black background). The panel selection mechanism and feedback is similar to text selection in some popular mouse-based environments, and it is already familiar to many computer users. The second panel of Figure 6.2 is selected.

After we select this panel, the numbered radio buttons at the bottom of the history viewer displays its granularity level. By adjusting the granularity of a panel, we adjust the cohesive “glue” that holds its individual operations together. This granularity value is not related to the search parameter described in Chapter 3. The granularity numbers themselves are not significant to the end user—only their relative ordering. At larger granularities more operations tend to be coalesced together in the same panel. When we select the second panel, the history mechanism highlights the granularity button labeled 3. When we press granularity button 2, this panel subdivides into the two panels selected in Figure 6.3. The first new panel shows the Move-Caret command that sets up the Add-Box command in the following panel. Though the Move-Caret command shows the effect of moving the caret to where we want the box to begin, it also acts as a prologue to the following Add-Box panel, by showing the scene before the box was created.

The ability to alter the granularity of panel coalescing is crucial. For fine-grained exploration and editing of the history, users often must examine individual operations. However, it is usually easier to understand the history when more panels are coalesced together, since then they represent logical operations rather than physical ones. For example, the logical operation of adding this box includes two operations that specify the two corners.

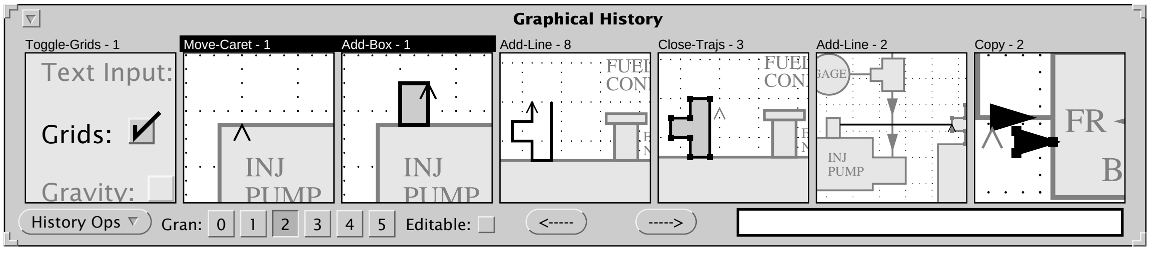


Figure 6.3 Prologues and epilogues. Expanding the selected panel of Figure 6.2 produces two panels, showing the scene before and after a box is added.

By representing these two operations together, the graphical histories structure the presentation to convey a single important idea and facilitate understanding. Coalescing these panels has the ancillary advantage of reducing the screen real estate consumed by the explanation. When no panels are selected, the highlighted granularity button indicates the default granularity to be used for constructing new panels. Pressing a different granularity button changes the default setting.

All of the panels in Figure 6.2 except the first contain multiple operations, so each can be expanded into additional panels. As another example, we select the third panel of Figure 6.2, representing the creation of a polyline. We expand this panel once, and it is replaced by seven panels depicting the individual Add-Line operations that build the polyline. This is shown in Figure 6.4. The first of these new panels contains two operations. If we were to expand it, Chimera would generate two new panels: a Move-Caret panel that sets one end of the polyline, and an Add-Line panel extending a line from it. Panel groupings form a multiple level hierarchy, and all levels of the hierarchy can be easily explored. After viewing the panels of 6.4, we can restore their original granularity, thereby coalescing them into a single panel.

We now resume examining the panels of Figure 6.2. In the fourth panel we select the polyline created in the prior panel, and use the Close-Traj command to close it and fill it. The prior panel serves as a prologue for this panel. In the fifth panel we draw a line between the box and polyline that we created during this history sequence. Again, the color of the line and position of the caret make it clear which object was added. By expanding this panel to show a prologue, we would make this even clearer. In the sixth panel we select an arrowhead elsewhere in the scene, and copy it. In the final panel we drag the arrowhead to a position on the line we created. If we translate the same object multiple times, with no intervening selections, then these translations are coalesced together, being equivalent to a single translation.

In addition to being useful for reviewing past commands, these graphical histories serve as an interface for editing the past. This history mechanism enables the user to delete previ-

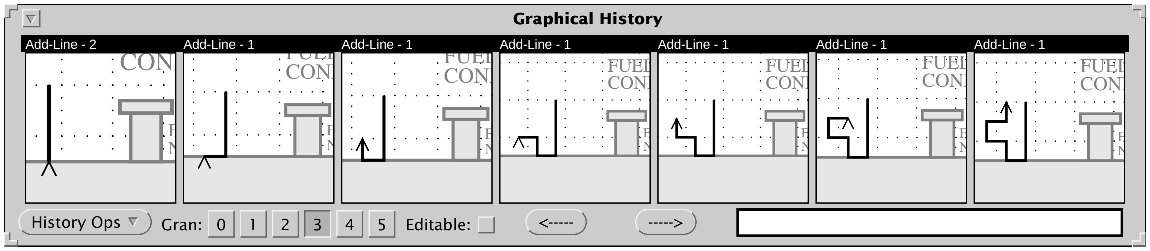


Figure 6.4 Expanding an Add-Line. The third panel in Figure 6.2 expands to show some of its component operations.

ous commands, and add new commands anywhere in the temporal progression. For example, an earlier version of Figure 6.1 had a capping rectangle at the top of the filler neck of the fuel tank. When we show the current illustration to our supervisor, she requests that we add the capping rectangle back. Fortunately we can scroll through the edit history of the document using the arrow buttons at the bottom of the history viewer, until we find the unwanted operations. These two operations appear selected in Figure 6.5. The first panel shows the rectangle being selected, and the second shows it being deleted. Chimera will not coalesce the selection with the deletion at any granularity level, since the earlier panel provides the only clue to the deleted object's identity. After selecting these two panels, we choose the Delete-Selected-Panels command from the History-Ops menu in the history viewer. The selected panels disappear from the history, and by deleting the deletion, we restore the capping rectangle to the editor scene.

The remaining panels of Figure 6.5 show steps in the creation of the fuel filters. In the third panel we add the text string "FILTER" to the circle. At this point we have created the first fuel filter element of the schematic. In the fourth panel we select the elements of the filter element and copy them, yielding a second filter. In the fifth panel we drag the new filter to a different location. In the sixth panel we extend the selection to the original filter and copy them both, yielding two more filters. Finally, in the last panel we drag these two new filters to a different place in the scene.

In addition to deleting operations from the history, we can add new ones. To do this, we make the panels *editable* by selecting the checkbox at the bottom of the history viewer. This replaces each of the visible static panels with a fully editable Chimera scene, effectively the same as the one that existed at the time represented by the panel. The viewing transformation for each panel's editor scene is the same used to display the panel objects originally, so the editable panels look identical to the static panels, with one exception. Chimera renders all objects in editable panels in their normal rendering style, since users may want to query and manipulate their properties. Users can scroll the editable windows to view parts of the scene not displayed in the small panels, or load this scene into a larger editor window.

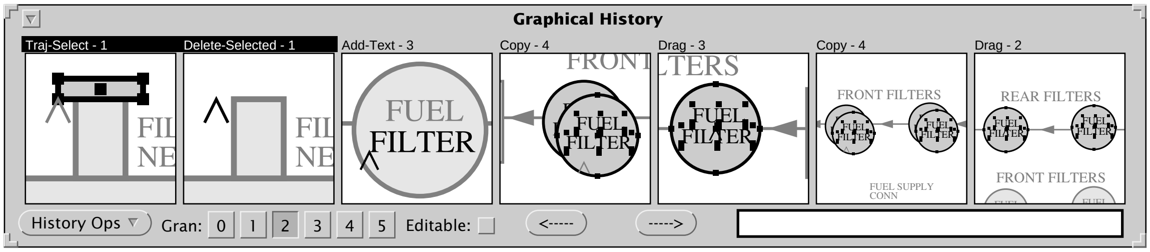


Figure 6.5 Another sequence from the history that generated Figure 6.1.

After showing another draft of the picture to our supervisor, we are asked to modify the color of the filter elements. These components of the illustration should have white letters on a black background. To make this change, we could edit all the filters in the illustration by hand or use graphical search and replace, but instead we choose to modify the original filter, after it was created, but before any of the copies were made.

To do this, we select the two text strings in the third panel of Figure 6.5, specify the color white in the Text Input widget of the control panel, and execute the Set-Fill-Color command. Next, we make the circle in this same panel black. We select this panel, and execute the Propagate-History-Changes command from the History-Ops menu. Changes that we made to the history are inserted at the proper point in the illustration construction process. The illustration updates automatically to incorporate these changes. Figure 6.6. shows the modified scene. Note that all of the fuel filters have been updated to be white-on-black. Also the capping rectangle that was restored by the previous history edit appears in this illustration.

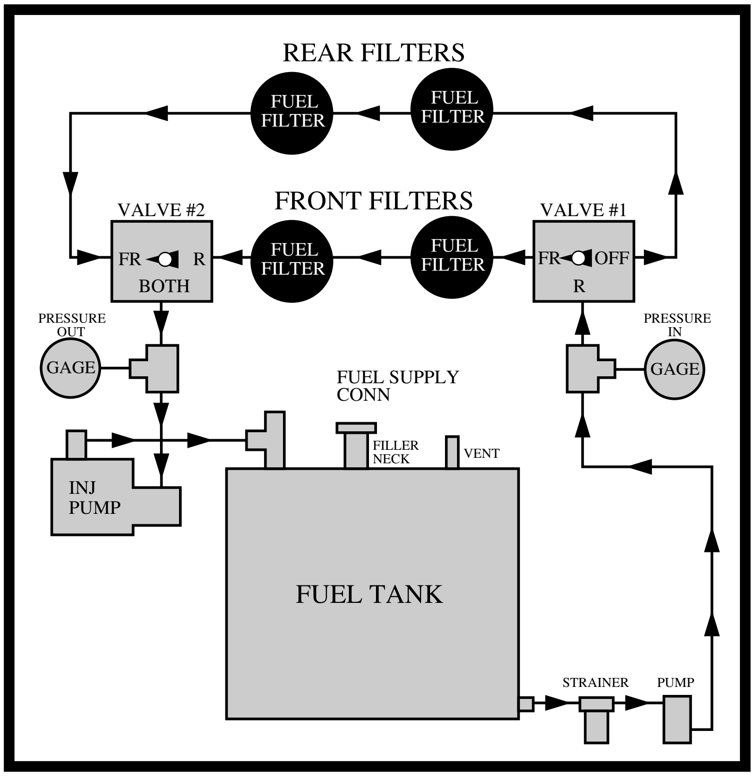
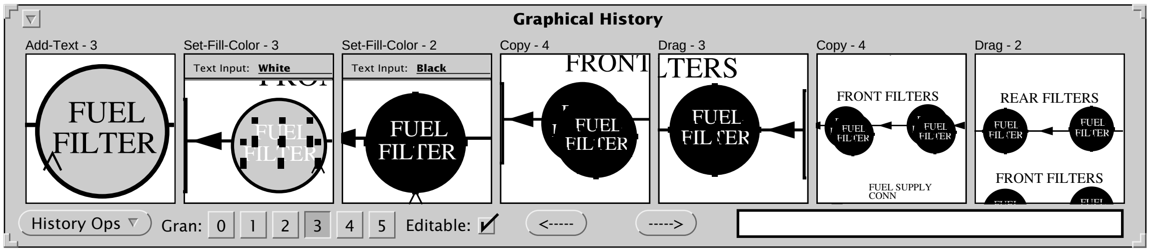


Figure 6.6 The resulting scene, after restoring the capping rectangle of the filler neck, and changing the colors of the original fuel filter, prior to the copy.

Propagating changes into the history also generates updates to the history representation itself. Figure 6.7 shows the history as it appears after our edits. The deleted panels have been removed, and the two Set-Fill-Color operations have been given their own panels within the history. The representation of these two panels is worth noting. Since we used the Text Input widget to specify the color argument, a copy of the widget appears at the top of the panels. The other argument to the Set-Fill-Color command is supplied by the selection in the editor scene, and it too is important to convey to the user. As a result, Chimera splits these panels into two subpanels, showing the critical contents of both windows.



Text Input: White

Text Input: Black

Editable: ☑

Figure 6.7 The modified history, after deleting a pair of panels, and propagating changes to the fuel filter.

Also the subsequent panels in the history are altered when the history is edited. The history mechanism automatically updates these panels to reflect earlier changes. The fuel filter and its copies now appear with their new colors in later panels. Also, if any of these panels were to include the region occupied by the capping rectangle, it would now appear. Note that the history viewer shown in Figure 6.7 differs from the viewers shown earlier in that it is editable. As explained earlier, Chimera renders all objects present in editable panels in their normal rendering style.

6.3 The Rule Set

To automatically construct graphical histories like the ones shown in the last section, Chimera needs information about the command set of the application. Particularly, to govern command coalescing, Chimera must have information on which physical commands combine together to form logical commands. Also, to help with composing panels, Chimera requires information about the types of objects that need to be included in the various panels. This information is represented by a collection of rules. An earlier implementation had this knowledge encapsulated in procedures, but it became difficult to maintain as Chimera's command set grew. Changing to a declarative representation has made it easier to add information about new commands, debug the knowledge base when Chimera produces unexpected results, and in the future it will make it easier to apply Chimera's history mechanism to new applications.

Chimera history rules are not intended to be edited (or even seen) by the end user, though they are fairly easy to write, and could certainly be learned by sophisticated users planning to extend the editor command set through programming. The rules are Lisp s-expressions of the following form:

```
(history-class classname
               :parents parentclasses
               :rules ([[(granularity-level
                         ([regular-expression]^n)
                         ([([panel-element]^+)]^n)
                        )]^+)
               :contents ([panel-element]+)
               )
```

In this expression, keywords are in bold, [item]$^+$ represents one or more consecutive occurrences of item, and [item]^n means *n* consecutive occurrences of item, where *n* is positive and the same value for the entire expression. Keywords beginning with a colon are optional, and can appear in any order.

Each history-class rule defines a new classname, the name of a new command class in an object-based command class hierarchy. Having a command class hierarchy enables us to write rules that refer to broad collections of commands at once. The :parents keyword specifies the parent classes of this new command.

The :rules entry declares how other commands coalesce to form this one, as well as the contents of the resulting panels. This entry contains any number of sub-rules tagged by a granularity level. The sub-rule will only be tested when the current system granularity level is greater than or equal to its granularity. Sub-rules are tested in order of decreasing granularity. Also, a sub-rule will only be tested against a command sequence that ends in a command of the rule’s class. The :contents entry specifies the default panel contents for a command, when none of its coalescing rules fire, or the panel contents information is omitted from a sub-rule. The following example should make this clearer. Below is a set of history rules for several Chimera command classes:

```
(history-class ‘(Dot-Select Segment-Select Traj-Select
                  Top-Select Extend-Dot-Select
                  Extend-Seg-Select Extend-Traj-Select
                  Extend-Top-Select Select-All Deselect-All
                  Incl-Box-Select Excl-Box-Select)
               :parents ‘Select-Op)
```

Chimera includes more than 10 different object selection commands (for example, Dot-Select selects a single vertex, Segment-Select selects a segment-level object, and Incl-Box-Select selects all objects within a rectangular area). The above rule defines all of Chimera's object selection commands to be sub-classes of a parent-class called Select-Op. Many Chimera commands act on the current selection, so these commands should typically coalesce with selection operations. Coalescing rules will now be able to specify Select-Op to refer to any of these commands, rather than specifying them all explicitly. The following rule indicates how all Select-Ops should coalesce:

```
(history-class ‘Select-Op
              :rules ‘((*gran3*
                         (((:+ (:or Select-Op Caret-Op))))
                         ((SELECTED)))
                       (*gran2*
                         (((:* Move-Caret) (:e Select-Op)))
                         ((SELECTED)))
              :contents ‘((SELECTED)))
```

There are two different ways that Select-Ops can coalesce, as specified by the :rules entry. At a granularity of 3 or above (*gran3*), any contiguous sequence of Select-Ops and Caret-Ops, ending in a Select-Op, will coalesce into a single panel. This is given by the regular expression (:+ (:or Select-Op Caret-Op)), which represented traditionally would be $(\text{Select-Op} \mid \text{Caret-Op})^{+}$, with the extra constraint that the sequence must end in a Select-Op, since this rule is being defined for the Select-Ops class. It makes sense to coalesce together multiple Select-Ops, since when Select-Ops occur in a sequence, later Select-Ops either reset or add to earlier ones. Move-Caret operations coalesce with Select-Ops, because Select-Ops reset the caret position, overriding earlier Move-Carets in the same sequence. The line following this regular expression indicates that panels coalesced by the expression must show the selected scene objects.

The above rule also contains a sub-rule that specifies at granularity 2 or above (*gran2*), any number of Move-Caret commands, followed by a single Select-Op command will be coalesced together. This is captured by the s-expression: (:* Move-Caret) (:e Select-Op), which in common regular expression notation would be: Move-Caret^{*} Select-Op. So at a higher level of granularity, multiple selections can be coalesced together, but when the granularity is reduced, only a single selection can appear in a panel. At this granularity level, too, the panel presentation must include all selected objects.

If a sequence of commands ending in a Select-Op does not match any of these coalescing rules, subject to granularity restrictions, then the Select-Op will be allocated a panel to itself. The :contents entry indicates that this panel should also show the selected objects in the scene. In addition, the :contents entry is used as the default panel contents for all of the sub-rules that lack an explicit content specification. In this rule, we could have omitted the content specification for all of the sub-rules, since they are identical to the default.

The next rule is similar to the first discussed in this section. It simply declares Rotate, Scale, Flip-About-X, and Flip-About-Y as Anchor-Transforms. We call these Anchor-Transforms since they all depend on the center of the coordinate system, called the *anchor*. The first two commands rotate and scale selected objects about the anchor, the second two flip selected objects about coordinate axes going through the anchor. The anchor must be placed prior to Anchor-Transforms.

```
(history-class ‘(Rotate Scale Flip-About-X Flip-About-Y)
               :parents ‘Anchor-Transform))
```

Before concluding this section, we consider one final rule that introduces additional features of the history rule language. This rule defines how Anchor-Transforms should coalesce with the commands immediately preceding them. The Anchor-Transform rule appears below.

```
(history-class ‘Anchor-Transform
        :rules ‘((*gran5*
                  (((:* (:or Select-Op Anchor-Op Caret-Op))
                    (:+ Last-Op)))
                  ((CARET ANCHOR SELECTED)))
                 (*gran4*
                  (((:* (:or Select-Op Anchor-Op Caret-Op)))
                   ((:+ Last-Op)))
                  ((CARET ANCHOR SELECTED)
                   (CARET ANCHOR SELECTED)))
                (*gran3*
                  (((:* (:or Select-Op Anchor-Op Caret-Op)))
                   ((:e Last-Op)))
                  ((CARET ANCHOR SELECTED)
                   (CARET ANCHOR SELECTED)))
                (*gran2*
                  (((:* (:or Select-Op Caret-Op)))
                   ((:e Last-Op)))
                  ((CARET ANCHOR SELECTED)
                   (CARET ANCHOR SELECTED))))
        :contents ‘((CARET ANCHOR SELECTED)))
```

At the highest level of granularity, any number of contiguous Select-Ops, Anchor-Ops, and Caret-Ops, followed by one or more Last-Ops coalesce together into a single panel.

Last-Op is actually a special keyword, rather than a user defined class. It expands to the most specialized subclass of the last operation in the sequence of commands against which we test the rule. Recall that this rule will only be matched against command sequences ending in an Anchor-Transform. If this Anchor-Transform is a Rotate operation, then this sub-rule will match contiguous Rotates, but not Rotates mixed with Scales. This feature helps prevent class rules from being too general to be useful.

Since Select-Ops, Anchor-Ops, and Caret-Ops all contribute towards setting up Anchor-Transforms, they can be viewed as part of the same logical operation. So at the highest granularity level, these commands preceding any number of the same Anchor-Transform, coalesce together. The next granularity level (*gran4*) appears to have a similar rule, but the parentheses grouping differs. The composite regular expression from the prior rule is split in two, so at this granularity two panels will be formed out of matching command sequences. Any number of Select-Ops, Caret-Ops, and Anchor-Ops can group together in the first panel, which serves as a prologue, and one or more of the same Anchor-Ops group together in a second panel, the epilogue. This sub-rule also has two content specification expressions for the two panels that it generates. Both panels show the Caret, Anchor, and Selected objects, since they affect the operation. Potentially, a single rule can create any number of panels.

At the next lowest granularity level (*gran3*), the rule specifies that only a single Anchor-Transform can appear in the epilogue panel. At granularity 2 (*gran2*), Anchor-Ops can no longer appear in the prologue panel. If none of these sub-rules match given the current granularity level, the Anchor-Transform at the end of the command sequence will be given its own panel.

Chimera’s rule language is extensible. Descriptive content names, such as ANCHOR, CARET, and SELECTED, are ordinary symbols with functions on their property lists. These functions return a set of tagged objects for the history mechanism to incorporate in the panel. As will be discussed in the next section, objects are tagged according to their role in the explanation. The rule writer need not know the details of coding up such a function, and can simply refer to the descriptive name. However, a person can add a new content name to the language by writing an additional function, without modifying existing ones.

Instantiated widgets in Chimera’s UIMS can have names. These names can be used as part of the content specification to declare that particular widgets should appear in the panels. Content specifiers can also include additional tags for panel objects. When a single rule divides a command sequence into multiple panels, the PRIOR-VIEW directive can be included in the content specifier of any panel (other than the first) to guarantee that panel will show the same view as the prior one.

6.4 Implementation

This section describes how Chimera automatically builds history panels for a given command sequence. To build panels, and allow the user to expand them, coalesce them, and make them editable, Chimera needs to recreate past editor states. The next subsection describes how Chimera does this. Building panels consists of three steps: partitioning operations into panels, choosing the contents and view of panels, and rendering them. Three later subsections describe these steps.

6.4.1 History Contexts

When Chimera executes a command, it creates an operation object. Operations have three fields: a type, a redo expression, and an undo expression. The type field indicates the kind of operation. The redo expression, when executed in the state that existed prior to the operation, will produce the next state, and the undo expression, when executed in the state following the operation, will produce the previous one.

Saving copies of the application state between each operation would waste memory. Instead, Chimera can regenerate any previous application state by taking the current state, and executing the undo expressions of the operations issued in between. Chimera has an object type, called a *history context*, that contains an application state, a list of operations executed prior to this state, and a list of operations executed subsequently. Both lists order the operations temporally, with those nearest the state at the front. These lists indicate where in the operation sequence the context's current internal state occurred. Chimera has internal procedures that take a history context, and a single operation, and moves the history context forward or backward in time to the point immediately preceding or following the operation. This is simply a matter of, one-by-one, removing operations from the front of a list, executing its undo or redo expressions, and adding it to the front of the other, until the given operation appears at the head of the appropriate list.

