

Graphical Editing by Example in CHIMERA

David Kurlander

**450 Computer Science Building
Columbia University
New York, NY 10027**

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 many times 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 paper 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 to be discussed here address both types of repetition. By incorporating new techniques to automate repetition in an application, the programmer makes the application more *extensible*. Extensibility is important in that it enables users to customize applications for tasks that they often perform, making them more efficient. Experienced users, or

those with a particular technical skill, can also encapsulate their knowledge in a form that others might invoke, thereby benefitting the entire user population.

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 MacDraw and Adobe Illustrator, although some of these techniques apply to 3D editing and other domains as well. Graphical editors are an apt focus for this research, since they can be used for many different tasks, such as constructing technical illustrations, organizational charts, network diagrams, flow charts, and architectural drawings. Since the 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 a more interesting target for this work. 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 here 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 here hopefully will be of wide interest, and potentially be quite useful. By choosing a domain with which other researchers have worked, I can compare and contrast

my 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, we have already used these techniques to construct interfaces as well as illustrations.

2 Example-Based Techniques

In all of the new techniques presented here, the user indicates a desired repetition, at least in part, by presenting an *example*. Hence, these methods are all example-based or *demonstrational techniques* [Myers90], in which the user provides 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 if the user does have programming experience, they 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 requires 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 speci-

fied using concrete examples, so those people that have difficult working with abstractions will probably find these interfaces easier to use..

However, the mapping from concrete examples to abstractions is often many-to-one, 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 mapping, or requiring that the user explicitly choose a particular mapping. 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 many complex extensions, and will be so for the foreseeable future. However, there are many extensions that can be expressed without programming, and the goal here is to identify classes of these and develop new demonstrational techniques for expressing them.

3 CHIMERA

To test the ideas contained in this paper, I have built the CHIMERA editor system. CHIMERA is actually an editor framework or substrate, in which other editors modes are embedded. Three different editor modes are currently present in CHIMERA: modes for editing graphics, interfaces, and text. The methods presented in the next section have been implemented so that they work in both the graphics and interface modes. Both the graphics and interface modes of CHIMERA are very complete, and have extensive coverage of the primitives and commands one might want in such editors. The graphics mode of CHIMERA can create and manipulate boxes, circles, ellipses, lines, text, arcs, beziers, freehand curves, beta splines, and cardinal splines (cyclic and non-cyclic). The interface mode 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 mini-buffers. All of

CHIMERA's interfaces were generated in CHIMERA. Both the graphics and interface editing modes have around 180 commands (most of them shared).

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 in the subsequent subsections, along with examples of their use. All figures 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 [Kurlander92a].

4.1 Graphical Search and Replace

Often shapes are repeated many times in a single illustration. Similarly, many illustrations contain the same graphical attributes, 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 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 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

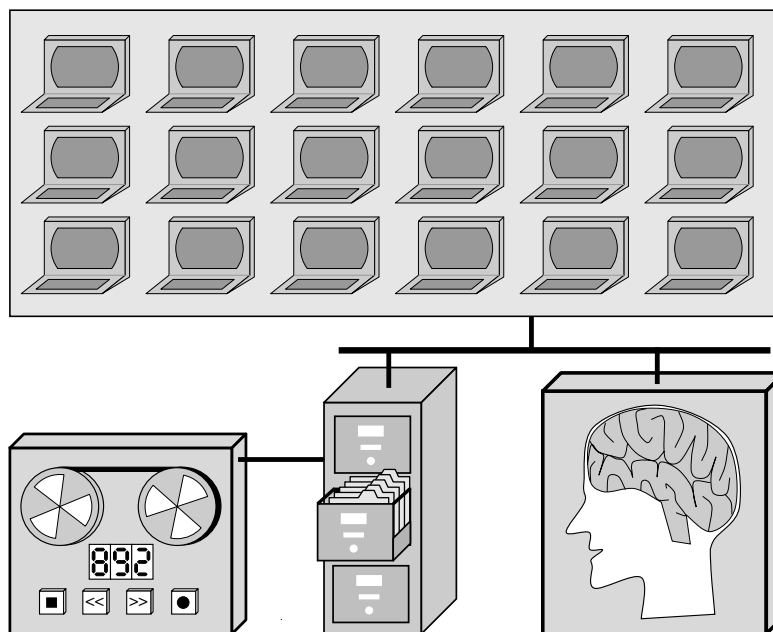


Figure 1 A network diagram drawn in CHIMERA.

a terminal and replace it with a drawing of a workstation, but this would be repetitious. 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 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 attributes 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 thickness, and replace these attributes or others. Below the search and replace panes and to the left are rows of graphical attributes followed by checkboxes. The first column of checkboxes specifies which attributes of the objects in the search pane must be present in each match. During a replacement, the second column specifies which attributes of the objects in the replace pane will be applied to the match.