6.4.2 Partitioning Operations

Even when the history viewer is not open on the screen, additional operations are added to the internal history list as the user executes new commands. When the user invokes the history viewer, these operations must be partitioned into panels. Chimera takes the list of operations, ordered from the last executed, and matches them against regular expressions in the rule base. It parses the history from the end, using reversed versions of the regular expressions provided by the rules. Only a subset of the regular expressions must be tested: those that have a granularity less than or equal to the history viewer's current granularity level, and are specified by rules for the leading operation's member classes. The history mechanism tests regular expression sub-rules in order of decreasing granularity, and finds the longest regular expression match for the first matching sub-rule. If a sub-rule specifies expressions for multiple panels (e.g., a prologue and epilogue), Chimera looks for a contiguous sequence matching the concatenation of all these panels' regular expressions, but keeps track of which operations map to each panel. As soon as an expression parses a non-empty sequence starting at the end, the system allocates the necessary number of panel objects, in which it stores a pointer to the sub-rule that matched, as well as the matching operation sequence. The process continues on the remaining portion of the history list that was not matched in the last iteration, until all operations have been partitioned into panels.

One reason for parsing from the end is that it simplifies incremental parsing. Chimera's history viewer can be left open while the scene is edited. When the viewer is open, additional panels appear at the end, showing new commands as the user executes them. It is important that these panels be generated quickly so that the editing process is not disrupted by panel formation, and the history presentation keeps up-to-date. Every time a new command executes and the history viewer is open, the operation is consed onto the front of the complete history list, and the list is reparsed until a sequence parses to the same panel it did before. As soon as a sequence parses the same way, we know the last parse for the rest of the list is still good. Typically only a small number of operations need to be reparsed. Chimera creates new panel groupings for these operations, and removes the old groupings. Thus existing panels at the end of the history can be remade in response to subsequent user commands, and new panels can appear.

For example, consider again the third panel of Figure 6.2 showing the construction of a polyline. To set one endpoint of the polyline, we used the Move-Caret operation. This initially parsed into a Move-Caret panel by itself. When we added the first line, the regular expression sub-rule of the Add-Line rule (Move-Caret* Add-Line$^+$) parsed the new Add-Line operation and the Move-Caret together into an Add-Line panel. Each of the six additional times we added new lines, the Add-Lines and Move-Caret parsed together into a new last panel.

6.4.3 Choosing Panel Contents (or a Panel with a View)

Next, Chimera must choose the contents of the new panels. Since each panel must reflect the visible state at the time it represents, Chimera first recreates the scene state that existed after the last operation of the panel was executed. It does so by having a history context associated with each history viewer that can be temporally scrolled to any point in the operation sequence. This history context can be the same one associated with the editor scene, or a copy, as described in a later section.

After establishing the proper scene state, Chimera examines the content specification term of the panel's matching sub-rule. As explained earlier, content specifications can be descriptive terms, such as CARET or SELECTED, from a presupplied yet extensible list. They can also be names of instantiated widgets. For each of these content selectors, Chimera fetches the corresponding objects, along with a descriptor indicating in which scene they reside. All objects in Chimera belong to a *scene*, be they widgets in a control panel, or graphical shapes in an editor buffer. This is similar to Garnet's retained object model [Kosbie90]. Chimera checks whether all objects in a panel belong to the same scene. If they do not, it allocates multiple subpanels, each of which receives objects from a single scene. This leads to a split-panel effect, as shown in the second and third panels of Figure 6.7.

While fetching objects to be included in each panel, Chimera supplements these objects with tags to describe their role in the explanation. For example, objects included by the SELECTED specifier would be tagged as SELECTED, since this is why they are included. Additional tags for objects matching the content descriptors can also be specified in the rule itself. Next, Chimera calculates an initial estimate of the *view* of the panel, the region of the scene that it will show, to be the bounding box of the set of objects chosen for inclusion thus far. If a panel has multiple subpanels, Chimera calculates an estimate for each.

It is important to include a little bit of scene context in each panel, to help indicate the region that it spans. For example, if an Add-Box panel only shows the created box, then it might be unclear which box of many in the scene was built at this step. Chimera adds a landmark to each panel, to help disambiguate the location of the view. Ideally this object would have a unique appearance, semantic importance, and lie near the objects already chosen for the panel. Finding good landmarks can be difficult, and is an interesting research topic. In Chimera's current implementation, we look only for an object that satisfies the last property. It may be that an object chosen for the panel previously had been tagged as a landmark. For example, labeled widgets make good landmarks, and are tagged this way during the content selection process. However, if no panel objects were already identified as landmarks, Chimera looks for the scene object whose complete inclusion in the panel would result in a view of minimum area. Chimera adds this object to the panel or subpanel, and labels it Landmark and Context.

For all panels generated by the single firing of a rule, Chimera attempts to use the same view. The motivation for this is that these sequences typically represent before and after comparisons (such as a prologue and epilogue), and the same view helps the user perform a mental differencing operation. Chimera computes the bounding box of all the views represented by such a sequence, and if the area of any of the views exceeds a certain percentage of the area of the combined view, then its view is set to be the combined view. Also, if the content list of one of these panels, other than the first, contains the PRIOR-VIEW directive, then its view is set to be that of the prior panel.

In the next step, for each panel in the sequence just considered, Chimera chooses the placement and relative dimensions of its subpanels. Automatic layout has been the subject of other graphics research. Chimera uses a simple technique, initially assigning the entire panel extent to the subpanel deemed most important (editor scenes have priority over control panel scenes), and breaking off horizontal or vertical strips for the other subpanels (depending on whether their view is more horizontal or vertical).

Then, for each panel or subpanel, Chimera reshapes the view. It increases the width or height of the view so that its aspect ratio (width divided by height) is the same as the subpanel itself. This ensures that the scene shown in each panel is isotropically scaled. Also in this step, Chimera enlarges the views by a small percentage (8%) in each dimension to create a margin in the panels and subpanels.

Now that the final view is set, Chimera determines all scene objects that lie at least partially within the view, but were not yet chosen for inclusion in the panel. It includes these objects, and tags them as Context. Finally, the history mechanism sorts the objects in the panel to have the same overlap order as those in the scene.

6.4.4 Rendering Panels

To draw a panel in the history viewer, Chimera first sets the viewer's history context to the state represented by that panel. Next it renders all of the objects selected for inclusion in the panel. During the object selection phase, all objects were tagged with one or more roles. These tags are used during the rendering process to determine a style for displaying these objects. In Chimera's default configuration, objects tagged as Context are rendered with lightened colors, formed by linearly interpolating their RGB color values towards white. Alternatively, users can choose an option to desaturate the colors of contextual objects, or make them invisible. Chimera renders objects not tagged as Context as they appear normally in the scene.

Isolating the scene composition step, which determines the roles of objects, from the rendering process, makes it very easy to alter the appearance of objects playing the same role in the explanation. The use of tags for this purpose was used earlier in the APEX system for generating automated pictorial explanations [Feiner85]. To incorporate a new style, only the final pass, the rendering step, needs to be modified by adding a tag test and graphics code that implements the display effect.

6.4.5 Making Panels Editable

When the user sets the "Editable" checkbox at the bottom of the history viewers, all of the panels become editable Chimera scenes. For each of the panels in the viewer, Chimera recreates the state represented by that panel, using the viewer's history context. After it reproduces a scene state, it copies the state, installs a child window in the viewer with the same dimensions as the panel (or sub-panel), and makes this the window of the new graphics scene. It also sets the initial view of the new editor scene to be that of the original uneditable panel, so that its contents appear to have the same size and location.

When Chimera copies a reconstituted scene state into a newly created child window, it copies the entire scene, not just the visible portion. By copying just the visible portion, Chimera would save memory, since typically the panels show only a small portion of the scene. However, then users could not scroll the panel, as they can in the current system.

An alternate approach would be to initially copy just the visible portion of the scene, but to load the scene in its entirety if the user tries to scroll the panel.

Though the history mechanism makes copies of the entire scene for each visible panel, it only instantiates editable scenes for panels visible in the history viewer. Off-screen panels only receive editable scenes when they scroll onto the viewer. The history mechanism reclaims memory associated with editable panels as they scroll off.

6.4.6 Editing the History

Currently in Chimera, there are two ways to edit the history—existing panels can be deleted, and new operations inserted. When the user chooses to delete a sequence of panels, the system first reverts the editor scene to its state immediately prior to the operations represented by the selection. It then redoes all of the undone commands after the selection. This has the net effect of undoing the selected commands at the point they occurred in the edit sequence. When users modify editable panels and propagate the changes into the history, the history mechanism first undoes the editor scene to the point in time represented by the panel, redoes the panel modifications in the real editor scene, and then redoes the commands that were undone. This has the effect of inserting new operations in the middle of the history.

Since each editable panel is its own editable scene, Chimera records edits to the panels in their own history lists. This makes it simple to keep track of panel modifications to be propagated into the main scene’s history. Since editable panels are Chimera scenes, we can open up history viewers on editable panels of other history viewers, and history viewers on editable panels of history viewers on editable panels of other history viewers, and—you get the idea.

When the user either inserts or deletes history panels, the system deletes all panels after the change, and builds new panels for the operations after this point, as though they were just executed and added to the end of the history. This has the effect of removing panels selected for deletion, and creating new panels for operations inserted into the history. Also, by rebuilding these panels, the history mechanism allows changes to the history to be reflected in the new panels. If Chimera kept the old panels, they might reflect antiquated scene state.

6.4.7 Refreshing the History Viewer

Implementing refresh of the history viewer was tricky. If we have a viewer filled with panels, and the first panel becomes damaged (perhaps a menu appeared and disappeared on top of it), how should the system repair it? Or if the user decides to scroll the viewer backwards, how can the system regenerate earlier panels? Initially we considered saving the PostScript for each of the panels in a database, and restoring the panels from that.

Some initial experiments showed that this would be too slow (in part because of the inefficiency with which Lucid Common Lisp prints floating point numbers).

Eventually we decided to repeat some of the panel composition and all of the rendering for each panel to be refreshed. At first we had feared that this would take too long, particularly since the history viewer's history context must be restored to the state represented by the panel, prior to rendering it. However, since graphical editing commands tend to be fairly simple, it is actually very fast to undo and redo the operations stored in a viewer's worth of panels. Every viewer has an associated history context, that typically is set to a state represented somewhere in its visible panel sequence.

When a visible panel must be refreshed, the context's state can quickly be adjusted to that of the panel. History viewers that are scrolled all the way to the present, share the context of the main editor scene itself. So when panel screen damage occurs, the scene is undone to the damaged panel, the panel is refreshed, and the scene is redone to the present. All this happens transparently, and the history mechanism turns off refresh to the main editor window during these undos and redos. When the user scrolls the history viewer back in time from the present, Chimera gives it a new history context that remains closer to the states represented by the panels. If the user scrolls the history viewer back to the present, Chimera reclaims its history context, and reinstalls that of the main editor scene.

6.5 History of Editable Graphical Histories

This section discusses how the editable graphical history representation has evolved from its initial implementation in the Chimera editor to its current form, and the reasons for the changes. One of the goals behind the design of these histories is that the users of an application should understand them without learning a multitude of new visual conventions. This discouraged the use of special meta-objects in the panels, such as arrows to signify a translation, when such objects do not appear in the original interface. Instead Chimera generates a sequence of before and after panels, to explain the operations, with the intent that the user would perform a mental differencing operation on the panels to detect the change. In earlier implementations of its history, Chimera always grouped panels into prologues and epilogues.

Figure 6.8a is an editor scene containing a simple drawing of four heads, generated by the steps depicted in the history shown in Figure 6.8b. The panels are grouped in pairs: the top panel shows the scene before the main operations represented by the pair, the bottom panel shows the scene afterwards. Typically the prologues show the steps used to set up an operation, and the epilogues show the operation's effects. The first panel pair includes those operations used to draw the outline. The second panel pair adds an eye, the third a mouth, and the fourth an ear. The fifth panel pair shows the head being selected and scaled. In the remaining panels, the head is copied, the copy is translated alongside the original, the two heads are copied, and the copies are translated above.

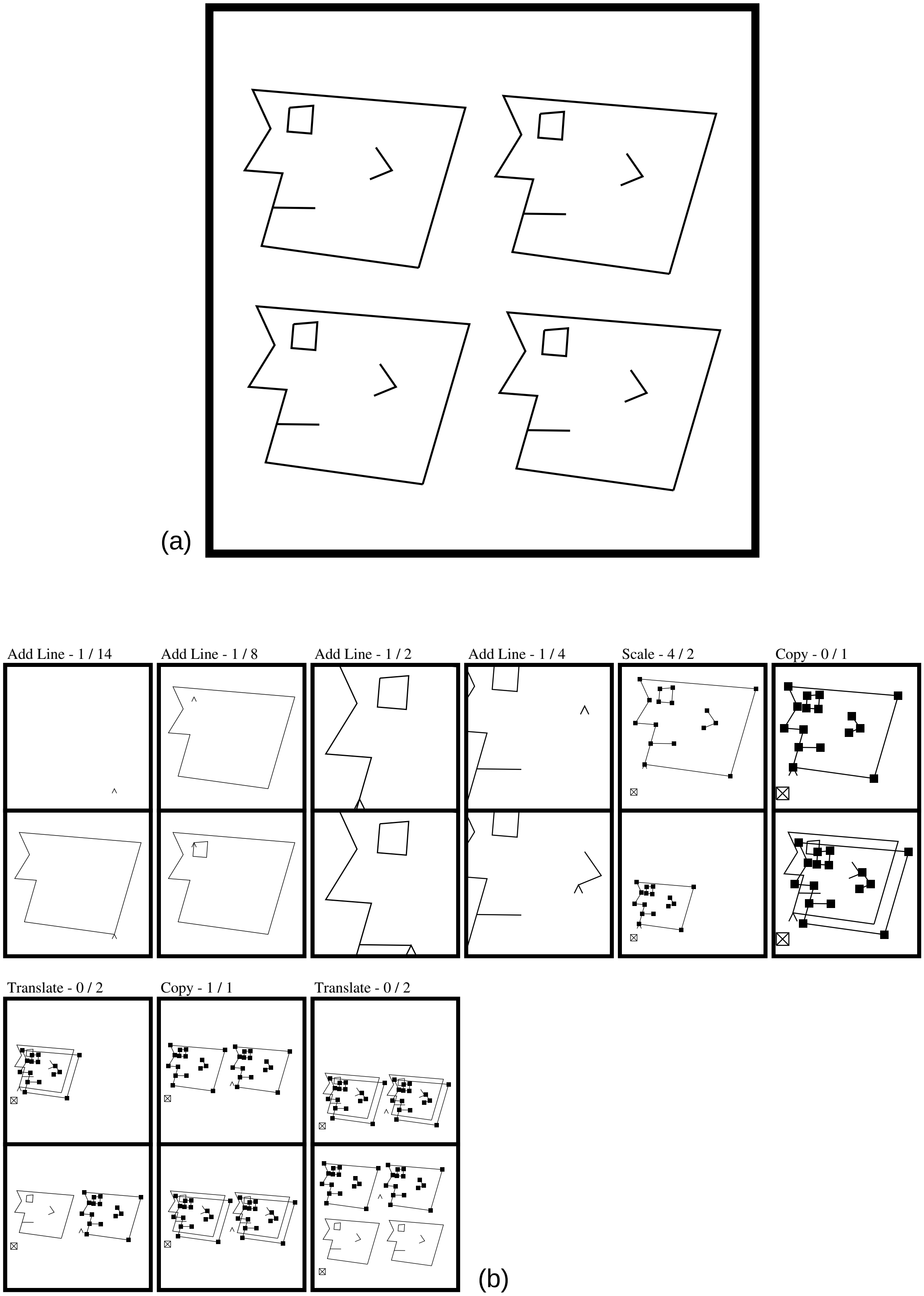


Figure 6.8 Four heads from 1988. (a) Chimera editor scene; (b) its graphical history.

This history was generated in 1988, and it represents Chimera's initial implementation of graphical histories [Kurlander88b]. As in the current histories, related operations were coalesced into panels. For example, the first panel pair contains several Add-Line operations, as well as operations to move the cursor. Each label contains two numbers, indicating the number of operations represented in the prologue and epilogue. These numbers included lower level operations invoked by the editor itself to accomplish a task, so the numbers could be confusing to users. For example, the first prologue of Figure 6.8a contains 14 operations. The user invoked seven Add-Line operations, and each of these in turn executed a Move-Caret operation, placing the software cursor at the end of the new line. The current histories now reflect only command events, and this should be preferred by end users. However, as described later, the capability of representing arbitrary events can benefit other classes of applications.

These early histories used only a single rendering style, could only depict objects from the editor scene, and were generated without a declarative rule base. Instead of using rules to determine panel content and coalescing, each editor command required two associated functions to support graphical histories: one to return a value indicating how well it coalesced with operations preceding it on the history list; another to search through the scene and find objects to include in panels ending in this operation. The declarative rule base is more elegant, and has been easier to extend, understand, and maintain, as Chimera grew.

As illustrations generated in Chimera became more complex, it became more difficult to quickly differentiate between the contents of the before and after panels. We needed a method to draw one's eye to the important parts of the panel, away from auxiliary scene context. To accomplish this, we added rendering styles to the graphical histories. As in the current histories, objects chosen for inclusion are tagged with their reason for inclusion, or equivalently, their role in the explanation. These tags are examined by the renderer to determine what style should be used to draw the objects. This represents the state of the histories in 1989, and these histories are further discussed in [Kurlander90].

This technique was employed in producing the history panels of Figure 6.9 to make the important objects in each panel stand out. Figure 6.9a shows a bar chart drawn in Chimera, and Figure 6.9b shows steps that added the bars to the chart, and a command that changed the stroke width of the frame. The bar created in the first panel pair is an important result of the operation, so it is rendered in its normal fashion, while the rest of the chart, which is present only to show where the bar was added, has been subdued. In this and in subsequent panels, it is clear which objects are critical to the explanation, and which provide context.

Next it became clear that we needed to abandon the two row layout, and adopt a single row of panels. There were many reasons for this decision:

- Most importantly, two rows of panels require far too much screen real estate.

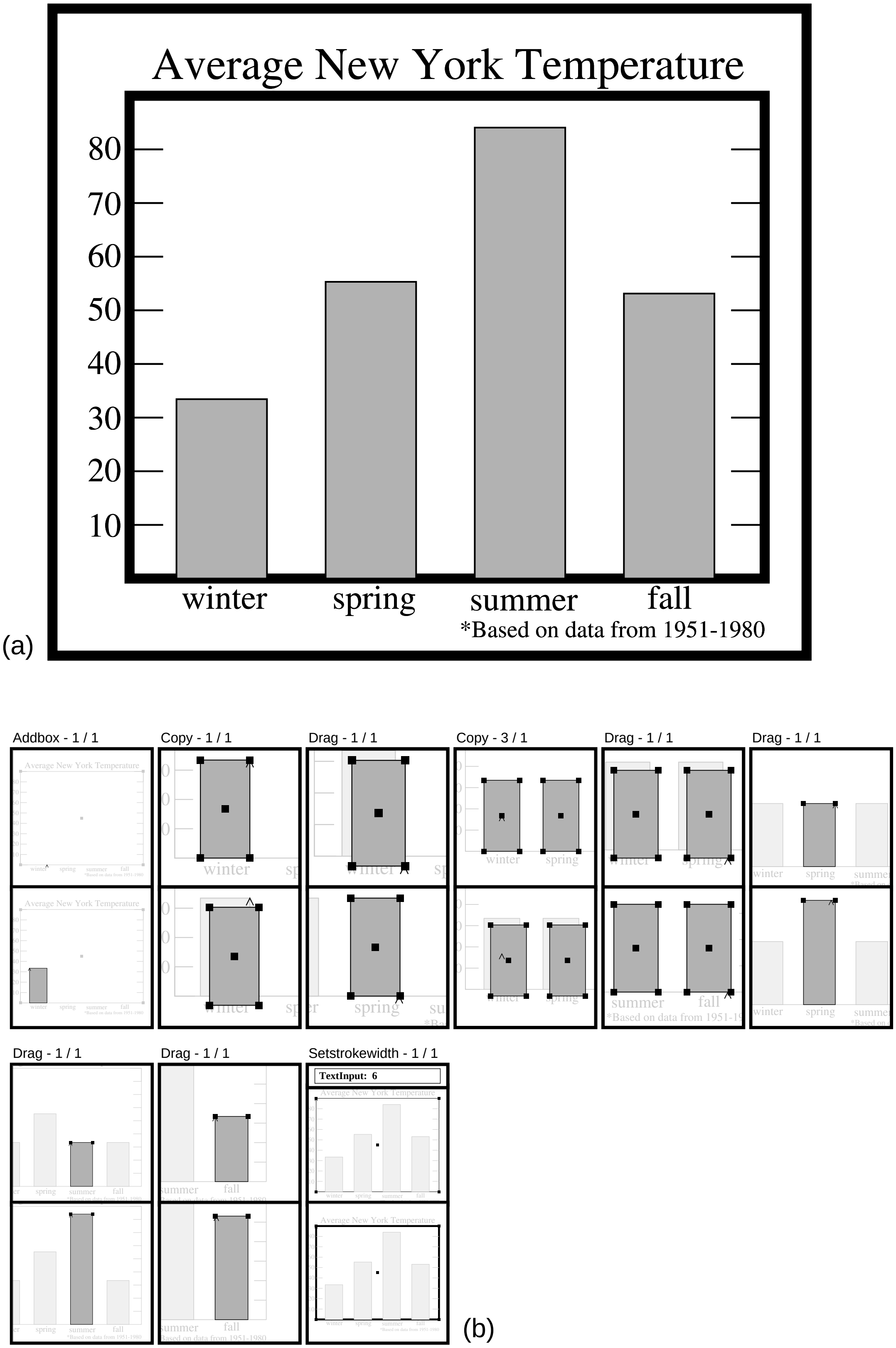


Figure 6.9 Bar chart from 1989. (a) Chimera editor scene; (b) its graphical history.

- Often information in the prologue is unimportant to the user. For example, since the Set-Fill-Color operation acts on selected objects, it is not necessary to have a prologue panel to show by comparison which objects changed.
- There is a tendency for prologues to repeat information. Because there is a large amount of coherence in editing tasks, the prologue of one pair is often identical to the epilogue of the previous pair.
- Having a single row of panels results in faster drawing time.
- A single row of panels is more intuitive to read. When asked to interpret sample histories for the first time, most people try to read across the first row, then across the second. This is incorrect. (However, it had been suggested that we could turn the history on its side).
- Prologues are less important now that different rendering styles distinguish important, participatory objects from context.

Another important decision was to move the histories out of the graphical editor component of Chimera, into its UIMS layer so that other graphical applications can take advantage of the facility. This task is only partially completed, but we are already reaping some of the benefits. One of our goals is that there be no added overhead on the application writer to get reasonable graphical histories for most applications. The system defaults to one operation per panel, unless the application writer provides a grammar to show how related operations should be coalesced into a single panel.

Eventually, we would like the history mechanism to make a good guess about which graphical objects and widgets should be included in each type of panel, without the explicit rules that must be provided now. These would be the objects and widgets modified or referenced by the operations in the panel, and the UIMS should be able to detect this. However, the application writer needs to be able to override the default, and explicitly specify the contents for each kind of panel.

Initially the histories were editable only in the sense that the user could explicitly execute a command to undo the editor to a state represented by any panel, make changes to the editor scene, and redo the undone commands. This process often felt indirect and clumsy, so we added an option to convert the history panels into fully editable Chimera scenes. Now changes can be made directly in the panels themselves, and users can execute a command to propagate these changes into the history stream.

Though editable graphical histories were initially developed with the intention of using them as a visual representation for macros by example [Kurlander88b], we postponed implementing this capability until the individual panels could be made editable. Having editable panels allows object and property arguments to be selected directly in the panels, and permits macros to be edited in place. Editable graphical histories support macros by example in many other ways as well, as is discussed in the next chapter.

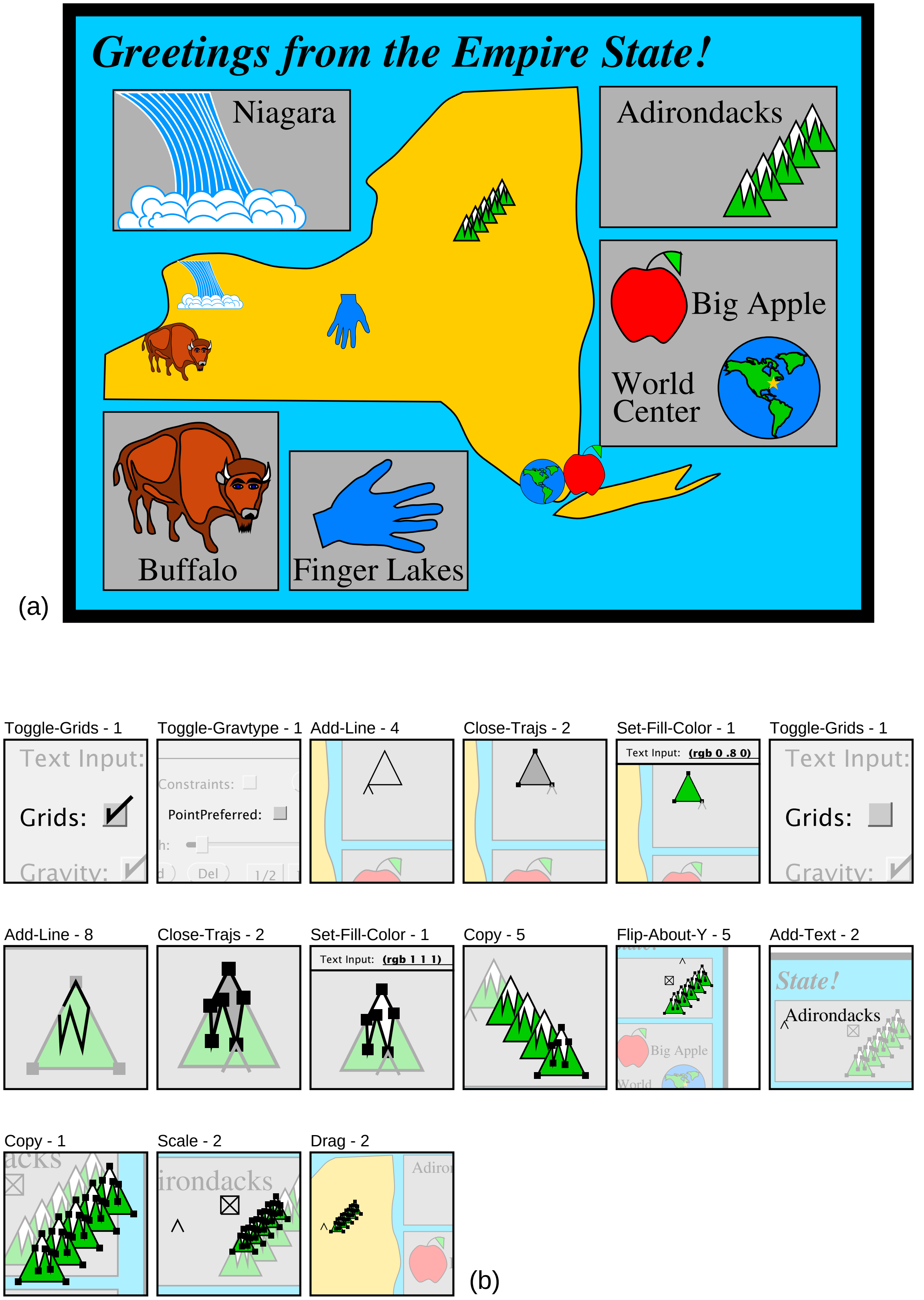


Figure 6.10 Postcard from 1992. (a) Chimera editor scene; (b) its graphical history.

Figure 6.10a is a postcard drawn in Chimera, and Figure 6.10b is the graphical history showing how the Adirondack mountains were added to the illustration. The graphical history here was recently generated by Chimera (March 1992), and reflects the changes that we made over the years. Note that panels are no longer separated into before and after rows. Also, now that the history mechanism is partially in the UIMS, it is much cleaner for the system to generate panels showing parts of the control panels, since the goal of separating applications from interfaces dictates that the editor itself not know about the appearance of these components.

An earlier 1989-class history, corresponding to the last seven panels of this sequence, appears in [Foley90, Plate I.23]. That reproduction is in color, and better demonstrates the effect of lightening colors of contextual objects. Since it represents a nearly identical command sequence, it serves as a good example of how the histories have changed over time.

6.6 Conclusions and Future Work

Editable graphical histories offer a new visual representation for commands in a graphical user interface. They merge aspects of spatial browsing and temporal reexecution to overcome some of the limitations of each. Based on a comic strip metaphor, editable graphical histories display important interface events over a sequence of panels. The histories rely on a number of strategies to make the presentation terser and easier to understand. Consecutive, related operations are automatically coalesced together in panels representing logical operations, rather than physical ones. These panels can be expanded into lower-level panels if the user needs to examine the component operations in detail.

The history mechanism automatically chooses the contents of each panel, rather than relying on screen miniatures or prefabricated icons. It chooses a style for rendering the contents of each panel, based on its role in the explanation. The history panels rely on the native visual language of the interface to convey the changing state of the application interface, rather than using special meta-symbols, whose meaning would have to be learned. This makes these histories accessible to all users of the interface.

Though editable graphical histories allow users to browse through the operations that they have performed in a session, and share this information with collaborators, they also serve as an interface for undoing and redoing sequences of operations. Users can restore the editor state to the point in time represented by any panel, and they can delete sequences of panels from the history. They can select sets of operations to be redone, or they can redo undone operations. Furthermore, the history panels can be made editable, and be replaced with miniature Chimera editor scenes. Users can modify the contents of these panels, and propagate their changes into the history stream.

These histories have been implemented as part of Chimera, and they have been evolving since 1988. Though they currently work in conjunction with a graphics and interface editor, they could readily be applied to other applications. To experiment with this, the editable graphical history implementation is being extracted from Chimera's editor subsystem, and transferred to its UIMS. A result of this is that editor panels can now show the contents of windows other than the editor canvas.

Several issues regarding these histories need further investigation. Currently, Chimera cannot visually represent changes to a history sequence within the sequence itself. There is no record of panel deletions, and after new operations are added to the middle of a history, there is no record that this was done pursuant to the original operations. This Orwellian, *1984*-based approach to history modification should be rectified by developing a special visual notation for these history edits.

On the topic of science fiction, editable graphical histories allow a variation of the *grandfather paradox*. The grandfather paradox is a contradiction introduced by time travel: if time travel were possible, then what would happen to you if you were to go back in time and shoot your grandfather during his youth? Essentially editable graphical histories permit a kind of time-travel—the user might go back in time and shoot... himself in the foot! For example, what would happen if the user created a rectangle, copied it, and deleted the creation of the original? Care must be taken to avoid conflicts such as this. Immediately prior to history edits, Chimera saves a copy of the scene and history in case the user introduces conflicts. When a conflict arises, Chimera offers to restore the original scene and its history. A better approach might involve statically analyzing each potential edit, to identify dependencies and conflicts in advance. If the user tried to delete a set of selected panels, the system could highlight all dependent panels (and their dependencies, recursively), and offer to delete them all. Another alternative would have the system treat panels containing conflicts as though the conflicts were never introduced. For example, if the user deletes the creation of a rectangle from the history, any copies of that rectangle would be unaffected. Chimera should allow users to choose from all these alternatives.

Chimera's mechanism for choosing landmark objects is primitive, and could be improved. Good landmarks have a unique visual appearance, but Chimera does not ensure that its chosen landmark is unique. Chimera could use graphical search to determine whether a prospective landmark does in fact have a unique visual representation. Even an object that is a good landmark when a panel is created may become a poor choice if the user adds similar objects to the scene. We could experiment with optimizing the appearance of earlier panels as the history grows. Graphical histories can become quite long. Graphical search and other search mechanisms might also be used to find a history panel satisfying user-specified criteria.

In our graphical editor domain, history panels record only user-initiated commands. Now that we are moving the history mechanism into the UIMS, the histories will be able to record interesting non-interactive events as well. Predominantly non-interactive graphical programs might generate graphical logs, rather than interaction histories, to record the changing state of their display.

Though picture panels can visually represent operation sequences performed in the past, they can also depict sequences that might be invoked in the future. Such futures (as opposed to histories) could be used by a system that automatically infers likely interaction paths in presenting potential steps to the user. They could be used as part of an active tutorial system for an application, to teach people how to reach probable targets. Alternatively, these futures might act as a type of power steering by experts to reach their goals faster.

Several interesting representational issues present themselves that could also be the target of future research. It would be helpful to investigate how histories might represent multiple applications, sessions, or users. Additional methods could be developed to represent these histories more compactly and facilitate navigation through them. For example, users might want to coalesce arbitrary sequences of commands that they associate together. It would be interesting to try to automatically increase the granularity of panels farther in the history, creating a graphical *fisheye view*. Fisheye views distort space, allowing more detail for important parts of a presentation, and less for context [Furnas86].

"I respond to three questions," stated the augur. "For twenty terces I phrase the answer in clear and actionable language; for ten I use the language of cant, which occasionally admits of ambiguity; for five I speak a parable which you must interpret as you will; and for one terce, I babble in an unknown tongue."

"First I must inquire, how profound is your knowledge?"

"I know all," responded the augur. "The secrets of the red and the secrets of the black, the lost spells of Grand Motholom, the way of the fish and the voice of the bird."

"And where have you learned all these things?"

"By pure induction," explained the augur. "I retire into my booth, I closet myself with never a glint of light, and, so sequestered, I resolve the profundities of the world."

"With all this precious knowledge at hand," ventured Guyal, "why do you live so meagerly, without an ounce of fat to your frame and these miserable rags to your back?"

The augur stood back in fury. "Go along, go along! Already I have wasted fifty terces of wisdom on you, who have never a copper to your pouch."

— Jack Vance, *The Dying Earth*

Chapter 7

A History-Based Macro By Example System

7.1 Introduction

When applications are made *extensible*, the entire user community benefits. Individuals can customize their applications to the tasks that they often encounter, and experts can encapsulate their expertise in a form that less skilled users can exploit. By writing a macro or program, users can extend an application to perform tasks not included in the original interface; however this typically requires both programming skills and familiarity with the application's extension language. Systems with a *macro by example* or *programming by example* component generate code automatically in response to tasks demonstrated by the user through the application's own interface. These systems make the benefits of extensibility accessible to the entire user community.

Many applications, such as GNU Emacs [Stallman87], have a macro by example facility, but lack a visual representation for the macros. Without a visual representation, it is difficult to review the operations that compose the macro. When there is an error in such a macro, the macro must be demonstrated once again from scratch. If an error occurs in a macro without a visual representation, the system cannot provide a comprehensible error message explaining which step generated the error.

Though visual representations are clearly important for a macro by example facility, many systems omit this component since it is problematic how to statically display commands executed through an application's graphical user interface. The last chapter described editable graphical histories, which provide such a representation in Chimera. This chapter investigates a macro by example facility that uses editable graphical histories as its visual representation, and discusses the many ways that the macro facility takes advantage of these histories.

In the next section we discuss how other example-based systems have dealt with the issue of visual representation, and in the rest of the chapter focus on how editable graphical histories support a macro by example facility.

7.2 Related Work

Since most programming by example research has dealt with problems other than representation, many systems ignore this issue. Peridot [Myers88] and Metamouse [Maulsby89a] provide highlighting or feedback for individual program steps, however they depict a single step at a time with no visual representations for the complete procedures that they infer. A more comprehensive graphical representation would allow the user to quickly examine and edit any step.

Representing commands in text-based systems tends to be easier, since the textual commands themselves form a convenient representation. Tinker, a text-based programming by example facility, has a textual audit-trail of steps used in constructing procedures [Lieberman86]. To edit the demonstrated procedure, the user can either textually edit these steps or the resulting Lisp procedure. Tweedle, a graphical editor with both a WYSIWYG view and a textual code view, allows procedures to be generated in both views [Asente87]. However, to edit a procedure, the user must be able to understand the code view. In the MIKE UIMS, graphical macros can also be defined by demonstration [Olsen88]. In this system macros can be defined and edited largely in demonstration mode, but the visual representation of graphical commands is textual.

A programming by example component of SmallStar, a subset of the Star user interface, adopts a mixed text and iconic representation for macros [Halbert84]. As shown in Figure 2.4, this system uses a predefined set of icons or pictographs to represent entities on the desktop. The domains for which our system is targeted are more graphical in nature, so prefabricated icons will not suffice. As will be discussed in more depth in the next section, our approach is to generate graphics automatically to represent the operations in the macro. The Mondrian graphical editor also dynamically builds panels to present a history, but Mondrian's panels are uneditable screen miniatures, and do not yet participate in many phases of macro definition [Lieberman93b].

Most of the programming by example systems discussed thus far have special operations to start and stop recording events. In our system, operations are always being recorded by an undo/redo mechanism. When users realize that a set of operations that they had performed are generally useful, they can always open up a history window and encapsulate the interesting operations into a macro. A programming by example system named EAGER also generates macros from a history [Cypher91]. It constantly monitors the command stream for repeated operation sequences. When a repetitive task is detected, the system presents feedback that indicates the tasks it anticipates, and when users are confident in EAGER's predictions, they can have it automatically generate a generalized procedure. However, this procedure has no graphical representation.

7.3 Macro Definition

Macro definition in Chimera consists of two primary passes. In the first pass the task is demonstrated using the regular user interface. The dialogue for this pass is indistinguishable from regular user-interaction—there are no special commands to execute, and no special operations to start and stop macro recording. Since people often do not think of defining a macro until they have executed the steps at least once, the commands may already have been demonstrated, and no additional repetition is necessary.

In the second pass, the demonstrated sequence of commands is supplemented with additional information to convert this sequence into a macro. The commands executed in this pass are different than those forming the ordinary application dialogue. This pass includes selecting a set of previously executed commands to encapsulate into the macro, selecting arguments for the macro, generalizing the commands to work in other contexts, and debugging and saving the macro. Splitting macro definition into a demonstrational step and a generalization step was first done by Halbert in SmallStar [Halbert84]. It has the advantage that the demonstrational pass of the macro is purely demonstrational, and certain constructs, such as conditionals and loops, which are difficult to add by demonstration, can be introduced in a separate non-demonstrational pass.

7.3.1 The Demonstrational Pass and Operation Selection

Unlike SmallStar which had special commands to start and stop recording a macro, commands in Chimera are always being recorded by an undo/redo facility. At any time, users can open up the history window, review the commands executed in a session, or undo and redo some of these commands. They can also select a set of commands to be incorporated into a macro.

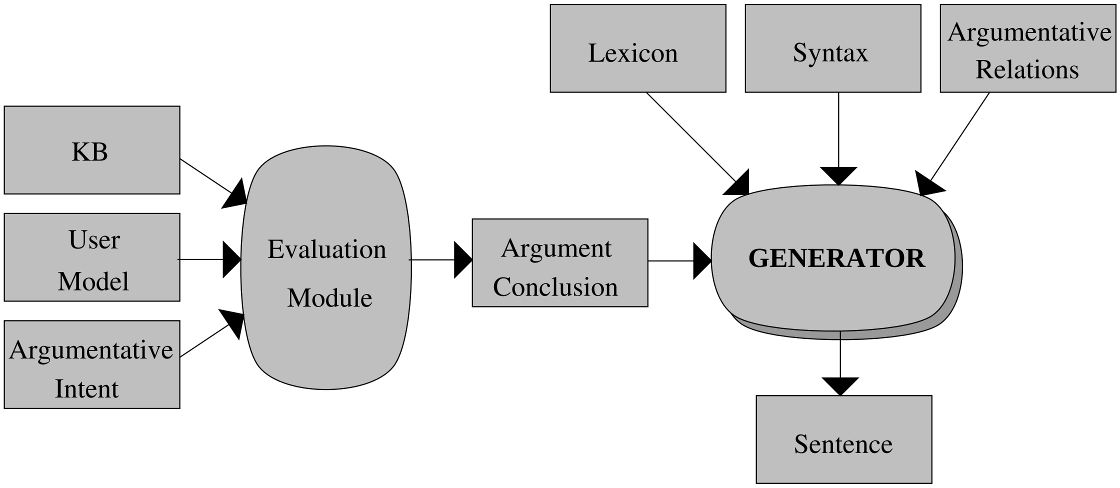
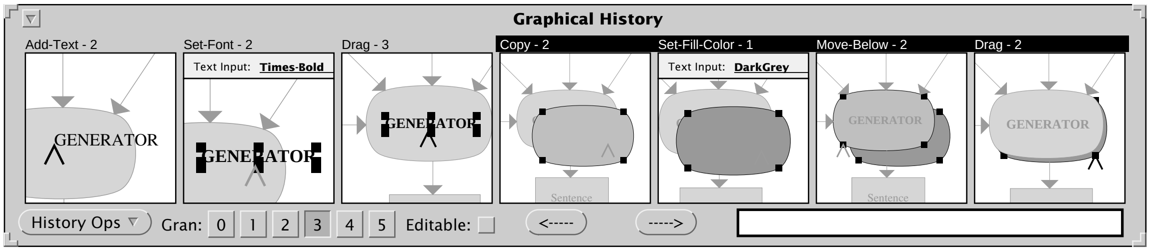


Figure 7.1 A technical illustration created with Chimera.

Figure 7.1 shows a diagram drawn with Chimera. After creating a drop shadow for the horizontal oval labeled GENERATOR, we realize that other scene objects will need drop shadows as well, so we decide to create a macro. The steps performed creating the first shadow need not be wasted. We can open up a graphical history, and use these steps as the starting point for the macro definition. Figure 7.2 displays the graphical history representation of the commands that added text and a drop shadow to the horizontal oval in the diagram.

In the first panel we added the text string to the oval, in the second we changed the font of the text string. The third panel shows the text being centered, and in the fourth panel the oval is copied. The fifth panel sets the copy to have a darker shade of grey, the sixth moves it below the original, and in the last panel the copy is moved to have a different offset.



Text Input: Times-Bold

Text Input: DarkGrey

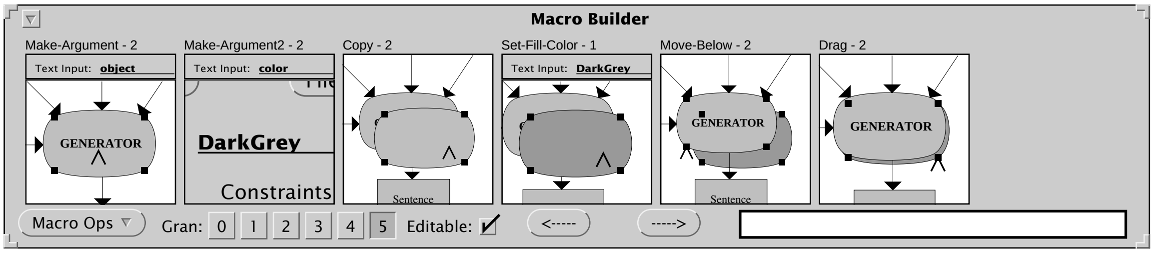
Figure 7.2 A graphical history representation of steps that add text to an oval and create a drop shadow. These steps were used in creating part of Figure 7.1. Panels whose labels are shown in reverse video have been selected by the user to create the macro shown in Figure 7.3.

In the first additional step necessary for macro creation, we select those panels from the history that contributed towards adding the drop shadow. These are the last four panels of Figure 7.2, and in this figure they appear selected. The *macroize* operation, which is executed next, takes a panel selection, and opens up a new Macro Builder window on these panels. This window initially contains only those panels that were selected in the graphical history.

7.3.2 Argument Declaration

As the next step, we declare the arguments to the macro. We do this by selecting the arguments where they appear in the panels, and providing them with names. To select a component of a panel, we first have to make the panels *editable*. This is done by checking the editable box at the bottom of the Macro Builder, which has a similar effect to checking the box at the bottom of a history viewer—it replaces the static graphical representations of the panels with fully editable graphical canvases. Objects in these canvases can be selected and manipulated in the same manner as objects in a regular scene. The first argument to this macro will be the object that casts the shadow. We examine the panels in the macro builder window for an instance of the original oval. This oval appears in each of the panels, so we select any one of these instances, give it the name “object” by typing this name in the Text Input widget of the control panel, and execute the Make-Argument command.

A panel is added to the beginning of the history, depicting the argument selection. Argument declarations are placed at the beginning of our macros, just as they appear at the beginning of traditional procedures. The resulting panel is the first of the sequence of panels depicted in the Macro Builder of Figure 7.3. The argument declaration panels show the arguments as they appear before the operations in the macro were invoked, plus additional scene context. They are not just copies of the panels that were used for selecting the arguments. Scene objects that do not exist at the beginning of the macro, such as the oval produced by the copy operation in the third panel of Figure 7.3, are not plausible arguments, and Chimera will not allow them to be used for this purpose.