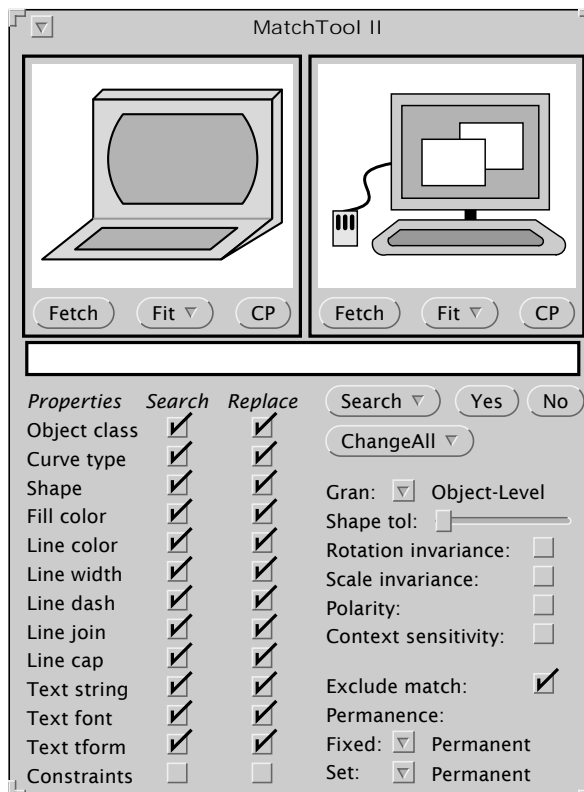


Figure 2 A MatchTool 2 window containing a search and replace specification that changes drawings of terminals to workstations.

Here we check nearly all the graphical attributes in the two columns since we want to match and replace all of these properties, and then press the ChangeAll button. Figure 3 shows the resulting scene.

Since the search and replace panes contain example objects as part of the specification, this technique is example-based. These single 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 repetition in addition to the recurrence of terminals or workstations. The same fill color is used in many of the boxes. 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, and triangular holes in the reels of tape, and

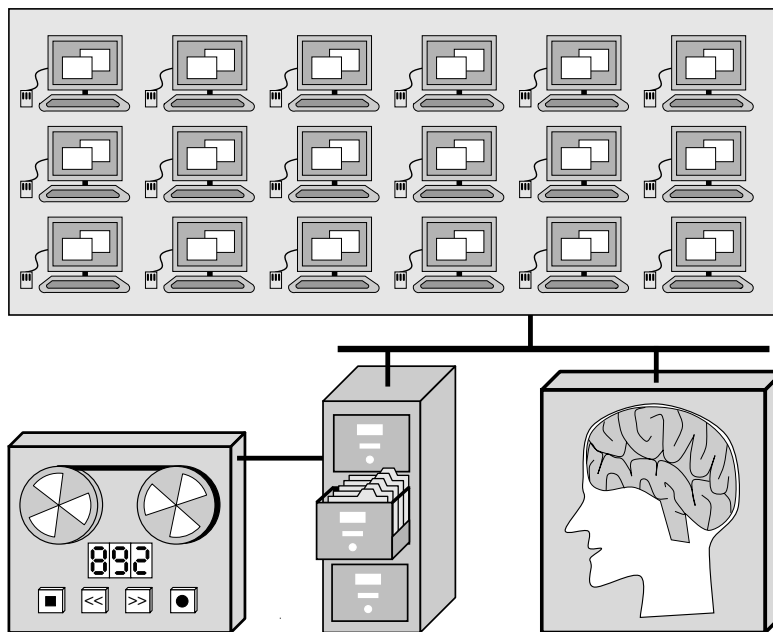


Figure 3 The network diagram of Figure 1, after applying the graphical search and replace specification of Figure 2. All terminals are replaced by workstations.

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 discussed in [Kurlander88], 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.