Text Input: object

Text Input: color

Text Input: DarkGrey

Gran: 0 1 2 3 4 5

Editable: ☑

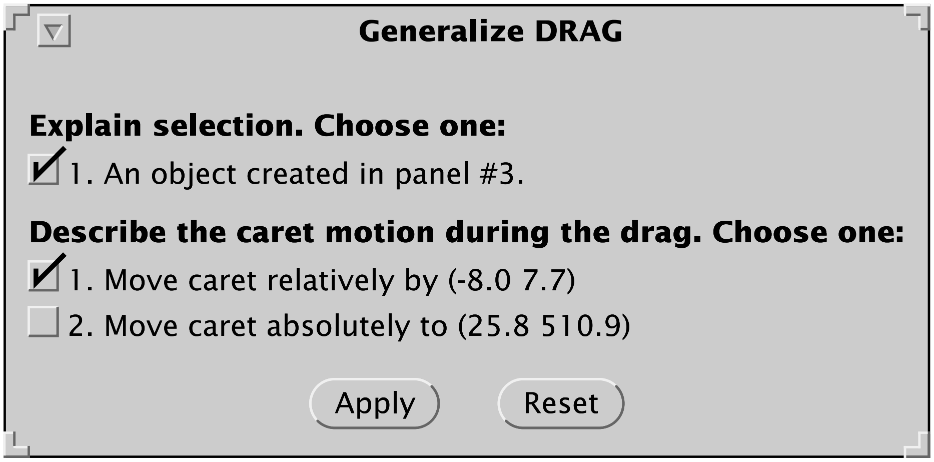
Figure 7.3 Macro Builder window containing operations to add a drop shadow to an object.

In addition to declaring graphical objects as arguments to a macro, we can also choose graphical properties such as color or linestyle. To choose a graphical property, we can select from the history a widget in which this property is displayed. Widgets can be selected just like any other graphical object. For this macro, we would like the color of the drop shadow to be a second argument. First we locate the panel in which we specified the color. This is the Text Input field of the fourth panel of Figure 7.3. We select this widget, and give the second argument the name "color". A new argument declaration panel is created, which is the second panel of Figure 7.3.

7.3.3 Generalization

Next it is important to generalize the macro operations to work in new contexts. This generalization can be either specified by the user, or inferred by the system with the help of a built-in inference engine. Chimera has been supplied with a set of different interpretations for its editor commands, as well as heuristics for distinguishing when each interpretation is likely. These interpretations are based on the generalizations of three classes of command arguments, and are described in detail in Appendix D. When choosing a default generalization of a command, the system evaluates the heuristics in the context of the graphics state to produce an ordered list of possible intents. The user can view the system's generalizations and override them if necessary. Once again the graphical history representation is useful as a means of selecting panels, this time for choosing panels to be generalized.

For example, after selecting the last panel of Figure 7.3, we execute the Generalize-Panel command. The window shown in Figure 7.4 appears, containing the various generalizations that the system considered plausible in the given context, with the most likely interpretation selected. The generalizations of all the operations contained in a panel can be



Generalize DRAG

Explain selection. Choose one:

☑ 1. An object created in panel #3.

Describe the caret motion during the drag. Choose one:

☑ 1. Move caret relatively by (-8.0 7.7)

☐ 2. Move caret absolutely to (25.8 510.9)

Figure 7.4 A form showing the system's generalizations for the last panel of Figure 7.3.

viewed and modified at once. This panel contains the selection of the drop shadow, and the subsequent translation of the shadow to lie at the appropriate offset under the original object. Only one of the built-in interpretations for the selection is valid in the context of the last panel: that the object selected at this step is the object created in the third panel.

The Generalize-Panel command need not be executed explicitly for every panel in the macro. Another command can be used to set or reset all panels to their default generalization. When the macro is executed, all panels that have never been generalized are automatically given a default generalization.

7.3.3.1 Generalizing a selection

The system has a number of possible interpretations of object selections. As an example of the types of generalizations Chimera is capable of performing, we list the various classes of selection generalizations here. An object may be selected because of the following classes of reasons (which are further described in Appendix D):

- **Argument.** The object is an argument to the macro.
- **Constant.** The object is a constant in the macro.
- **Search Iterator.** The object is the result of a MatchTool search iteration.
- **Component.** The object is a particular component of another object, or a parent of another object. Example: first vertex of a polyline.
- **Temporal Reference.** The object was referenced in a particular macro step. Example: object created in panel #3.
- **Position.** The object shares a particular geometric relationship with another object. Example: the center of a circle.

Selection criteria can also be combined in two ways:

- **Disjunction.** Multiple objects selected for different reasons. Example: an object is selected because it is either argument 1 or argument 2.
- **Composition.** The composition of multiple selection criteria. Example: first vertex of the second segment of argument 1.

This set of selection criteria is by no means complete. For example, a set of objects may have been selected because they share a particular graphical property in common (e. g., the same fill color), and Chimera cannot detect such an intent. Even within the categories above, there are many other selection criteria that we would like the system to consider. For example, it will not propose that an object was selected because it overlaps another interesting object.

7.3.3.2 Generalizing a move

The second checklist of Figure 7.4 explains the system’s generalization of the move or drag operation. There are two possible explanations that fit the bill: a relative translation and an absolute move. In this case, the system chooses the relative translation as most likely. If the dragged objects were moved so that the caret, the software cursor, snapped to an object or an intersection point (of either scene objects or alignment lines), then this would be considered the most likely interpretation. This allows us to define macros that perform geometric constructions, using the snap-dragging interaction technique developed by Bier [Bier86]. For example, we can use this technique to define macros in Chimera that bisect angles, construct the midpoint of lines, and align shapes. Section 7.5 further describes the use of alignment lines in macros.

7.3.3.3 Representing generalizations textually

Our macro facility represents generalizations as textual supplements to the graphical display of commands. Another approach might involve adding graphical symbols to the panels in order to make the system's interpretations of the commands clear. This approach has several problems. If the number of generalizations known by the system is large, then the graphical vocabulary must also be large. Unless the same graphical conventions are used by the system during normal editing, the user would need to learn a new visual language in order to define macros. By representing generalizations textually, in English, our generalizations are accessible to all users of the system. Our approach is similar to that of SmallStar, in which generalizations are displayed as textual data descriptions (see Figure 2.5) [Halbert84].

7.3.4 Macro Archiving and Invocation

After a macro has been generalized, it can be named, saved, and invoked. Currently we save macros with all of the scene state that was present at definition time. This allows subsequent viewing and editing of an editable graphical history representation of the macro that is identical to the panels originally displayed in the Macro Builder window during macro construction time.

To invoke a macro, the user executes a menu command and a macro invocation window pops up on the screen. For the drop shadow macro that we have just defined, this window is shown in Figure 7.5. The window contains an entry for each of the arguments declared previously. The first argument, "object" is assigned two buttons: one to set the argument and the other to show it. The second argument, "color", is a property argument, and Chimera uses a different technique to set and show property arguments. For each property argument, a copy of the widget used to specify the argument during the original macro demonstration is included in the invocation window. Since the Text Input field of the control panel was originally used to specify the color of the drop shadow, this widget is copied and added to the invocation window. As a default, the widget contains the value specified for this parameter during macro demonstration time. Effectively this parameter is treated as an optional parameter with a default. If the user does not change its value in the macro invocation window, the macro will use the value that was chosen during demonstration.

Add Drop Shadow

object:

color: DarkGrey

Figure 7.5 Window for setting arguments and invoking the drop shadow macro.

7.3.5 Testing and Debugging

An important part of programming, demonstrational and conventional alike, is the testing and debugging phase. In the top row of Figure 7.6 we have created four different shapes, and in the second row we apply the drop shadow macro to each, using the colors listed at the bottoms of the columns. To create the drop shadow for the circle, we use the dark grey default color already in the macro invocation window. Next, we change the shadow color to black, and apply the macro to the shape composed of splines and arcs. Then, with the shadow color set to light grey, we apply the macro to the text and finally the Bezier curve.

At this point we notice a bug in the macro. Though we expect the macro to add a light grey drop shadow to the Bezier, the drop shadow is black. To debug the macro, we go back to the original Macro Builder window in Figure 7.3 and examine the commands that it contains. The bug quickly becomes apparent. Though we changed the fill color of the drop shadow, we never changed the line color. On inspecting the results of this initial test again, it is clear that the circle’s drop shadow is incorrect as well since it too has a black line, yet we did not notice a problem at first because the shadow is dark.

The graphical history representation supports editing operations on either macros or the history directly, in place. When the panels are made editable, new commands can be executed directly on the panel objects. These additional commands can be propagated into the history at the point at which they are inserted, by executing the Propagate-Panel-Changes command. When this command is executed, the system transparently undoes all of the operations after the newly inserted operations, executes these new operations, and redoes all of the operations that had been undone. As described last chapter, if commands are added to the history, rather than a macro, the editor scene corresponding to this history is updated according to the changes. In all cases, the subsequent panels of the history are regenerated to take into account the changes that had been inserted earlier.

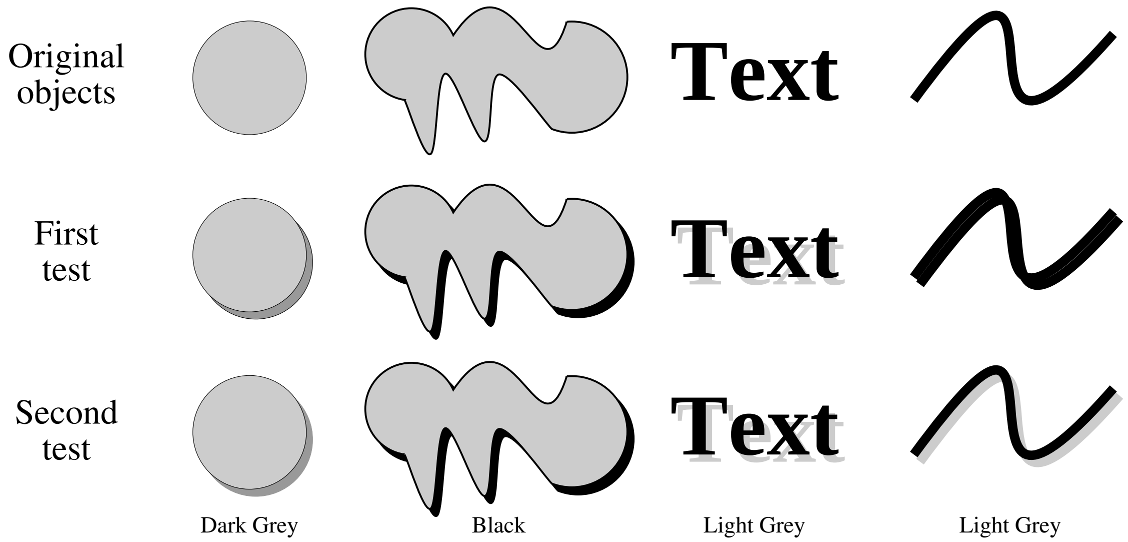
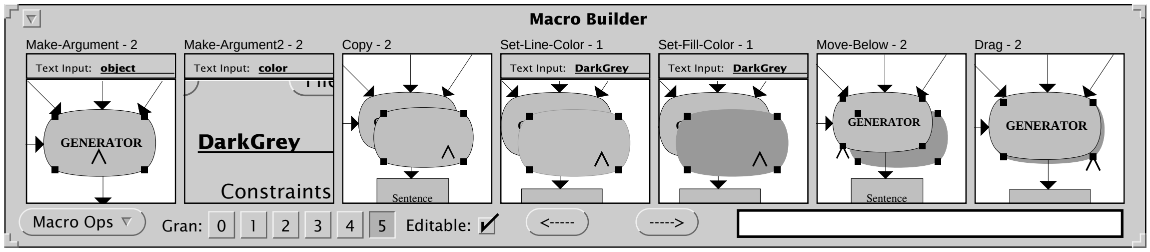


Figure 7.6 Testing the macro. The first row contains a set of test objects. The next row contains the results of invoking the original macro on these objects, using the colors named at the bottoms of the columns. The final row shows the results of invoking the debugged version of the macro.

To fix the bug in the macro shown in Figure 7.3, we need to add a Set-Line-Color operation. To do this, we type DarkGrey in the Text Input widget of Chimera's control panel, and execute the Set-Line-Color command in the third panel of the Macro Builder window where the copy is already selected. Next we select this panel, and execute the command that propagates the newly inserted command into the history just after the copy command. The resulting macro is shown in Figure 7.7. An additional panel was created (panel 4) to represent the newly added operation, and subsequent panels all show the copy with its new line color.



Text Input: object
Text Input: color
Text Input: DarkGrey
Text Input: DarkGrey

Figure 7.7 Final version of the Macro Builder window containing operations to add a drop shadow to an object.

After adding the new Set-Line-Color panel to the macro, we generalize this panel, and execute the macro on our test cases once again. The results, shown in the last row of Figure 7.6, are now as we expected.

Panels can also be deleted from histories or macros. The user can select a sequence of panels, and execute a command that removes these commands as well as any effect that they had. As with command insertion, Chimera must reformulate the panels appearing after the change, taking into account the modified scene.

7.3.6 Adding Iteration

Now we would like to invoke the macro on the other grey-filled rectangles and ovals in Figure 7.1. Instead of having to run the macro separately on each of these objects, we can change the argument references in the macro to refer to the results of a MatchTool search. Specifying an object iterator is similar to declaring an argument. The user selects the object in any panel in which it appears, and executes the Make-Loop-Iterator command. The selected object is then automatically copied by the system into the search pane of the MatchTool, and a dialogue box appears, asking the user to adjust the MatchTool search specification according to the desired search criteria, and to press a button when done. First we delete the initial panel of Figure 7.7, since we no longer want an argument named “object”. Next we select an instance of the original oval, and execute Make-Loop-Iterator. This places the oval in MatchTool’s search pane, and since we want to iterate on all scene objects of this fill color, we select only the fill color checkbox in the search column. After pressing a button to indicate the search specification is complete, we execute a command to have the system choose new default generalizations for the panels.

Chimera automatically predicts the boundaries of loops. It finds all references to the search iterator, and makes sure these are included in the body of the loop. It also extends the boundaries of the loop to include all references to objects created within the loop. This is a heuristic process, and although it typically produces good results, it can in some cases guess incorrectly. In the current example, the first panel of the macro that references the loop iterator directly is the Copy panel, so this is selected by Chimera as the first panel of the loop. The last panel of the macro that applies directly to the loop iterator is the Move-Below panel, so it also is included in the loop. However, the last panel of the macro, the Drag panel, contains operations on an object that was created in a panel that has been designated as part of the loop, so the heuristic also places this final panel within the loop. So Chimera assumes that the loop spans from the Copy panel to the Drag panel, which is what we want. This heuristic for predicting loop boundaries, together with the generalization heuristics presented in Appendix D, form all the heuristics used by Chimera’s macro by example component.

The new macro is ready for a trial run. We invoke the macro on an editor scene in which we have loaded a copy of Figure 7.1, and delete the one drop shadow that we had already created (incorrectly). When we invoke a command to run the macro, a window appears

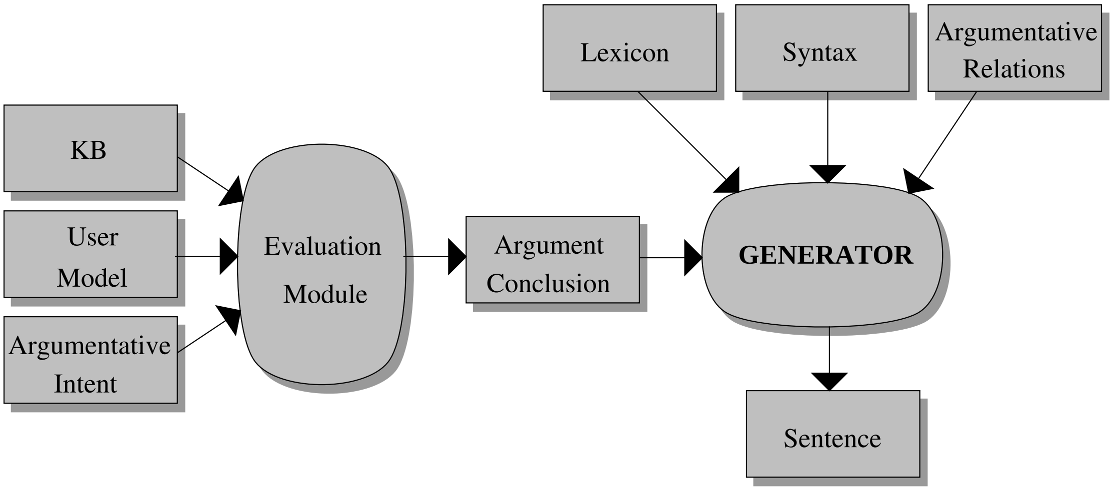


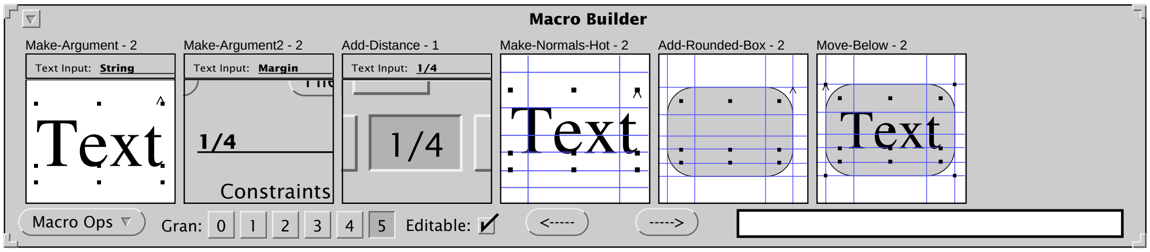
Figure 7.8 Applying the macro with iteration, all grey-filled objects in Figure 7.1 receive drop shadows.

that allows the one remaining argument to be set. We keep the color argument at its default, DarkGrey, and press the Apply button. The result appears in Figure 7.8, and this time the macro works correctly the first time.

7.4 Wrapping Rounded Rectangles Around Text (Revisited)

Chapter 4 describes how to use constraint-based search and replace to wrap a rounded rectangle around text, a task posed by Richard Potter. Chimera's graphical macro facility can also solve this problem. To use Chimera's macros for this purpose, we first demonstrate the task on an example, performing the same steps as we would if we were not automating the task.

Figure 7.9 shows the macro that will be generated after completing all of the steps in this section. Initially we type in a sample text string, or find an existing string to use. Next we type the distance 1/4 (inches) into the control panel's Text Input field, and execute the Add-Distance command to activate distance alignment lines of this measure. This step is represented by the third panel. The value typed in the Text Input widget appears at the top of this panel, and a side-effect, the 1/4 inch radio button created in the control panel to allow these alignments to be turned on and off, appears below. As shown in the fourth panel, we select the text string, and execute the Make-Normals-Hot command to have it generate alignment lines. In the fifth panel we add a rounded rectangle. We specify the placement of the rectangle by positioning its lower left corner at the intersection of 1/4 inch distance lines generated by the left and bottom segments of the text, and its upper right corner at the intersection of 1/4 inch distance lines generated by the top and right segments of the text. Each text object in Chimera has invisible segments along all four of



Text Input: String

Text Input: Margin

Text Input: 1/4

Editable: ☑

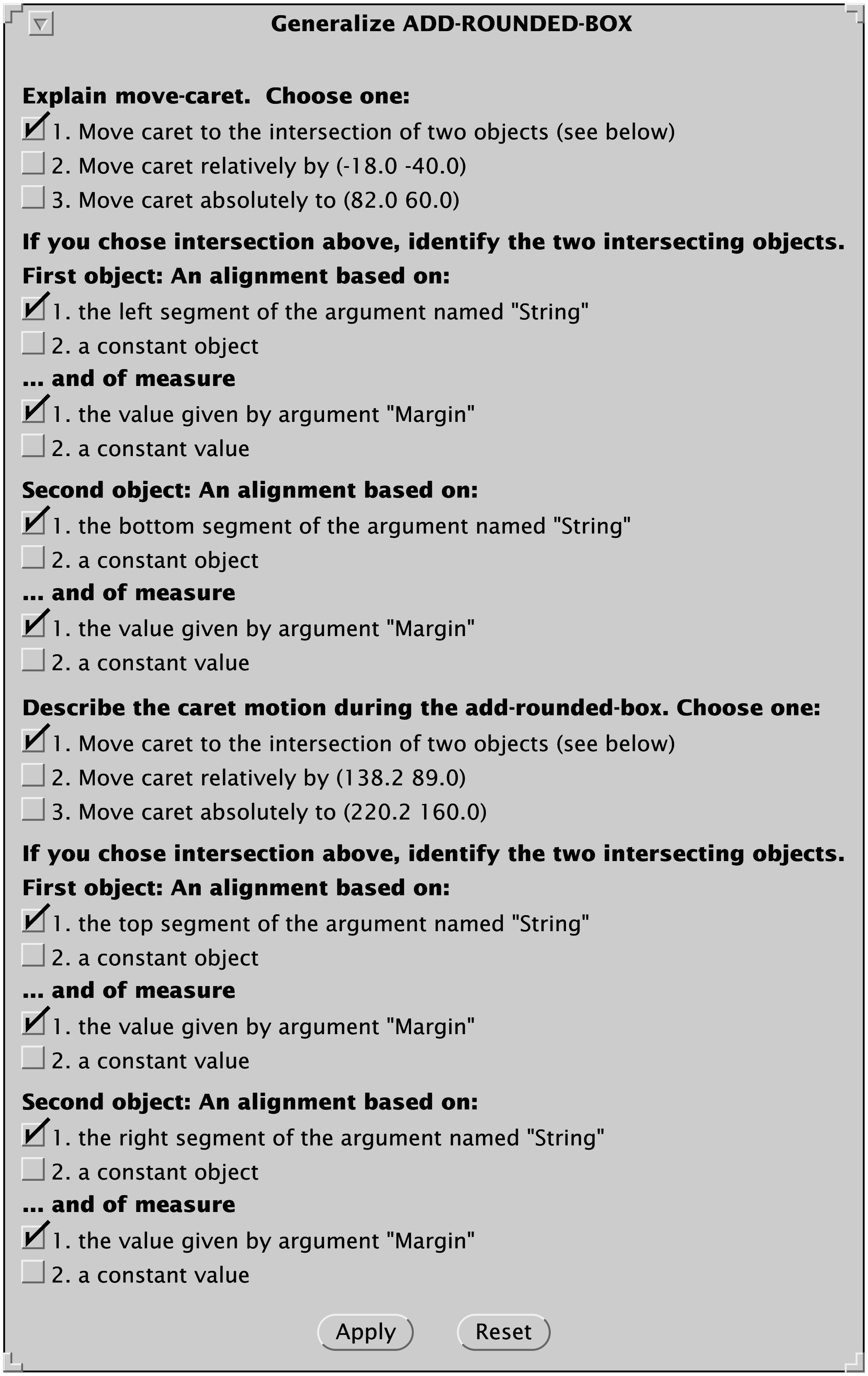
Figure 7.9 A macro to wrap rounded rectangles around text.

its edges, as well as its baseline. As in this example, alignment lines generated by these segments allow other scene objects to be precisely positioned around the various sides of the text object. In the final panel, we extend the selection to include the newly drawn rounded rectangle, and execute a command to move the rectangle behind the text.

To generate the Macro Builder window of Figure 7.9, we selected these four panels from the history, and executed the *macroize* command from the menu. We also declared two arguments, shown in the first two panels: the *String* argument specifies the text around which the macro should place the rounded rectangle, the *Margin* argument specifies the gap between the text and the rectangle.

Next we can examine the system's default generalizations for the panels, and change them if necessary. Figure 7.10 shows the generalizations for the penultimate panel. In this panel we added the rounded rectangle to the scene. To do this, we first moved the caret to the intersection of two alignment lines. The system correctly inferred that we moved the caret to the intersection of two objects: an alignment based on the left segment of the argument named *String*, with a measure (distance) given by argument *Margin*, and another alignment based on the bottom segment of *String*, also of distance *Margin*. The system also correctly guessed that we dragged the caret representing the other corner of the rectangle to the intersection of two more alignment lines: both of distance *Margin*. One was based on the top segment of *String*, and the other was based on the right segment of *String*.

As this example shows, the generalization window's explanation can be somewhat difficult to understand. The explanation of the initial caret movement is split between five different sets of checkboxes. The reason for doing this rather than having a single list of generalizations is to reduce the size of the generalization window, and make it faster for the user to read and set. If this Move-Caret operation were represented by a single checklist, then it would have 18 entries, and these entries would tend to be long. The reason for this is that each alignment has two orthogonal properties with two values, leading to four combinations. Since the caret can be moved to the intersection of two alignment lines, this yields 16 potential explanations. Add to this the possibilities of a relative or absolute move, and this yields 18 possibilities.



Generalize ADD-ROUNDED-BOX

Explain move-caret. Choose one:

☑ 1. Move caret to the intersection of two objects (see below)

☐ 2. Move caret relatively by (-18.0 -40.0)

☐ 3. Move caret absolutely to (82.0 60.0)

If you chose intersection above, identify the two intersecting objects.

First object: An alignment based on:

☑ 1. the left segment of the argument named "String"

☐ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"

☐ 2. a constant value

Second object: An alignment based on:

☑ 1. the bottom segment of the argument named "String"

☐ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"

☐ 2. a constant value

Describe the caret motion during the add-rounded-box. Choose one:

☑ 1. Move caret to the intersection of two objects (see below)

☐ 2. Move caret relatively by (138.2 89.0)

☐ 3. Move caret absolutely to (220.2 160.0)

If you chose intersection above, identify the two intersecting objects.

First object: An alignment based on:

☑ 1. the top segment of the argument named "String"

☐ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"

☐ 2. a constant value

Second object: An alignment based on:

☑ 1. the right segment of the argument named "String"

☐ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"

☐ 2. a constant value

Figure 7.10 Dialogue box to view and set generalizations for fifth panel of Figure 7.9.

All of Chimera's generalizations for this panel and the other panels in the macro are correct. Now we can try out the macro on test data. Figure 7.11 shows the result of applying the macro with several different arguments. This time we defined the macro correctly the first time, so no debugging is necessary.

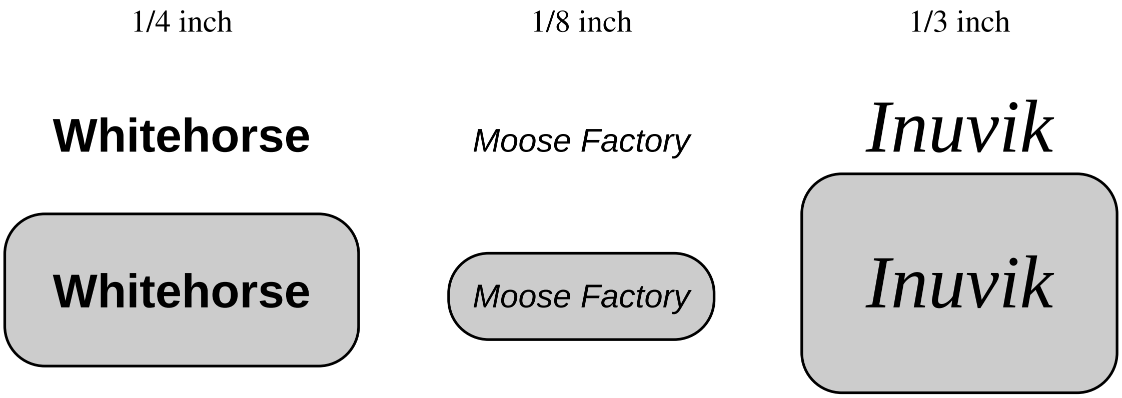


Figure 7.11 Testing the rounded rectangle macro. Three examples are shown in three separate columns. The *Margin* arguments appears in the first row, the *String* arguments in the second row, and the results in the third row.

7.5 Raising the Abstraction Level with Alignment Lines

In the last example, the use of alignment lines made it clear to the macro mechanism that the positioning of the rounded rectangle was likely intended to be relative to the text. If we had positioned the rectangle by eye, or used grids, then the inference mechanism would need to be more sophisticated to determine the relationship between the two objects. There are two approaches that can be taken for building a macro by example system. One might build a system with a powerful mechanism for inferring high-level abstractions, or a system in which these high-level abstractions can expressed through commands. In the latter case, the inferencing mechanism can be weaker, but in the former, the command set can often be reduced.

Peridot has a special facility for inferring the geometric relationships between two objects [Myers88]. Chimera could be extended to allow the system designer *or* end user to define constraint-based search and replace patterns that specify arbitrary interesting geometric relationships to which Chimera's macro inferencing component should be sensitive. For example, in the macro just defined, we created a rounded rectangle, centered around one of the macro's arguments. Using constraint-based search and replace, we can currently write a rule that finds rounded rectangles nearly centered around text, inferring that they should be centered. Perhaps the macro facility could be extended to use this rule, searching for this relationship as new objects are created and moved, to infer that the placement of the rectangle relative to the text is important in this example.

Currently in Chimera, such relationships must be explicitly demonstrated through the use of alignment lines. Alignment lines are a powerful and easy-to-use mechanism for specifying precise geometric relationships. Users of the Gargoyle editor, responding to a survey, indicated that they generally make use of alignment lines, and an analysis of interaction logs confirmed this [Bier88]. If people use this technique to establish precise geometric relationships during normal editing, then they would probably use it, without additional thought, while demonstrating a macro. In Metamouse, users construct tools in order to make positioning operations explicit [Maulsby93a]. In Chimera, alignment lines act as tools, though users can also build tools by hand. Alignment lines appear to have several advantages over tools constructed out of ordinary scene objects:

1. Alignment lines have been shown to be useful in ordinary editing. There is no need to learn their use explicitly for defining macros or programs.

2. There are several different kinds of alignment lines, parameterized by slope, angle, or distance. These parameters can be easily made into parameters for the macro. For example, above we used 1/4 inch distance lines to create a margin of this size, and later we specified that this measurement be one of the macro’s arguments.

3. Most alignment lines are infinite in length, so the tool used during demonstration will handle new contexts encountered during execution. For example, if we are aligning objects along an alignment line in a demonstration, we will be certain the line will be long enough to handle arbitrary objects.

4. Since alignment lines are measuring devices, they can be trivially generated to precise dimensions. Typically more effort is required to construct scene objects with precise measurements. When alignment lines help to define macros, the measurement devices themselves act as the tools.

Another example of alignment lines in a graphical macro appeared in Section 1.4.5, where they assisted in left-aligning two rectangles. One potential disadvantage of employing alignment lines as tools is that these lines cannot be created with arbitrary shape.

Incorporating alignment lines in the interface allows the end user to specify geometric relationships explicitly. So it raises the *abstraction level* of the interface to be more in line with the classes of tasks typically performed by graphical editor macros. Constraints could also be used for the same purpose. Chimera might also plausibly raise the abstraction level higher by including a command specifically to create an object centered about another.

However, it would be unrealistic to expect Chimera to provide prefabricated commands for all high-level intents—that's the whole motivation behind macros! Also, as Myers points out in [Myers88], implementing commands for many geometric positioning intents would result in a proliferation of commands for the user to memorize. In interface design, it is good practice to choose an abstraction level high enough to permit users to accomplish the tasks they need to perform without excessive steps, but low enough so that the operations are generally useful, and users can find and remember the commands they need. When these commands are at a high enough level to convey the user's intent, yet low enough to be exploited for many purposes, and are generally useful within the non-programming component of the application, then they are a real win. Fortunately, this is the case with alignment lines.

7.6 Conclusions and Future Work

We have developed a graphical history representation that supports macro construction in a variety of different ways. The graphical history representation allows people defining new macros by example to review the commands that they have performed. Others who were not present during macro definition time can examine the contents of a macro. The commands are displayed graphically, in the same visual language as the interface itself, thus people who have used the system for ordinary editing can understand the macro representation.

The history representation provides a means of selecting operations. This is useful for two different steps of macro creation. At any time, the user can scroll through the history, and select out useful commands for a new macro. Accordingly, Chimera needs no additional commands to start and stop macro recording. Later, a user may want to view or change the generalizations associated with a set of panels. Again, the graphical representation can be used to select the appropriate panels.

The macro representation makes it very easy to select arguments. After selecting the checkbox that makes the panels of a macro editable, we can select objects directly in the panels and turn them into arguments. Graphical properties can also be turned into arguments by selecting the widgets that set these properties from the macro panels.

Macros are not always defined correctly the first time, and the histories present an interface for editing commands. New commands can be inserted by invoking additional commands in the editable panels, and executing a command that propagates the changes. Unwanted commands can be removed by selecting panels and deleting them.

The macro system itself often refers to the representation when communicating important information to the user. During the generalization process, the interpretation of a command may refer back to steps made at an earlier point in time. For example, the system often needs to refer to an object that was created in a particular panel, or a measurement that was made at a certain step. Macros can generate run-time errors if they are invoked on objects of the incorrect type. Chimera also uses the macro representation to indicate which panel of the macro generated an error.

In summary, graphical histories currently facilitate macro definition in five different capacities:

1. Reviewing macro contents

2. Selecting operations
    - to encapsulate in a macro
    - to set and view generalizations

3. Selecting arguments

4. Editing macro contents
    - inserting operations
    - deleting operations

5. Referencing operations
    - during generalization
    - in error messages

In the future, the graphical histories representation might be extended to support macro definition and testing in several other ways. We could use this representation to show, step by step, the effects of applying a macro to new arguments. By placing together multiple macro viewers vertically, aligned so that similar panels are registered together, we could easily compare the effects of applying a macro to different argument sets, and quickly find the panel in which one of the macros generates unexpected results.

We could also use the panels to specify additional command generalizations when the existing ones fail. For example, one step in a macro might require reducing a box's width by half. If the system is incapable of inferring the desired intent of this operation, the user might be able to add annotations to the panels that make the intent explicit. Since the panels are editable, the user might be able to use direct manipulation techniques to define a *temporal constraint* between the width of the box at two different points in time (or panels). The interface for doing this might be similar to Chimera's interface for defining constraints between separate objects.

There are a number of basic ways in which our macro by example facility can be enhanced. Currently when we save macros, we save all of the scene state, which allows us to restore the original graphical representation of macros for subsequent editing. This increases the storage requirements of the macro. We could also provide an option that automatically strips superfluous scene objects from the macro. Not only would this reduce storage, but it might also make the graphical macro representation clearer since it would contain no extraneous objects.

We would also like to expand the generalizations that Chimera is capable of making, so that a greater number of useful macros by example can be defined. Though graphical search and constraint-based search can be used as a flow of control mechanism for graphical macros, others would be helpful as well. As described in Section 7.4, Chimera employs a simple heuristic to determine loop boundaries. Currently users cannot override the boundaries selected by the system, but there should be a mechanism for doing so.

Finally, it is important to provide a means of representing changes to the macros within the graphical representation, as it is to represent changes to the history. Currently, Chimera can generate histories for macro panels that have been edited, but once a panel's changes have been propagated into the macro, or a panel has been deleted, there is no indication that these changes were made after the initial macro definition.

"I say, David, to the young this is a world for action, and not for moping and droning in. It is especially so for a young boy of your disposition, which requires a great deal of correcting; and to which no greater service can be done than to force it to conform to the ways of the working world, and to bend and break it."

"For stubbornness won't do here," said his sister "What it wants is, to be crushed. And crushed it must be. Shall be, too!"

He gave her a look, half in remonstrance, half in approval, and went on:

"I suppose you know, David, that I am not rich. At any rate, you know it now. You have received some considerable education already. Education is costly; and even if it were not, and I could afford it, I am of the opinion that it would not be at all advantageous to you to be kept at school. What is before you, is a fight with the world; and the sooner you begin it, the better."

— Charles Dickens, *David Copperfield*

Chapter 8

Conclusions

The Chimera editor provides a collection of related techniques to reduce repetition in graphical editing tasks, cut down on the tedium, and make the editor command set more readily extensible. Some of these techniques work together, others have related goals, and all use *examples* in some way to achieve their benefits.

8.1 Summary

This dissertation presents five new example-based techniques to facilitate graphical editing. The first technique, graphical search and replace, adapts a proven technique in text editing to the graphical domain. Graphical search allows all scene objects matching a common graphical description to be identified. Graphical replace changes a set of their graphical properties to new values. Unlike two other techniques for making coherent, repetitive changes, instancing and grouping, graphical search and replace demands no dependence on prior scene structuring to support these changes.

Graphical search and replace allows searches and replaces on graphical properties, such as line width and fill color, as well as shape. The user specifies the search and replace pattern, in part, using example objects. Since the user works with graphical examples, there is no need to learn textual terms for values of the search and replace properties. A suite of parameters adjusts how the search is carried out; for example, the user can adjust rotation and scale invariance, shape tolerance, and polarity. In addition to helping with everyday graphical editing tasks, graphical search and replace serves as a tool for building shapes defined by graphical grammars, making complex scenes out of simple ones using graphical templates, performing searches for editor scene files by content with a graphical grep facility, and adding control flow to graphical macros.

Constraint-based search and replace extends graphical search and replace to find and modify geometric relationships, which form the elements of shape. It can perform sophisticated transformations of these relationships, repeatedly throughout an editor scene. One class of such transformations is scene beautification, neatening up a roughly drawn illustration. Unlike other systems for scene beautification, constraint-based search and replace allows new beautification rules to be defined by the end user, graphically, and without programming. Unlike beautification, this technique also allows entirely new objects to be added to the scene, constrained to existing objects in useful ways.

In constraint-based search, the user incorporates constraints in the search specification to indicate which geometric relationships should be sought in the scene. The tolerance of the search is provided by example: scene objects that satisfy the search pattern constraints at least as well as the search pattern objects will match the specification. Two different classes of constraints can participate in the replacement pattern: the first specifies new relationships that must be established, the second specifies old relationships that must be maintained. Like graphical search and replace patterns, constraint-based patterns are described, in part, by example. Users can define collections of constraint-based search and replace rules to be archived together, or applied in sequence. With dynamic search and replace, the search process can repeat whenever the scene is edited.

Constraint-based search and replace can be viewed as a technique for inferring intended constraints in a static scene. However since constraints often govern how objects *move* with respect to one another, there may not be enough information in a static scene to infer certain intended constraints. Another technique, constraints from multiple snapshots, infers constraints from a *changing* scene. It examines multiple configurations of a scene, and identifies those constraints, belonging to several classes of interesting geometric relationships, that are invariant in all configurations. It then instantiates these constraints on the scene objects, allowing these constraints to regulate future interactions.

Traditionally, users often specify geometric constraints declaratively; however figuring out which constraints must be instantiated can be a very difficult process, particularly when many constraints must be added to the scene. Constraints from multiple snapshots can in some cases simplify the process by determining the constraints automatically. Users of this technique provide examples of how scene objects move, and the system instantiates the constraints directly. The users need not convert their high-level mental model of the objects' motion to a low-level mathematical constraint description, since the technique does this for them. Constraints from multiple snapshots employs an algorithm that watches the transformations used to convert one example to another, to quickly determine or rule out the presence of certain constraints. By instantiating constraints in the scene, this technique relieves the user of the need to repeatedly reestablish geometric relationships by hand.

Another new technique, editable graphical histories, uses a comic strip metaphor to display commands executed in a graphical user interface. The history mechanism automatically generates panels to show the visual effects and names of the commands. To build a better presentation, it coalesces related commands together in the same panel, it shows only objects relevant to the operation performed plus a little scene context, and it chooses rendering styles for the panel objects based on their roles in the presentation.

These histories serve to remind users of the operations they performed in an application, but they also support an undo and redo facility. Panels in the history can be selected and deleted, or the user can point to a panel, and request that the editor scene undo to that point in time. The panels can be converted into miniature editor scenes themselves, and modified. Then the user can propagate modifications into the command history, changing both the main editor scene and its history. These histories support redo of either selected or undone panels. Through the use of redo, the histories automate simple repetition. Furthermore, since deleting panels and adding commands to the middle of the history requires that the history mechanism transparently perform undos and redos, these histories reduce repetition in this secondary way—the users need not undo and redo steps themselves.

Redoing literal command sequences is often of limited use, since they may not perform the necessary function in other contexts. They may be too *specialized* to perform the task in situations other than that in which they were demonstrated. The history representation can also support building graphical macros by example. Users can scroll through the history, and choose a generally useful sequence of operations to be encapsulated in a macro. The system will copy these panels into a macro window. Next, users can specify the arguments to the macro, pointing to them in the history panels. Since sequences from the history tend to be too specialized to work in other contexts, the system will automatically *generalize* them according to a set of built-in heuristics. Users can view these generalizations in a textual checklist, and override them if necessary.

Graphical macros benefit from the graphical history representation in numerous ways. The representation makes it easy to specify which operations to include in the macro, avoiding the need for special commands to start and stop recording. It allows the user to specify arguments to the macro by pointing, and it displays them graphically. It allows the user and others to review the contents of the macro. If there is a bug in the macro, the user can edit the representation. The representation facilitates providing error messages, and is useful for other purposes as well.

These five techniques interrelate in interesting ways, and while some are useful by themselves, they all relate synergistically to others in the group. Chapter 1 describes many of these relationships. All of these techniques have been implemented within the Chimera editor, an editor framework with media editors for graphics, interfaces, and text. These techniques all operate in both the graphics and interface editing modes of Chimera.

8.2 Limitations and Future Work

This section summarizes some of the limitations of this work, and proposes future work to extend the capabilities of Chimera. First we consider the five editor techniques individually, reviewing in some cases limitations that were described earlier. Then we address limitations to how the techniques work cooperatively, and suggests some ways that Chimera's components could be made to work better together. Some of these limitations derive from Chimera's implementation, while others derive from the techniques themselves.

Chimera's interface for graphical search and replace, the MatchTool, has far too many widgets that must be set prior to its use. This is due in part to the large number of options available for search and replace operations, but time would be well spent simplifying the interface, making it easier and faster to use. It would be worthwhile exploring how to specify search and replace operations within the editor window itself, perhaps providing special shortcuts for the most common types of searches. Dan Olsen suggested that search properties might be inferred from multiple examples (eliminating the need for the search column), and Brad Myers proposed that replacement properties and the search and replace parameters might be determined the same way.

By having a single search column for all objects in the search pane, MatchTool limits the kinds of operations that can be specified. It is impossible to use MatchTool, for example, to search at once for a text string and a red object, since object class and color would need to be checked in the search column, but then the red object's class and the text string's color would be significant to the search. This problem could be remedied by having search columns for each object in the search pane, or by multiplexing the single search column to work independently on each of the search pane objects (perhaps determined by object selection). It remains to be determined whether people would really want such a capability, and whether it could be added to the interface without increasing the complexity of specifying other searches.

In Chimera's constraint-based search and replace interface, search tolerances can be specified only by demonstration. This works well when the user has a mental picture of how far off a relation can be and still satisfy the match; however people that know an exact numeric tolerance for their search should be able to type this in directly. There should also be a mechanism for displaying tolerances numerically. It would be easy to modify MatchTool to support this.

More difficult, yet also important, would be building a mechanism to prevent constraint-based replacements from adding new constraints to a scene that are already expressed in another form. It is easy to avoid adding identical constraints to the scene, yet it is significantly more difficult to avoid adding different sets of constraints that mean the same thing. This is also a problem that needs to be better addressed in constraints from multiple snapshots—the problem of redundant constraints.

When replacements add new constraints to the scene that conflict with existing constraints, there should be a way of identifying only those constraints that participate in the conflict, and allowing the user to choose which to remove. Right now, MatchTool only knows when it cannot find a solution to the augmented constraint system. Developing techniques for identifying conflicting constraints would be a valuable research direction, of benefit to the entire constraint community.

One problem with specifying constraints through multiple snapshots is that the system may infer incidental constraints, unintended relationships that were coincidentally present in all of the snapshots. While manipulating the scene objects, if the user notices that incidental constraints interfere with setting up a particular configuration, he or she can always turn off constraints, set up that configuration and take an additional snapshot. As mentioned in Chapter 5, the constraint set inferred by the snapshot mechanism can be reduced to decrease the likelihood of incidental constraints, but this has the potential of resulting in intended constraints not being identified as well.

The snapshot technique finds all relationships from the set presented in Appendix B that are present in a series of snapshots. It does not find the minimal set. Passing redundant constraints to the solver slows the system down needlessly. General algorithms for identifying redundant constraints would help. Currently Chimera looks for certain relationships known to yield redundant constraints, and filters these constraints as a post-process. Additional types of groups could also help filter redundant constraints. Since the snapshot technique starts out with a completely constrained scene, and incrementally removes constraints, as opposed to traditional declarative constraint specification that works in the other direction, it tends to work better on highly constrained scenes than weakly constrained ones. Also, the technique works best when it is easy to manipulate a scene into a few sample configurations.

The implementation of editable graphical histories in Chimera could be improved by adding better support for isolating and removing inconsistencies in the history brought about by editing it. Currently, Chimera copies the history before it is edited. If an inconsistency arises, the user can restore the history to its prior state. Ideally when a user deletes a panel, the editor would highlight dependent panels and notify the user that these need to be deleted as well. Alternatively dependencies could be automatically broken. For example, if the user fetches the color from one object, applies it to a second, and then goes back and deletes the first object in the history prior to any of these other operations, the set color operation should probably continue to use the color of the object deleted from the history. Edits to the history should be represented within the history itself as being distinctively different than edits to the scene. It should be possible to undo history edits that have already been propagated into the history. It is easy to delete panels corresponding to operations newly added into the history, but there is currently no way to explicitly request that deleted panels be added back into the history. There would be a mechanism for this if the act of deleting a history panel were recorded in another panel. In its editable graphical history window, Chimera displays the single linear sequence of operations that yielded the current editor state. Ideally it would display a tree of potential pathways produced as the history is edited, and make it easy to return to previously visited branches.

Invoking macros in Chimera typically involves too many user interaction steps. This should be streamlined. Chimera macros do not allow variable length argument lists, and even if they did, additional iteration constructs would be necessary to exploit them. Lack of diverse flow of control is one of the major limitations to Chimera's macro by example facility. Ideally, Chimera would be able to infer some flow of control constructs—currently it cannot.

When demonstrating macros, people often do not bother to restore small details of the editor state back to its initial configuration. This is especially the case in Chimera, where macros contain commands extracted from editor histories. During macro definition, users should be able to specify which elements of the editor state should not be affected by the macro. For example, the user might not want a macro to change the caret position, object selection, or alignment line state, even though it alters these in the course of generating the desired effect. Alternatively the user might need to write a macro that changes these environmental attributes, so the system should provide a means of selecting which components of the current state will be restored after macro invocation. Chimera's macros would benefit from a more extensive set of heuristics for inferring the user's intent. The dialogue boxes generated by Chimera to display a panel's interpretation should be made simpler to understand, and often more concise.

All of the limitations described earlier in this section apply to individual techniques or their implementation; however, there are also limitations to the way these techniques work together in Chimera. Figure 1.15 shows existing and potential interrelationships between the five techniques, but only those on the left currently exist in Chimera. This is itself a limitation. Also, not all links on the left represent usage dependencies. For example, a link exists between constraint-based search and replace and constraints from multiple snapshots because the two techniques infer constraints in illustrations, not because they are intended to work cooperatively.

Graphical and constraint-based search and replace however do work well together. In fact, all the examples in Chapter 4 include graphical properties as well as constraints in the search specifications, and the single MatchTool interface supports both techniques. The mechanism for mapping between search and replace pane objects that is used during constraint-based search and replace should also be exploited in graphical search and replace. This would provide a means of disambiguating many currently ambiguous replacements, without resorting to multiple context sensitive searches.

Graphical and constraint-based search and replace are both convenient mechanisms for specifying iterations over scene objects in graphical editor macros. However, other important flow of control operations cannot be specified with the MatchTool. Chimera's macros should have an operator to loop *n* times, where *n* could be generalized to be a numeric argument to the macro or a constant. We should be able to specify that a sequence of commands applies only to objects of a particular type (without requiring that the sequence be the body of the loop).

Currently, Chimera requires that constraints specified in the search and replace panels of MatchTool be created using Chimera's declarative constraint specification interface rather than the snapshot technique described in Chapter 5. This is a limitation of the current implementation that could be remedied easily. Allowing constraints from multiple snapshots to work cooperatively with constraint-based search and replace would allow the constraints in the search and replace panes to be inferred by example.

Editable graphical histories represent editor operations verbatim, and in their current form cannot represent the generalization or intent behind each operation. As a result, Chimera's macro facility generates additional dialogue boxes to convey this information using English language fragments. Perhaps the history panels could be annotated with a visual notation conveying the generalizations, but this would require that the end user learn a new visual language to understand them. The graphical histories could also include a special notation for loops. Since iteration is part of a command sequence's generalization, this information is currently conveyed via supplementary displays, but it would be worthwhile experimenting with how loops and conditionals could be represented in the histories themselves.

Editable graphical histories should visually represent graphical and constraint-based search and replace operations, as well as snapshots taken while specifying constraints from multiple examples. Currently none of these appear in the visual history record. Chimera also does not permit snapshot operations to be encapsulated within macros. Both of these are restrictions of the implementation rather than fundamental limitations of the techniques.

8.3 User Studies

Each of the last five chapters proposes ways to extend Chimera's demonstrational techniques and implementation in interesting and potentially useful ways. However at this point, the most valuable continuation of this research would be user studies.

In order to better evaluate the utility of the various techniques, and whether real people (as opposed to the implementer) can truly benefit from them, numerous experiments must be performed. We met with members of Microsoft's Usability Laboratory to discuss how the new editor techniques described in this thesis might be evaluated, and although none of these experiments have been carried out, the tests are still worth describing. This section outlines some experiments that might prove useful for deciding whether to include these features in future graphical editors.

In evaluating a product feature, usability professionals explore three phases of its use: discovery, learning, and performing. During the discovery period, users find out that the feature exists by roving through menus, reading documentation, or seeing mention of it in a tutorial. During the learning period, the user gains proficiency with the feature by understanding the necessary steps and executing sample tasks. The learning period contains the steepest part of the learning curve. The performing period begins after the user becomes proficient with the feature. In practice, these phases are not completely separate. Users discover new components of a feature when using it, and learning often happens together with performing; nevertheless, it helps to consider how well a feature fares in each of these phases.

The techniques described in this thesis have been developed to facilitate the learning and performing phases rather than the discovery period. The synergistic relationships between the components could promote the discovery of new features, but this has not been a focus of the work. For example, users of graphical search and replace would see a number of constraint-related widgets in MatchTool, and might assume MatchTool has certain constraint capabilities. Users of editable graphical histories would see the Macroize command within the history menu, and might assume a history-based macro facility. It would be worthwhile to test how quickly users could glean the presence of these techniques, given a basic familiarity with the rest of the editor, but this has not been a focus of the research. Inadequacies found in Chimera's support for discovery could be fixed through low-level improvements in the menu structure, tutorials, and documentation.

The most valuable experiments would evaluate how Chimera's techniques support the learning and performing phases. Experiments could measure the amount of study and practice necessary for subjects to become proficient at the techniques. Tests following the learning period would ascertain whether proficiency was achieved. Additional experiments could compare the length of time subjects take to learn the five techniques described in this thesis with the time required to learn competing techniques. One of the goals behind exploiting example-based techniques is that skills that users develop during ordinary interaction with an application should partially transfer to these methods. Hopefully this will make example-based techniques easy to learn.

To test the learnability of graphical search and replace, we would initially train a group of subjects to have basic Chimera graphical editing skills. After the training period, we would randomly split the group into four subgroups, and assign a different learning duration to each group (e.g., 15, 30, 45, and 60 minutes). After providing each individual with a learning period of the assigned duration, we would test each individual on a suite of new example tasks to test how well he or she learned the technique. Our hope and expectation is that people would become skilled enough to benefit from graphical search and replace in less than 30 minutes, and our limited experience showing MatchTool’s interface to prospective users supports this. If test subjects take longer, it would be important to study the problems they encountered in the interface, and redesign the interface to reduce learning time. One traditional technique for finding problems in an interface is to videotape subjects attempting an application task, while having them vocalize at every step the goal behind their interactions, the steps they are performing towards this goal, and whether the interface is conforming to their expectations. Reviewing the videotapes often reveals the common difficulties people encounter using the interface. It may be that people have no trouble understanding the technique, but they have trouble working with the interface. Alternatively, the videotapes can indicate difficulties that subjects have mapping a conceptual task to an application feature. It may be that people do not mentally decompose problems in a suitable way to be addressed by a particular technique.

In the case of constraint-based search and replace, we would perform similar learnability experiments. we would test a group that has already shown proficiency at declarative constraint specification in Chimera, and graphical search and replace, since many of the skills are shared. An important question that these experiments would answer is how readily people can learn the difference between set constraints (constraints in the replace pane that establish new geometric relationships in the match) and fixed constraints (constraints in the replace pane that hold a geometric relationship in the match unchanged during the constraint solution). Users of constraints and graphical search and replace should be able to learn the technique within 30 minutes. If many people have trouble with this, we would isolate the problem areas using videotaped sessions and try to address them in an interface revision.

The time required to learn the snapshot technique for specifying constraints should be compared with declarative specification. An experiment to determine the learning time of editable graphical histories should break the process into two components: learning the visual notation of the histories, and learning how to work with the history interface. The visual language of Chimera's history panels is based on the visual presentation of the application itself, so we would expect users of the editor to quickly learn to interpret them. Editing the history stream can be tricky, since users of the current system must learn how to avoid creating dependency conflicts. It would be interesting to implement multiple techniques for resolving such conflicts, and determine which of these techniques users most readily understand and prefer.

A learnability study of Chimera's macro by example system would help determine if there are too many steps in macro definition for people to remember. If so, it would be easy to add a reminder facility that prompts users for the next step. To define macros in Chimera, users may need to understand parameterization. It will be interesting to determine how quickly non-programmers can learn this concept. Also, learnability studies might provide ideas for improving the readability of Chimera's macro generalizations. It would be instructive to compare the amount of time necessary to learn macros in Chimera with that needed to learn textual macros of the same complexity.

Perhaps the most important question to be determined from user testing is whether people can reap benefits from these techniques after taking the time to learn them. One way of testing this is to compare the amount of time to perform various tasks with these new techniques and with conventional means. For example, we would take two groups of individuals, only one of which has been taught graphical search and replace, and compare the amount of time individuals in both groups take to convert the terminals of Figure 1.1 to the workstations of Figure 1.3, the oak leaves of Figure 3.1 to the maple leaves of Figure 3.3, to create the maple tree of Figure 3.1 from scratch (which can be done by first building a template as in Figure 3.21), and to create the fractal snowflakes of Figure 3.7b. Both instancing and graphical search and replace can facilitate certain classes of coherent changes, but making changes with instancing requires that the scene initially be constructed using this technique. Extensive tests would compare graphical search and replace with other methods in scenes that have and have not been built with instancing.

To demonstrate the utility of constraint-based search and replace, we could ask test groups to attempt some of the tasks illustrated in Chapter 4 with and without this technique. For example, we might ask one group to write a rule to make nearly 90° angles precisely 90° and have the MatchTool automate the replacement. The other group would perform the scene changes by hand. We would have both groups perform the changes on scenes similar to Figure 4.2a, yet containing many more polygons. Similarly, we would create a scene similar to Figure 4.7a (but containing many more circles and lines) and ask the two groups to make all line endpoints that are nearly tangent to circles precisely tangent, using constraint-based search and replace and conventional techniques. To test whether experienced users benefit from specifying constraints by multiple snapshots, it would be useful to compare groups using this technique and conventional constraint specification on several examples (for example, the balance of Figure 1.6, the rhombus and line of Figure 5.1, and the Luxo lamp of Figure 5.4).

To test whether editable graphical histories are useful, we could take two groups, only one of which has been trained in Chimera’s history interface, and individually ask members of both groups to recreate a particular graphical scene. After this, the subjects would be asked to change the scenes in a way that could be achieved either by altering the history or editing the current scene, and then we would determine whether people make these modifications with or without Chimera’s history interface. It would also be useful to compare the graphical histories against textual histories for their ability to enhance user recall of editing steps.

Chimera’s graphical macros by example should be compared against a textual macro language or programming extension language to determine which of these techniques allow users to complete their editor extensions fastest. Since Chimera’s macros are currently more limited than programming languages and many textual macro languages, we would compare only tasks that can be accomplished using all three.

To show that the techniques presented in this thesis are worthwhile, it is also necessary to provide evidence that they address problems that really do come up in graphical editing. This type of question tends not to be answered in the usability lab; it requires an analysis of the tasks people tend to perform in graphical editors. Companies often determine the need for particular application features by studying market survey reports and focus group videotapes. The techniques described here were developed because we and others saw a need to reduce user repetition in graphical editing tasks and make graphical editors more extensible. However performing the experiments described in this section will be necessary to determine how well this goal has been met.

8.4 Contributions Reviewed

As described in Chapter 1, this research contributes a suite of techniques to:

- reduce the amount of repetition in graphical editing tasks
- allow experts to encapsulate their knowledge in a form that all users can employ
- permit users to customize or extend the editor command set
- enable users to accomplish this without conventional programming skills

Now, let us consider specifically which of the techniques presented here accomplish each of these goals, and the manner in which they do. Figure 8.1 displays a chart presenting the techniques and the objectives that they meet. Each row represents one of the techniques discussed in the thesis, and the columns represent these four goals. Large checks appear where the techniques accomplish the goals well; small checks appear where a weak argument can be made that they do.

Graphical search and replace reduces repetition by automating repetitive changes to graphical properties and shape. This is a common class of repetition in the graphical

	Repetition Reduction	Knowledge Encapsulation	Extensibility	Demonstrational Approach
Graphical Search and Replace	✔	✓	✓	✔
Constraint-Based Search and Replace	✔	✔	✔	✔
Constraints from Multiple Snapshots	✔	✓		✔
Editable Graphical Histories	✔	✔	✓	✔
Graphical Macros by Example	✔	✔	✔	✔

Figure 8.1 A diagram listing the five techniques, and the goals they accomplish.

editing domain. Though users can archive graphical search and replace rules, there is rarely a need to do this for the purpose of capturing expert knowledge, since graphical search and replace rules are typically easy to define. However, graphical grammar rules can be trickier, and building an archive of these may prove useful. Also, presenting novice users with a prefabricated set of learning rules might help them to become proficient faster. Graphical search and replace extends the editor command set in a very constrained way, allowing users to accomplish new tasks representable by the search and replace language. Since graphical search and replace rules contain example objects as part of the specification, and they represent larger classes of matching objects and replacements, we classify this technique as demonstrational.

Constraint-based search and replace finds objects obeying specified geometric relationships, and modifies them to obey another set of relationships. Users can apply these replacement rules to entire scenes, thus eliminating the necessity to set up the relationships by hand multiple times. Some useful constraint-based rules are very simple, but others are much more complex, so Chimera includes a facility that allows knowledgeable users to save and share their specifications. Constraint-based search and replace allows users to extend the operations the editor can perform in meaningful ways, and raise the abstraction level of the command set to better suit their tasks. The rounded corner rule of Chapter 4 serves a good example; instead of requiring users to splice arcs into line intersections, and set up multiple constraints to ensure tangency every time they want to make a corner round, users can abstract these steps into a rule. Constraint-based search and replace also uses examples within the search and replace specification, so it is demonstrational. Furthermore, users indicate the search tolerance by example, and there is a demonstrational heuristic for distinguishing between set and fixed constraints.

Graphical editor users often find themselves experiencing another class of repetition, setting up the same geometric relationships over and over again whenever moving scene objects. Constraints from multiple snapshots help here by adding constraints to the scene that automatically maintain these relationships. Graphical editing experts have little advantage over novices with this technique. However the technique can generate constrained clip art which can be archived, so in Figure 8.1 it receives a small check in the encapsulation category. The technique does not enable editor users to extend the command set, so it gets no check for this. Since each snapshot that the user provides is an example, and the examples together represent a larger class of possible scene configurations, this technique also qualifies as demonstrational.

By providing an interface and visual support for redo operations, editable graphical histories help reduce repetition. They also provide knowledge encapsulation, by allowing experts to construct visual records showing the steps they followed to accomplish a difficult task, and to share these visual representations with less knowledgeable users. Users can extend the editor command set in a superficial way by chunking sequences of past commands. Editable graphical histories enable users to perform commands on an example, and replay them elsewhere in the scene, so this also a demonstrational technique.

Graphical macros by example help reduce repetition further, since they permit sets of past commands to be generalized to apply to different contexts. Expert users can write macros for novices, so this technique supports knowledge encapsulation. Furthermore, users can write macros to extend the interface to include sets of operations that they wish were primitives. This technique is also demonstrational because users define macros, in part, by demonstrating their execution steps through the interface.

A final contribution of this work is Chimera itself, an editor system featuring all five of these new techniques, allowing them to be used together for editing graphics and interfaces.

Implementing Chimera required considerable effort, and we would like to see it used by large numbers of people. Having others work with the editor on a daily basis would be one of the best ways to truly evaluate the utility of its components. However, currently several factors impede Chimera’s widespread use: an interface biased towards experts, a dearth of documentation, dependence on a run-time Common Lisp environment and the OpenWindows 3.0 window system, and lack of built-in error-recovery.

If others cannot benefit directly from using Chimera, they can still profit from the ideas it introduces. Graphical search and replace, constraint-based search and replace, constraints from multiple snapshots, editable graphical histories and a history-based macro by example system present new ways of reducing the repetition in graphical editing tasks, making editors more extensible, and encapsulating expert knowledge, all without conventional programming.

Two programming by demonstration research projects at other universities have already applied ideas from Chimera, and have begun extending this work in new directions. Henry Lieberman's Mondrian project at the M. I. T. Media Laboratory has adopted a visual representation for graphical editor programs that was partially inspired by Chimera's editable graphical histories (see Section 2.2.8). Lieberman would like to add support for conditionals and recursion in Mondrian, and extend the panel representation to include diverse flow of control. Francesmary Modugno at Carnegie Mellon University is building a programming by demonstration system for a graphical shell [Modugno93]. The visual representation for these programs resembles editable graphical histories, but it is more abstract and it displays generalizations within the panels themselves. Modugno also plans to incorporate flow of control within the visual representation. Hopefully others will build on these techniques as well, solving new problems within graphical editing, and finding applications for these techniques in other domains.

Appendix A

The Naming of Chimera

In Greek mythology, Chimera (or Chimaera) was a monster consisting of three parts: the head of a lion, the body of a goat, and a serpent's tail. The hero Bellerophon was sent on a mission to kill Chimera, but to make the task more tractable he first sought several tools to help him in his quest, most prominently Pegasus. In modern usage, a chimera is an illusive, often unachievable goal. Keeping these facts in mind, below are the top 10 reasons for naming my editor "Chimera." Can we have a drumroll, please.

Top 10 Reasons for Calling My Editor "Chimera"

10. The Chimera editor and monster both consist of multiple pieces that work together to form a coherent whole. See Section 1.5 describing synergy in the Chimera editor.

9. The Chimera editor implementation contains code written in three languages: Lisp, C, and PostScript.

8. Bellerophon sought several tools to slay the monstrous Chimera. Similarly, the Chimera editor system introduces several new tools to help slay monstrous graphical editing tasks.

7. Though Chimera can be extended to edit other types of media, my original editor implementation supports the creation and modification of three classes of objects: graphics, interfaces, and text.

6. Example-based specification has been considered by many to be a chimerical pursuit.

5. Work on Chimera was completed in 1992, the demimillenium of Columbus' arrival in America. Take "America", rearrange the letters, and you get "Chimaera". Why the extra "H"? For all the late night Hacking.

4. Chimera is the first editor to support all of the following three methods of exact graphical measurement and placement: constraints, snap-dragging, and grids.

3. CHIMERA is an acronym: Computer-Human Interface Making Editing Really Awesome!

2. There is a grand tradition of naming graphical editors after fantastic creatures, for example, Griffin, Gargoyle, Gremlin, Unicorn, and MacDraw.

And finally, the number one reason for naming my editor “Chimera”:

1. I just like the name.

Appendix B

Chimera’s Constraint Set

This appendix describes the collection of constraints supported by the Chimera editor. Chimera’s declarative interface for instantiating these constraints and its interface for viewing these constraints will be described in Appendix C.

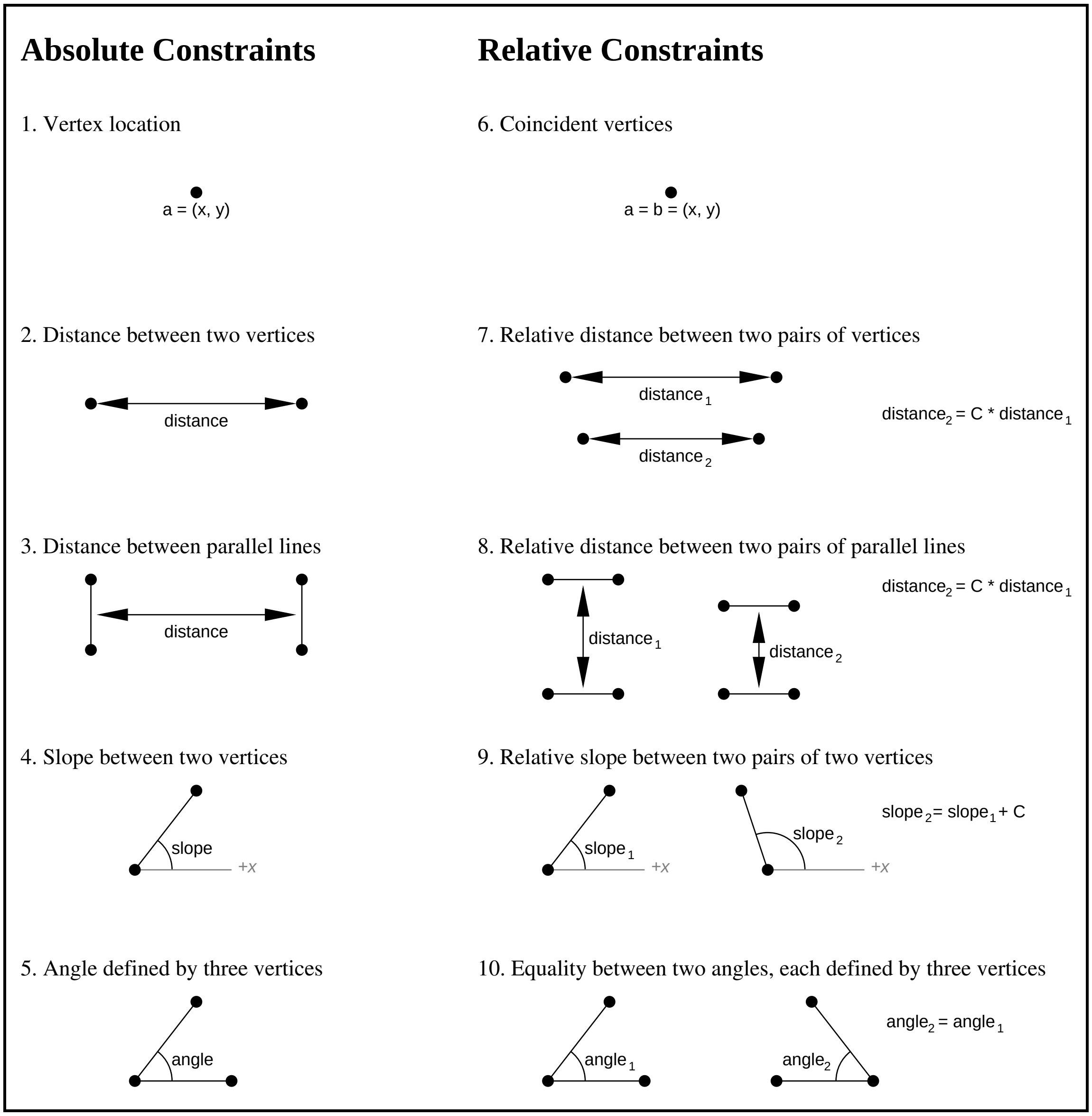


Figure B.1 Constraints in Chimera.

Figure B.1 lists the constraints currently available in Chimera. All of these constraints act on vertices (or “handles”) of graphics and interface objects, and they can be created by the multiple snapshots technique described in Chapter 5 or by declarative specification as described in Appendix C. Chimera has both absolute and relative constraints. Absolute constraints restrict geometric relationships to a specific value, for example, one might set the distance between two vertices to be a quarter inch, or the angle defined by three vertices to be 90°. Relative constraints create an association between multiple geometric relationships, such as setting two angles to be equal. In Figure B.1, each absolute constraint on the left corresponds to a relative constraint on the right.

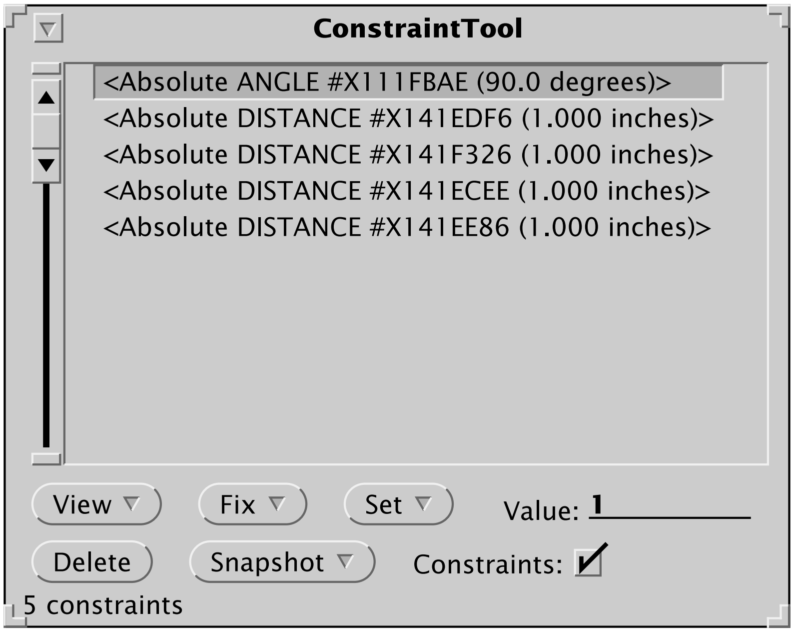
The relative slope constraint fixes one slope to be a constant offset from another (when represented in terms of degrees, not y/x ratio). Each of the relative distance constraints fixes two distances to be proportional to one another. Two of the above constraints subsume two others: the absolute distance constraint between vertices subsumes the coincident vertices constraint, and the relative slope constraint subsumes the absolute angle constraint. Chimera’s constraint solver does not explicitly support the subsumed constraints, since it handles the more general relationships. Similarly, the inference component has no support for coincident vertex constraints, though it does track absolute angle relationships since the algorithm uses these to find equal angle relationships. However, Chimera’s declarative constraint interface differentiates between all of the constraints in Figure B.1, since specifying the more general relationships requires additional input parameters.

Parallel and orthogonal vector relationships are largely captured by the relative slope constraint (which, for example, in the former case would not only fix the vectors between two pairs of two vertices as parallel, but would also constrain their relative directions). Similarly, the relative slope relation captures collinearity, with an additional ordering on the vertices.

Appendix C

Chimera’s ConstraintTool

Chimera’s utility for creating, deleting, and displaying constraints is called the ConstraintTool. An instance of the ConstraintTool is shown in Figure C.1.



<Absolute ANGLE #X111FBAE (90.0 degrees)>
<Absolute DISTANCE #X141EDF6 (1.000 inches)>
<Absolute DISTANCE #X141F326 (1.000 inches)>
<Absolute DISTANCE #X141ECEE (1.000 inches)>
<Absolute DISTANCE #X141EE86 (1.000 inches)>

Value: 1
Constraints: ☑
5 constraints

Figure C.1 The ConstraintTool.

C.1 Creating Constraints

Chimera has two mechanisms for adding constraints to a scene: the snapshot technique described in Chapter 5, and a more traditional declarative interface. To support instantiating constraints by multiple snapshots, the ConstraintTool includes a button to trigger snapshots, and a checkbox that allows constraints to be turned off while creating additional example configurations. To add a new constraint to the scene with the declarative interface, the user selects the vertices to constrain, and chooses a constraint class from either the FIX menu or the SET menu. Each of these menus list all of the constraint types known to Chimera. The FIX menu constrains the chosen geometric relationship to its current value, while the SET menu sets this relationship to a new value. When executing a command from the SET menu, the user must also specify a new value for the geometric relationship in the Value field.

For example, suppose we have a rectangle that we would like to convert to a square, an inch on a side. Since the shape is already a rectangle (the shape was drawn with a grid), the angles are already 90°. We choose to fix one of the angles at 90° by selecting the lower right vertex, the lower left vertex, and the upper left vertex, and choosing the Fix-Absolute-Angle command from the FIX menu. Next we make all of the lines one inch long by typing “1” in the Value field, selecting each segment in turn, and invoking the Set-Absolute-Length command on it from the SET menu. It is necessary to type a value and use the SET menu to create these distance constraints, because initially none of the rectangle’s sides had a length of one inch.

C.2 Viewing Constraints

To view these constraints, we must first add them to the scrolling list in the top portion of the ConstraintTool. This initial step is easily accomplished by choosing the Browse-Constraints command from the View menu, or simply by clicking the left mouse button on the View menu button to invoke this default action. A textual representation of each constraint appears in the scrolling list. The visual display of constraints in Chimera is this textual description along with a graphical representation. Each constraint in the scrolling list can be selected, and this activates its graphical representation in the editor scene. The Juno editor has graphical and textual representations of constraints as well, with the textual representations being procedures [Nelson85].

Since graphical constraint presentations quickly become cluttered as the displayed constraint set grows, it is important to allow constraints to be displayed selectively. The last section described how the ConstraintTool interface allowed us to convert a rectangle to a square, an inch on a side. The original square appears in Figure C.2a. All five constraints that were added to this rectangle appear in the ConstraintTool shown in Figure

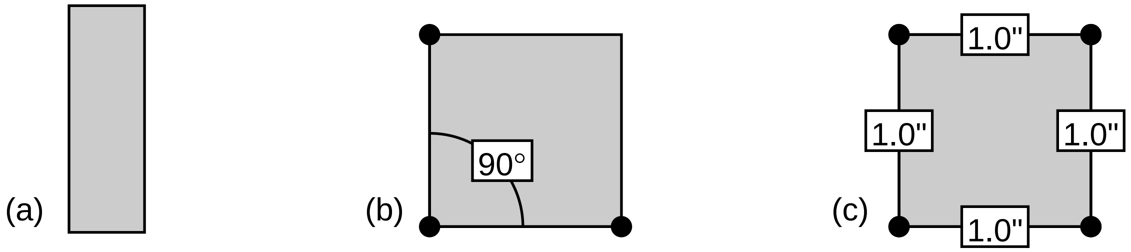


Figure C.2 Making a rectangle into a square. The initial rectangle appears in (a). After adding all the necessary constraints, we display the 90° angle constraint in (b), and the four 1 inch distance constraints in (c).

C.1. Here we have selected only the angle constraint, making it the single constraint visible in Figure C.2b. When we deselect the angle constraint, and select the four distance constraints, the editor scene appears as in Figure C.2c.

Textual and graphical constraint representations have complementary features. Chimera's textual view contains a descriptive constraint class name, as well as a unique identifier (which is particularly useful for debugging). It also contains numerical constraint constants displayed to several significant digits. Chimera's visual language for displaying constraints graphically appears in Figure C.3. The constraints and vertex positions in this figure correspond precisely to those in Figure B.1, which lists the constraint types available in Chimera. Chimera's graphical constraint notation indicates which vertices are affected by each constraint by highlighting them with small disks, and adds graphical annotations to link multiple dependent vertices or provide graphical syntactic sugar (such as the arcs for the angle constraints). Each graphical display employs a symbol or rounded number to indicate the constraint type and in some cases its value. Some constraints include only symbols in their graphical display, showing numeric parameters exclusively in their textual presentation. The correspondence between each constraint's textual and graphical presentations becomes clear to users as they toggle the textual entry and see the graphical display turned on or off.

Chapter 4 describes constraint-based search and replace. Users specify these patterns, in part, by drawing or copying shapes into two graphical editor scenes (the search and replace panes). These panes are actually instances of the Chimera graphical editor, and constraints can be created and viewed in them with the help of the ConstraintTool. As explained in Chapter 4, the replace pattern can contain two categories of constraints: fixed constraints and set constraints. For set constraints, Chimera uses the visual representations in Figure C.3. For fixed constraints, it uses these same graphical notations, with the symbols in the boxes followed by asterisks. It is not appropriate to include a constant in the visual presentation of fixed constraints (since these constraints refer to the value of the match, not the pattern itself), so when drawing a fixed constraint diagram that would normally include a constant, Chimera replaces this constant with a letter symbol determined by the constraint type.

C.3 Constraint Filtering

When an editor scene contains numerous constraints, looking for a particular one by toggling the graphical presentation of each entry in the scrolling list can be slow and tedious. The ConstraintTool includes commands to turn on and off the graphical presentation of all constraints in the scrolling list, but the simultaneous display of large numbers of constraints is usually confusing. To make it easier to find a certain constraint, the ConstraintTool features operations to filter or retrieve constraints according to specified properties. The user can select a set of scene objects, and execute the Browse-Constraints-Of-Selected command from the View menu. This replaces the contents of the ConstraintTool's scrolling list with only those constraints that affect these objects. This typically produces a much smaller list that is quick to search. The contents of the scrolling list represents the results of a query or filtering operation on the constraints in the scene. The Browse-Constraints command simply requests that all of the constraints in a scene be retrieved and placed in the scrolling list. At some later time we would like to add filtering operations that take the contents of the scrolling list as input, so that the user can cascade multiple queries.

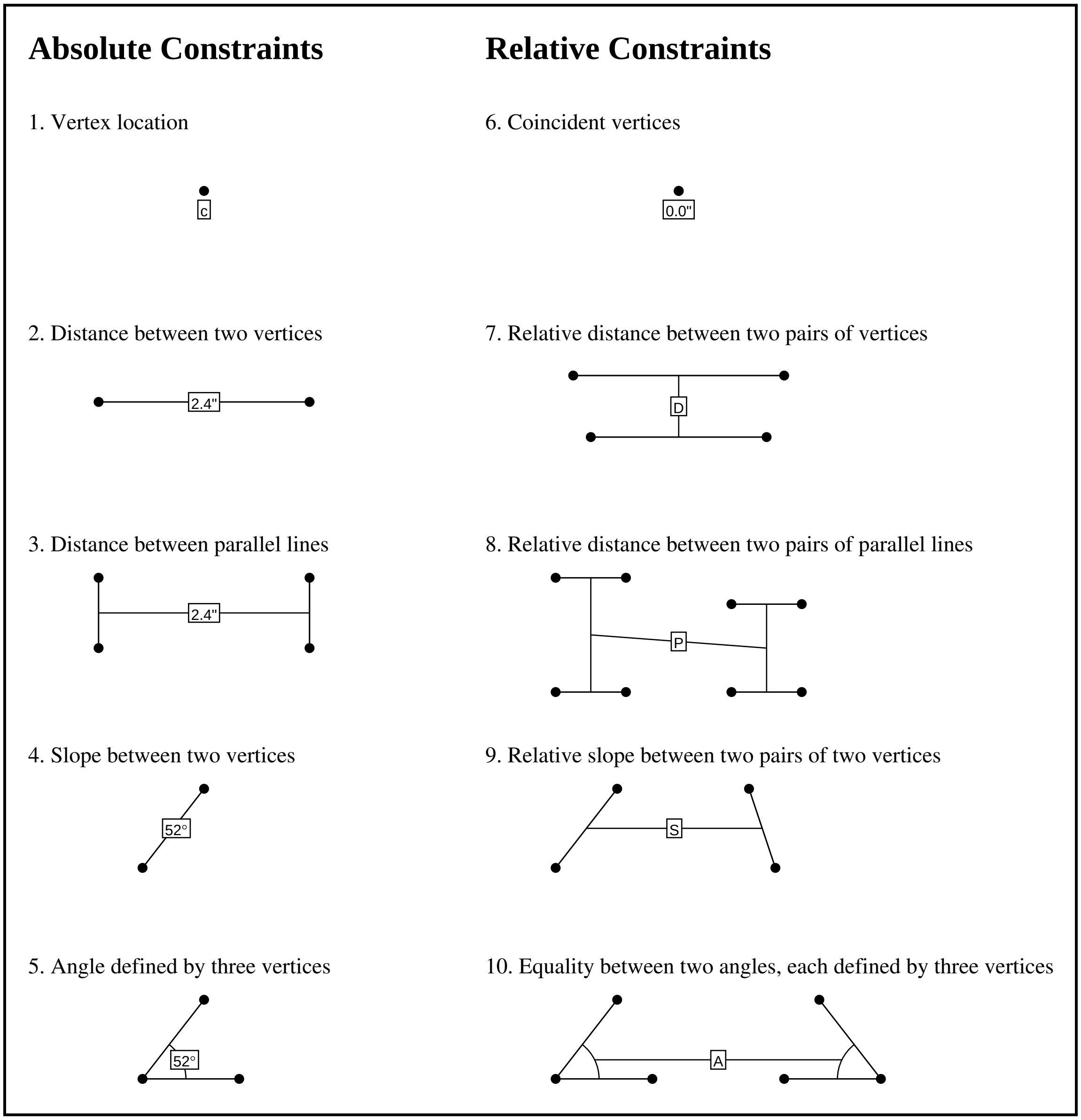


Figure C.3 Chimera's graphical notation for constraints. Each constraint and vertex in this figure corresponds to one in Figure B.1. (Because this figure has been reduced, the distance constraints do not accurately reflect distance on paper.)

If the user selects one or more constraints, and executes the Draw-Like-Constraints command, the system automatically highlights constraints of identical types in the ConstraintTool, and displays these in the scene. The Delete button of the ConstraintTool eliminates selected constraints from the scene, and erases their textual and graphical representations.

Appendix D

Macro Generalization Heuristics

Chimera has about 180 commands in its graphics and interface editing modes combined, with most of the commands being common to both of these modes. To explicitly code up separate generalization heuristics for each command would be a laborious task. Instead, we have classified Chimera's commands according to argument types, and coded generalization heuristics based mainly on the arguments and not the commands. Since there are only a few argument classifications (discussed below) this was significantly faster than writing separate generalizations for each of Chimera's commands, but most importantly, this works well because the generalization of nearly every Chimera command is determined by the generalizations of its arguments. For example, consider the delete and copy operations. Both act on the current object selection. The user might delete or copy an object for the same reasons (e.g., it was an argument to macro, a constant object, or an object generated by a particular macro step). This approach is similar to SmallStar's data descriptions, but here the generalizations of the arguments are computed by inferencing and restricted to plausible interpretations. In Chimera, theoretically a command can alter the standard generalizations of its arguments, but there are no cases yet where this has been needed.

For the purpose of generalization, arguments to Chimera commands are categorized into three broad types: object selections, properties, and caret states. Commands that take the current object selection as an argument typically accept either a single object or multiple objects. Components of single or multiple objects can also be selected. Properties in Chimera are attributes that have a single or compound value associated with them. Scene objects can have many properties, such as fill color and line style. The scene itself can have other properties, such as gravity strength or output file. The slope, distance, or angle measure of an alignment line is also a property. The caret is Chimera's cursor, and the caret state represents position information in Chimera. Note, this is a somewhat arbitrary classification, and the property argument type is actually an umbrella group for some very different attributes. However, for the purpose of generalizing arguments in Chimera, this classification scheme works well.

D.1 Object Selection Heuristics

The generalization for an object selection must be general enough to account for every object in the selection, yet specific enough to refer only to objects in the selection. Chimera constructs object selection generalizations from the following list:

1. the argument named [name], (where [name] is a macro argument)
2. the result of a search loop
3. an object created in panel #[panel number]
4. the whole object ancestor of X
5. the [ordinal] control point of X, (where X is a polycurve segment)
6. the [first | second] joint of X, (where X is a polycurve segment)
7. the [ordinal] segment of X, (where X is a polycurve)
8. the [left | top | right | bottom] segment of X, (where X is a box)
9. the [left | top | right | bottom | baseline] segment of X, (where X is text)
10. the [upper left | upper right | lower left | lower right | center] joint of X, (where X is a box)
11. the [left baseline | upper left | upper middle | upper right | middle baseline | right baseline | lower left | lower middle | lower right] segment of X, (where X is text)
12. the [left | top | right | bottom | center] joint of X, (where X is a circle or ellipse)
13. the parent of X
14. the combination of X and X
15. a constant object

In this list, values contained in square brackets are automatically determined by the generalization code. When multiple items appear in brackets separated by vertical bars, the system chooses one of these items, as appropriate. Each instance of an X in these lists must be replaced by another line from this list—this is how composition is achieved. For example, possible generalizations of an object selection might be "the left segment of the argument named Rectangle" and "the upper right joint of the result of a search loop" and "the combination of the argument named Text1 and the argument named Text2." The combination operator actually represents disjunction not conjunction (in the last example, a scene object will have been selected if it is either the argument named Text1 or the argument named Text2).

Chimera presents explanations for object selections, ordered according to an estimation of their likelihood. Simple explanations tend to be more likely than complex explanations. Accordingly, the first three explanations on this list are all considered likely when they can account for the selection of an object, since they do not have X substitutions (they are not explanations formed by the composition of multiple explanations). Each of these explanations are given a likelihood ranking of 1 (lower rankings are most likely). The explanations that have X's require at least one composition. The likelihood ranking of an explanation with an X substitution is one plus the likelihood ranking of its X substitution(s). Currently we limit the search to explanations that have a likelihood of three or lower. In a future version, we might allow the user to request a deeper search if the correct explanation is not presented, but this does not appear important for the fairly simple macros that we have generated. The explanation that an object is a constant in the macro, tends to be unlikely, even though it requires no composition. It is always ranked as the least likely explanation of those found. Currently we do not allow the constant object explanation to participate in compound explanations. In many cases this would not make sense (e.g., "the left segment of a constant object" is a constant object itself), and though the constant object explanation is rare, compositions of this explanation would be rarer still. The resulting explanations are sorted and presented to the user, with the most likely explanation chosen as the default. The English language presentations of these explanations are exactly as generated by the above rules, except multiple nestings of the combination operator are flattened before displaying them to them to the user (e.g., "the combination of A and the combination of B and C" is presented as "the combination of A, B, and C.").

D.2 Property Heuristics

Chimera can infer three explanations for the choice of a particular property value. These appear below in order of decreasing likelihood.

1. The value given by argument [name], (where [name] is a macro argument)
2. The value previously in the widget
3. A constant value

As described in Section 7.3.2, Chimera macros can have property arguments as well as object arguments. If a property value used in the macro is identical to that of a previously declared argument, Chimera assumes that the value was chosen for this reason. Most Chimera properties are set with a single widget, the Text Input field. If the property value used in the current panel was the value left previously in the Text Input field, Chimera considers this a likely explanation for its use. Chimera has about a dozen commands that query properties in the scene, and put their value in the Text Input widget. For example, the user can query the fill color or text transformation of a scene object, then assign this value to another object. This second generalization for property values allows this to be done in macros as well.

These generalizations, like all generalizations in Chimera, can be chosen as an intent hypothesis only when they can satisfactorily explain a particular command. For example, Chimera will not propose that a value used in a given panel is the value of a macro argument, when no macro argument of that value has been declared. Chimera will not suggest that a value was used because a control panel widget was already set to this value, if it was not set this way. In all cases a value may have been chosen for use as a constant.

D.3 Caret State Heuristics

Most drawing commands manipulate the caret, which visually represents Chimera's current location. The caret is gravity sensitive, and objects in the scene, as well as alignment lines attract the caret. Translations, rotations, and scales are guided by the caret to specify these transformations precisely. This technique, called snap-dragging, was initially used in the Gargoyle editor, and is described in [Bier86]. Most drawing commands in Chimera use the caret. For example, when adding a line to the scene, the line rubberbands between the position of the caret at the start of the operation, and the current position of the caret, as it follows the hardware cursor. Boxes are added to the scene similarly, with one corner of the box at the initial caret position, and its diagonally opposite corner at the current caret position. During translations, selected objects are moved by the offset between the current caret position and its position at the beginning of the translation. These are Chimera's interpretations for a change in the caret position, in order of decreasing likelihood.

1. move caret to [vertex | anchor]
2. move caret to the intersection of two objects
3. move caret relatively by [offset]
4. move caret absolutely to [location]

In the first case, the caret moves until it snaps to a control vertex in the scene, or the anchor, which represents the center of the coordinate system for transformations. Control vertices and the anchor are both point sources that attract the caret; intersections of objects are another. In Chimera (as in Gargoyle), the caret can snap to intersections of objects (where objects include scene objects and alignment lines). A valid generalization of a caret motion is that the caret moved to an intersection point. However, this is an incomplete generalization, because it includes no generalization for the two objects participating in the intersection. When Chimera verifies that this generalization is plausible for a caret movement, it then needs to build generalizations for the intersecting objects as well. This process is basically the object selection generalization process described in Section D.1, yet alignment lines can participate in intersections as well, though they are not included in the standard object selection heuristics. The generalization of alignment lines is described next.

Each class of alignment lines is defined by both a source and a measure. To generalize an alignment line, it is necessary to generalize both components. The source is a scene object, either a segment, a vertex, or the anchor. The measure is either an angle, slope, or distance. To generalize the source, Chimera simply uses the object selection generalizations described earlier. To generalize the measure, Chimera uses the property generalization heuristics. For example, consider the dialogue box shown in Figure 7.10. The first checklist indicates that the caret initially was moved to the intersection of two objects. The second and third checklist generalize the source and measure of one alignment line participating in the intersection, and the fourth and fifth generalize the source and measure of the other. Similarly, the sixth checklist indicates that as the rounded box was being drawn, the caret was moved to the intersection of two other alignment lines. The seventh and eighth checklists, and the ninth and tenth checklists generalize the source and measure of each of these alignment lines.

Chimera can also interpret caret motions as either relative or absolute moves. Relative moves are far more common than absolute moves, so they are ranked as a more likely interpretation.

Bibliography

[Adobe85] Adobe Systems Inc. *PostScript® Language Tutorial and Cookbook*. Addison-Wesley, Reading, MA. 1985.

[Adobe90] Adobe Systems Inc. *Adobe Illustrator User Manual*. Macintosh version 3. Part no. 0199-2045 rev. 1. 1585 Charleston Road, Mountain View, CA 94039. November 1990.

[Aldus90] Aldus Corporation. *Aldus PageMaker Reference Manual*. version 4.0. ISBN 1-56026-021-1. 411 First Avenue South, Seattle, WA 98104. April 1990.

[Arnon88] Arnon, Dennis, Beach, Richard, McIsaac, Kevin, and Waldspurger, Carl. CaminoReal: An Interactive Mathematical Notebook. In J. C. van Vliet, editor, *Document Manipulation and Typography: Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography (EP88)*, Nice, France. April 20-22, 1988, Cambridge, Cambridge University Press, INRIA. 1-18. Also available in Xerox PARC Technical Report EDL-89-1.

[Asente87] Asente, Paul. Editing Graphical Objects Using Procedural Representations. DEC WRL Research Report 87/6. November 1987. Revised version of Stanford Computer Science Ph. D. thesis.

[Beach85] Beach, Richard J. Setting Tables and Illustrations with Style. Xerox PARC Technical Report CSL-85-3. May 1985. Reprint of University of Waterloo Computer Science Ph. D. thesis.

[Bier86] Bier, Eric A., and Stone, Maureen C. Snap-Dragging. Proceedings of SIGGRAPH ‘86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics 20,* 4 (August 1986). 233-240.

[Bier88] Bier, Eric A. Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions. Ph.D. thesis. U.C. Berkeley. EECS Department. April 1988.

[Bier89] Bier, Eric A., and Kurlander, David. Interactive Graphical Search and Substitute. *SIGGRAPH Video Review.* Issue 48. 1989.

[Bloomenthal85] Bloomenthal, Jules. Modeling the Mighty Maple. Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22-26, 1985). In *Computer Graphics 19,* 3 (July 1985). 305-311.

[Bonadio88] Allan Bonadio Associates. Expressionist Version 2.0 User's Manual. 814 Castro Street, San Francisco, CA 94114. 1988.

[Borenstein88] Borenstein, Nathaniel S., and Gosling, James. Emacs: A Retrospective. *ACM SIGGRAPH Symposium on User Interface Software*. ACM Press. October 1988.

[Borning79] Borning, Alan. ThingLab: A Constraint-Oriented Simulation Laboratory. Xerox PARC Technical Report SSL-79-3. Revised version of Stanford Computer Science Ph. D. thesis. July 1979.

[Borning86] Borning, Alan. Graphically Defining New Building Blocks in ThingLab. *Human Computer Interaction 2,* 4. 1986. 269-295. Reprinted in *Visual Programming Environments: Paradigms and Systems*. Ephraim Glinert, ed. IEEE Computer Society Press, Los Alamitos, CA. 1990. 450-469.

[Brooks88] Brooks, Kenneth P. A Two-View Document Editor with User-Definable Document Structure. DEC SRC Research Report #33. November 1988. Revised version of Stanford Ph. D. thesis.

[Burr79] Burr, D. J. A Technique for Comparing Curves. *IEEE Conference on Pattern Recognition and Image Processing*. (Chicago, IL, August 6-8). IEEE. 1979. 271-277.

[Chamberlin88] Chamberlin, Donald D., Hasselmeier, Helmut F., Luniewski, Allen W., Paris, Dieter P., Wade, Bradford W., and Zolliker, Mitch L. Quill: An Extensible System for Editing Documents of Mixed Type. *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, Bruce Shriver, ed., 317-326.

[Chang89] Chang, Shi Kuo. *Principles of Pictorial Information Systems Design.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Chen88] Chen, Pehong. A Mutliple Representation Paradigm for Document Development. Ph.D. thesis, U.C. Berkeley. EECS Department. July 1988.

[Christodoulakis88] Christodoulakis, S., and Graham, S. Browsing within Time-Driven Multimedia Documents. *Proceedings of COIS '88*, Conference on Office Information Systems. Palo Alto, CA. March 23-25, 1988. 219-227.

[Chyz85] Chyz, George W. Constraint Management for Constructive Geometry. Master's thesis. MIT. Mechanical Engineering. June 1985.

[Claris88] Claris Corporation. *MacDraw II Reference*. 440 Clyde Ave., Mountain View, CA 94043. 1988.

[Cohen82] Cohen, Paul R., and Feigenbaum, Edward A. *The Handbook of Artificial Intelligence*. vol. 3. Kaufmann, Inc., Los Altos, CA. 1982.

[Cohen92] Cohen, Philip R. The Role of Natural Language in a Multimodal Interface. In *Proceedings of UIST '92* (Monterey, CA, November 15-18). ACM, New York, 1992. 143-149.

[Crowley87] Crowley, Terrence, Forsdick, Harry, Landau, Matt and Travers, Virginia. The Diamond Multimedia Editor. In *USENIX Conference Proceedings, Summer 1987*. June 1987. 1-18.

[Cypher91] Cypher, Allen. EAGER: Programming Repetitive Tasks by Example. CHI '91 Conference Proceedings (New Orleans, LA, April 27- May 2, 1991). 33-39. Revised version appears in *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 205-217.

[Digital80] Digital Equipment Corporation. *PDP-11 TECO User's Guide*. DEC Software Distribution Center, Maynard, MA 01754. Part no. DEC-11-UTECA-B-D. February 1980.

[Digital86a] Digital Equipment Corporation. *Guide to Text Processing on VAX/VMS*. DEC, P. O. Box CS2008, Nashua, NH 03061. Part no. AI-Y502B-TE. April 1986.

[Digital86b] Digital Equipment Corporation. *Text Processing Utility Reference Manual*. DEC, P. O. Box CS2008, Nashua, NH 03061. Part no. AA-EC64C-TE. April 1986.

[Ellman89] Ellman, Thomas. Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Surveys 21,* 2. June 1989. 163-221.

[Feiner82] Feiner, S., Nagy, S., and van Dam, A. An Experimental System for Creating and Presenting Interactive Graphical Documents. *ACM Transactions on Graphics, 1,* 1. January 1982. 59-77.

[Feiner85] Feiner, Steven. APEX: An Experiment in the Automated Creation of Pictorial Explanations. *IEEE Computer Graphics and Applications, 5,* 11. November 1985. 29-37.

[Foley90] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. *Computer Graphics: Principles and Practice.* Second Edition. Addison-Wesley, Reading, MA. 1990.

[Frame90] Frame Technology Corporation. *FrameMaker Reference*. Part no. 41-00524-00. 1010 Rincon Circle, San Jose, CA 95131. September 1990.

[Furnas86] Furnas, George. Generalized Fisheye Views. *CHI ‘86 Conference Proceedings* (Boston, MA, April 13-17). ACM, New York, 1986. 16-23.

[Garey79] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA. 1979.

[Gips75] Gips, James. *Shape Grammars and Their Uses: Artificial Perception, Shape Generation, and Computer Aesthetics*. Birkhauser, Verlag, Basel, Switzerland. 1975.

[Goines82] Goines, D. L. *A Constructed Roman Alphabet.* David R. Godine, publisher. 306 Dartmouth St., Boston, MA 02116. 1982.

[Green85] Green, Mark. The University of Albert User Interface Management System. Proceedings of SIGGRAPH ‘85 (San Francisco, CA, July 22-26, 1985). In *Computer Graphics 19*, 3 (July 1985). 205-213.

[Halbert84] Halbert, Daniel. C. Programming by Example. Xerox Office Systems Division Technical Report OSD-T8402. December 1984. Reprint of Berkeley Computer Science Ph. D. thesis.

[Halbert93] Halbert, Daniel C. SmallStar: Programming by Example in the Desktop Metaphor. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 103-123.

[Hudson90a] Hudson, Scott E., and Mohamed, Shamim P. Interactive Specification of Flexible User Interface Displays. *ACM Transactions on Information Systems 8,* 3 (July 1990). 269-288.

[Hudson90b] Hudson, Scott E. An Enhanced Spreadsheet Model for User Interface Specification. Report TR 90-33. Univ. of Arizona. Computer Science. October 1990.

[Hudson91] Hudson, Scott E., and Yeatts, Andrey K. Smoothly Integrating Rule-Based Techniques into a Direct Manipulation Interface Builder. In *Proceedings of UIST ‘91* (Hilton Head, SC, November 11-13). ACM, New York, 1991. 145-153.

[Jernigan88] Jernigan, Ginger. QuickDraw’s Internal Picture Definition. Macintosh Technical Note #21. Apple Computer. March 1988.

[Johnson88] Johnson, Jeff, and Beach, Richard J. Styles in Document Editing Systems. In *Computer 21*, 1 (January 1988). 32-43.

[Joy79] Joy, William. An Introduction to the C Shell. In *UNIX Programmer’s Manual*, Seventh Edition, Third Berkeley UNIX Distribution. Dept. of EE & CS, University of California, Berkeley, 1979.

[Joy80a] Joy, William. Ex Reference Manual, version 3.5/2.13, September 1980. Revised by Mark Horton. In *UNIX User’s Manual Supplementary Documents*, published by USENIX Association. December 1984

[Joy80b] Joy, William. An Introduction to Display Editing with Vi. September 1980. Revised by Mark Horton. In *UNIX User's Manual Supplementary Documents*, published by USENIX Association. December 1984.

[Karsenty92] Karsenty, Solange, Landay, James A., and Weikart, Chris. Inferring Graphical Constraints with Rockit. In *HCI '92 Conference on People and Computers VII* (September 1992). British Computer Society. Cambridge University Press, Cambridge, England. 137-153.

[Kernighan78] Kernighan, Brian W. A Tutorial Introduction to the UNIX Text Editor. September 1978. Available in *UNIX User's Manual Supplementary Documents*, published by USENIX Association. December 1984.

[Kosbie90] Kosbie, David S., Vander Zanden, Brad, Myers, Brad A., and Giuse, Dario. Automatic Graphical Output Management. In *The Garnet Compendium: Collected Papers, 1989-1990.* Brad A. Myers, ed. Technical Report CMU-CS-90-154. CMU. School of Computer Science. August 1990.

[Kurlander88a] Kurlander, David, and Bier, Eric. A. Graphical Search and Replace. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4 (August 1988). 113-120.

[Kurlander88b] Kurlander, David, and Feiner, Steven. Editable Graphical Histories. In *1988 IEEE Workshop on Visual Languages* (Pittsburgh, PA, October 10-12, 1988). IEEE Press. 127-134. Reprinted in *Visual Programming Environments: Applications and Issues.* E.P. Glinert, ed. IEEE Press, Los Alamitos, CA. 1990. 416-423.

[Kurlander89] Kurlander, David. Editor Extensibility: Domains and Mechanisms. Technical Report CUCS-516-89. Columbia University, Computer Science. May 1989.

[Kurlander90] Kurlander, David and Feiner, Steven. A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands. In *Visual Languages and Visual Programming*, Shi-Kuo Chang, ed. Plenum Press, New York. 1990. 257-275.

[Kurlander91] Kurlander, David and Feiner, Steven. Editable Graphical Histories: The Video. *SIGGRAPH Video Review.* Issue 63. 1991. Abstracted in *CHI '91 Conference Proceedings* (New Orleans, LA, April 27- May 2, 1991). 451-452.

[Kurlander92a] Kurlander, David and Feiner, Steven. Interactive Constraint-Based Search and Replace. In *CHI '92 Conference Proceedings* (Monterey, CA, May 3-7, 1992). ACM, New York. 609-618.

[Kurlander92b] Kurlander, David, and Feiner, Steven. A History-Based Macro by Example System. In *Proceedings of UIST '92* (Monterey, CA, November 15-18). ACM, New York, 1992. 99-106. Reprinted in *Watch What I Do: Programming by Demonstration.* Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 323-338.

[Kurlander93a] Kurlander, David. Chimera: Example-Based Graphical Editing. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 271-290.

[Kurlander93b] Kurlander, David, and Feiner, Steven. A History of Editable Graphical Histories. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 405-413.

[Kurlander93c] Kurlander, David. Graphical Editing by Example: A Demonstration. Videotape. *SIGGRAPH Video Review*. Issue 89. 1993. Abstracted in *INTERCHI '93 Conference Proceedings* (Amsterdam, The Netherlands, April 24 - 29, 1993). 529.

[Kurlander93d] Kurlander David, and Feiner, Steven. Inferring Constraints from Multiple Snapshots. *ACM Transactions on Graphics 12,* 4. (October 1993). Pages to be determined.

[Lamport86] Lamport, Leslie. *LaTeX User's Guide and Reference Manual.* Addison-Wesley, Reading, MA. 1986.

[Lee83] Lee, Kunwoo. Shape Optimization of Assemblies Using Geometric Properties. Ph.D. Thesis. MIT. Mechanical Engineering. December 1983.

[Letraset87] Letraset USA. *Ready, Set, Go! 4.0 User's Guide*. ISBN 0-944289-01-0. 40 Eisenhower Dr., Paramus NJ 07653. 1987.

[Levine83] Levine, Martin D. *Vision in Man and Machine*, Chapter 10, McGraw Hill, New York, New York. 1983.

[Lewis88] Lewis, Bil. *GNU Emacs Lisp Manual*. 2nd Draft. September 1988. Final version to be distributed by the Free Software Foundation.

[Lewis90] Lewis, C. NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery. In *Visual Programming Environments: Paradigms and Systems*. E.P. Glinert, ed. IEEE Computer Society Press, Los Alamitos, CA. 1990. 526-546.

[Lieberman84] Lieberman, Henry. Seeing What Your Programs are Doing. *International Journal of Man-Machine Studies 21,* (1984). 311-331.

[Lieberman86] Lieberman, Henry. An Example Based Environment for Beginning Programmers. In *Instructional Science 14*, (1986) 277-292.

[Lieberman93a] Lieberman, Henry. Tinker: A Programming by Demonstration System for Beginning Programmers. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 49-64.

[Lieberman93b] Lieberman, Henry. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 341-358.

[Mainstay91] Mainstay. *ClickPaste: The One-Click-Per-Paste Intelligent Scrapbook*. version 2.1. 5311B Derry Avenue, Agoura Hills, CA 91301. 1991.

[Makkuni87] Makkuni, Ranjit. A Gestural Representation of the Process of Composing Chinese Temples. *IEEE Computer Graphics and Applications, 7,* 12. (December 1987). 45-61.

[Martensson88] Martensson, Bengt. Simple BibTeX Mode for GNU Emacs. As modified by Marc Shapiro and Michael Elhadad. 1988.

[Maulsby89a] Maulsby, David L., Witten, Ian H., and Kittlitz, Kenneth A. Metamouse: Specifying Graphical Procedures by Example. Proceedings of SIGGRAPH '89 (Boston, MA, July 31-August 4, 1989). In *Computer Graphics 23,* 4 (July 1989). 127-136.

[Maulsby89b] Maulsby, David L. *Inducing Procedures Interactively: Adventures with Metamouse*. Masters Thesis. University of Calgary. Department of Computer Science. December 1988.

[Maulsby93a] Maulsby, David L. and Witten, Ian H. Metamouse: An Instructible Agent for Programming by Demonstration. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 155-181.

[Maulsby93b] Maulsby, David. The Turvy Experience: Simulating an Instructible User Interface. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 239-269.

[Meadow91] Meadow, Tony. *System 7 Revealed.* Addison-Wesley, Reading, MA. 1991.

[Microsoft91a] Microsoft Corporation. *Microsoft MS-DOS User's Guide and Reference*. version 5.0. Part No. 070-00367 from Gateway 2000. 610 Gateway Dr., North Sioux City, SD 57049. 1991.

[Microsoft91b] Microsoft Corporation. *Microsoft Word for Windows User's Guide*. version 2.0. Part No. 32756. One Microsoft Way, Redmond WA 98052. 1991.

[Microsoft92] Microsoft Corporation. Object Linking and Embedding: OLE 2 Design Specification. Draft. Version 2.00.09. September 1992.

[Modugno93] Modugno, Francesmary, and Myers, Brad. A. Graphical Representation and Feedback in a PBD System. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 415-422.

[Müller-Brockmann81] Müller-Brockmann, J. *Grid Systems in Graphic Design*. Verlag Arthur Niggli, Niederteufen, Switzerland, 1981. Translated by D. Stephenson.

[Myers86] Myers, Brad A., and Buxton, William. Creating Highly Interactive and Graphical User Interfaces by Demonstration. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics 20,* 4 (August 1986). 249-268.

[Myers88] Myers, Brad A. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

[Myers91a] Myers, Brad A. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. In *CHI '91 Conference Proceedings* (New Orleans, LA, April 27- May 2, 1991). 243-249.

[Myers91b] Myers, Brad A. Text Formatting by Demonstration. In *CHI '91 Conference Proceedings* (New Orleans, LA, April 27- May 2, 1991). 251-256.

[Myers92] Myers, B. A. Demonstrational Interfaces: A Step Beyond Direct Manipulation. *IEEE Computer 25,* 8. (August 1992). 61-73.

[Myers93] Myers, Brad. A. Peridot: Creating User Interfaces by Demonstration. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 125-153.

[Nelson85] Nelson, Greg. Juno, A Constraint-Based Graphics System. Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22-26, 1985). In *Computer Graphics 19,* 3 (July 1985). 235-243.

[Nix83] Nix, Robert Peter. Editing by Example. Ph.D. Thesis. Yale University. Department of Computer Science. Research Report 280. August 1983.

[Olsen88] Olsen, Dan. R. Jr., and Dance, J. R. Macros by Example in a Graphical UIMS. *Computer Graphics and Applications 8,* 1 (January 1988). 68-78.

[Olsen90] Olsen, Dan R., Jr., and Allan, Kirk. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings of UIST '90* (Snowbird, Utah, October 3-5). ACM, New York, 1990. 102-107.

[Palay88] Palay, Andrew J., Wilfred, Hansen J., Sherman, Mark, Wadlow, Maria G., Neuendorffer, Thomas P., Stern, Zalman, Bader, Miles, and Peters, Thom. The Andrew Toolkit—An Overview. *USENIX Winter 1988 Conference Proceedings*, February 1988. 9-21.

[Palermo80] Palermo, Frank, and Weller, Dan. Some Database Requirements for Pictorial Applications (Florence, Italy, June 1979). Edited by A. Blaser. In *Lecture Notes in Computer Science, 81*. Springer-Verlag, Berlin, Germany. 1980.

[Pavlidis78] Pavlidis, Theo. A Review of Algorithms for Shape Analysis. *Computer Graphics and Image Processing 7,* 2 (April 1978). 243-258.

[Pavlidis85] Pavlidis, Theo and Van Wyk, Christopher J. An Automatic Beautifier for Drawings and Illustrations. Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22-26, 1985). In *Computer Graphics 19,* 3 (July 1985). 225-234.

[Paxton87] Paxton, Bill. The Tioga Editor. Internal memo. Xerox PARC Computer Sciences Laboratory. 1984, updated in 1985 and 1987.

[Pier83] Pier, Kenneth A. A Retrospective on the Dorado, a High-Performance Personal Computer. *Proceedings of the 10th Symposium on Computer Architecture*. SIGARCH/ IEEE, (Stockholm, Sweden, June 1983). 252-269.

[Pier88] Pier, Ken, Bier, Eric, and Stone, Maureen. An Introduction to Gargoyle: An Interactive Illustration Tool. *Document Manipulation and Typography,* J. C. van Vliet (ed.), Cambridge University Press, 1988. Proceedings of the EP '88 International Conference on Electronic Publishing Document Manipulation and Typography, Nice, France, April 20-22, 1988. 223-238.

[Pike87] Pike, Rob. The text editor **`sam`**. *Software Practice and Experience 17,* 11 (November 1987). 813-845.

[Potter93a] Potter, Richard. Triggers: Guiding Automation with Pixels to Achieve Data Access. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 361-380.

[Potter93b] Potter, Richard, and Maulsby, David. A Test Suite for Programming by Demonstration. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 539-591.

[Press88] Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[Rubine91] Rubine, Dean. Specifying Gestures by Example. Proceedings of SIGGRAPH '91 (Las Vegas, NV, July 28-August 2, 1991). In *Computer Graphics 25,* 4 (July 1991). 329-337.

[Sankoff83] Sankoff, David, and Kruskal, Joseph B. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA. 1983.

[Shneiderman83] Shneiderman, Ben. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer 16*, 8 (August 1983). 57-69.

[Smith77] Smith, David Canfield. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought.* Birkhauser, Verlag, Basel, Stuttgart. 1977. Reprint of 1975 Stanford Computer Science Ph.D. thesis.

[Smith78] Smith, Alvy Ray. Paint. NYIT Computer Graphics Lab Technical Memo No. 7, Old Westbury, NY, July 20, 1978. Also available in Beatty, J. and Booth, K. (eds.), *IEEE Tutorial: Computer Graphics*, 2nd Edition. Silver Spring, MD: IEEE Computer Society Press. 1982. 501-515.

[Smith82] Smith, David Canfield, Irby, Charles, Kimball, Ralph, Harslem, Eric. Designing the Star User Interface. *Byte 7,* 4. (April 1982). 242-287.

[Smith84] Smith, Alvy Ray. Plants, Fractals, and Formal Languages. *Proceedings of SIGGRAPH '84* (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics 18*, 3 (July 1984). 1-10.

[Smith93] Smith, David Canfield. Pygmalion: An Executable Electronic Blackboard. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 19-47.

[Stallman84] Stallman, Richard M. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. In *Interactive Programming Environments*, David Barstow, Howard Shrobe, and Eric Sandewall, ed. McGraw-Hill, New York, 1984.

[Stallman87] Stallman, Richard. *GNU Emacs Manual*, Sixth Edition, Version 18. Free Software Foundation, Cambridge, MA. March 1987.

[Stiny75] Stiny, George. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser, Verlag, Basel, Switzerland. 1975.

[Sutherland63a] Sutherland, Ivan E. Sketchpad, A Man-Machine Graphical Communication System. Ph.D. Thesis. MIT. Electrical Engineering. January 1963.

[Sutherland63b] Sutherland, Ivan E. Sketchpad: A Man-Machine Graphical Communication System. AFIPS Conference Proceedings, Spring Joint Computer Conference. 1963. 329-346.

[Swinehart86] Swinehart, Daniel C., Zellweger, Polle Z., Beach, Richard J. and Hagmann, Robert B. A Structural View of the Cedar Programming Environment. Xerox PARC Technical Report CSL-86-1. 3333 Coyote Hill Rd., Palo Alto, CA 94304. June 1986.

[Symbolics86] Symbolics, Inc. *Text Editing and Processing.* Volume 3 of Genera 7.0 Documentation. 4 New England Tech. Center, 555 Virginia Rd., Concord, MA 01742. July 1986.

[Teitelman84] Teitelman, Warren. The Cedar Programming Environment: A Midterm Report and Examination. Xerox PARC Technical Report CSL-83-11. 3333 Coyote Hill Rd., Palo Alto, CA 94304. June 1984.

[Thomas85] Thomas, Robert H., Forsdick, Harry C., Crowley, Terrence R., Schaaf, Richard W., Tomlinson, Raymond S., Travers, Virginia M. Diamond: A Multimedia Message System Built on a Distributed Architecture. *Computer, 18,* 12 (December 1985). 65-78.

[Turransky93] Turransky, Alan. Using Voice Input to Disambiguate Intent. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 457-464.

[Unilogic85] Unilogic, Ltd. SCRIBE Document Production System User Manual. Fourth Edition. Suite 240, Commerce Court, 4 Station Square, Pittsburgh, PA 15219. June 1985.

[Vander Zanden89] Vander Zanden, B. T. Constraint Grammars—A New Model for Specifying Graphical Applications. In *CHI '89 Proceedings* (Austin TX, April 30-May 4, 1989). ACM, New York, 1989, 325-330.

[Weitzman86] Weitzman, L. DESIGNER: A Knowledge-Based Graphic Design Assistant. ICS Report 8609. University of California, San Diego. July 1986.

[Weller76] Weller, Dan, and Williams, Robin. Graphic and Relational Data Base Support for Problem Solving. Proceedings of SIGGRAPH '76 (Philadelphia, PA, July 14-16, 1976). In *Computer Graphics 10,* 2 (Summer 1976). 183-189.

[Witten93] Witten, Ian H., and Mo, Dan H. TELS: Learning Text Editing Tasks From Examples. In *Watch What I Do: Programming by Demonstration*. Allen Cypher, ed. MIT Press, Cambridge, MA. 1993. 183-203.