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. 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 relation-

ships, 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 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 4, and contain graphical objects and constraints as they are visually represented in CHIMERA. The search pane in Figure 4 (a)

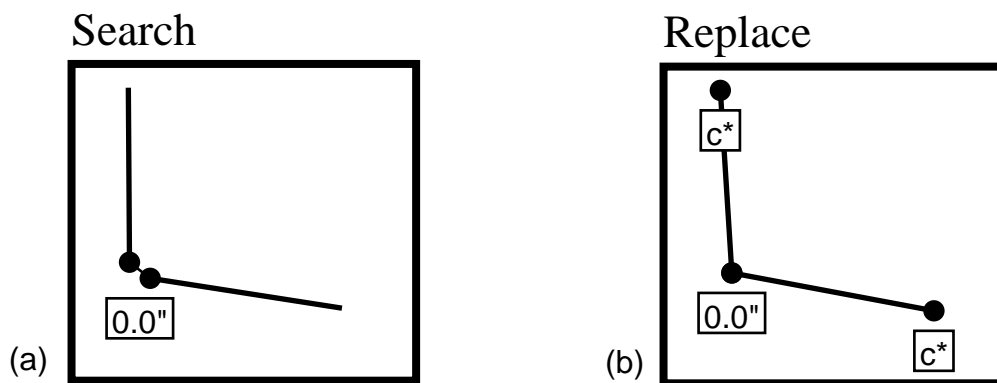


Figure 4 Constraint-based search and replace pattern to connect nearly connected lines. (a) the search pane; (b) the replace pane.

contains two lines with endpoints joined by a 0 inch distance constraint. Note 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 4 (b) 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 5 shows the result of applying this rule to a rough drawing of a W. All of the segments in Figure 5 (a) are nearly connected, so they become precisely connected after the replacement as shown in Figure 5 (b). Both graphical and constraint-based search and 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 is discussed in [Kurlander 92], this technique can be used for a variety of scene transformations, including illustration beautification.

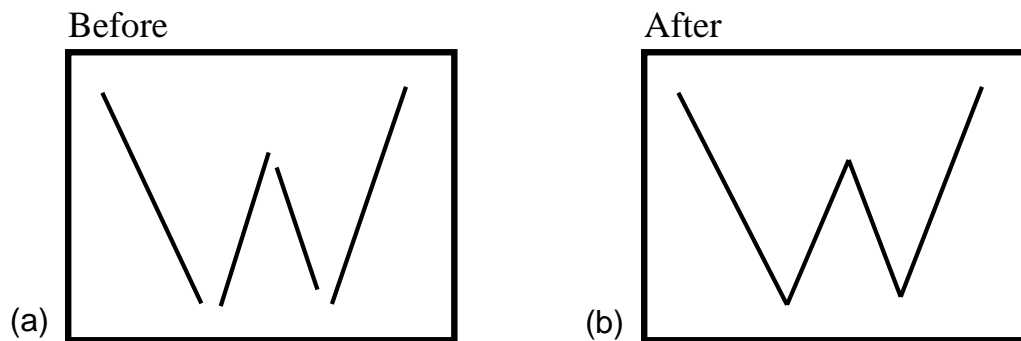


Figure 5 Application of the new rule. (a) a scene before the replacements; (b) the scene after the replacements.

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 instantiated. This technique uses examples of valid scene configurations to automatically determine these constraints.

For example, Figure 6a shows a drawing of a balance. After completing this initial

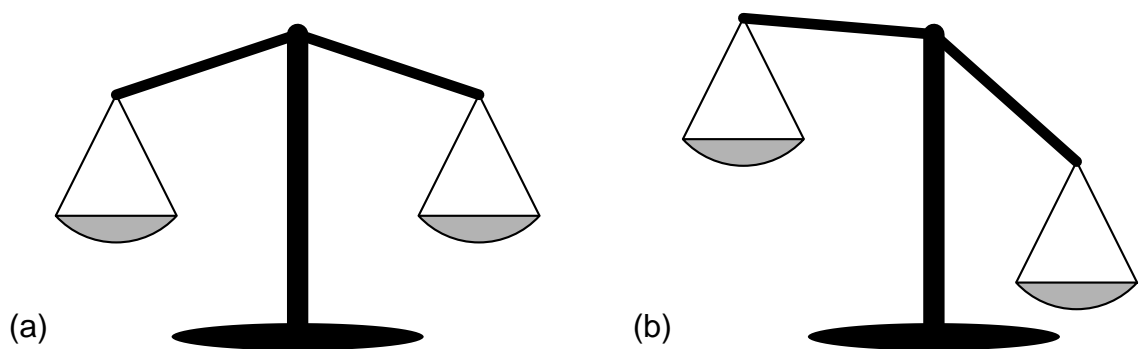


Figure 6 Two snapshots of a balance. (a) initial snapshot; (b) subsequent snapshot.

drawing, the user presses a button labeled “Snapshot”, to indicate this illustration is a valid

configuration of the scene. Next, the user rotates the arm of the balance on its axis, and translates each of the trays so that they are still connected to the arms, to produce the illustration shown in Figure 6b. They press the “Snapshot” button once more. The system calculates and instantiates constraints that are present in each snapshot. For example, the base remains fixed, 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 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 or that not enough snapshots were given to break all the undesired constraints. 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.

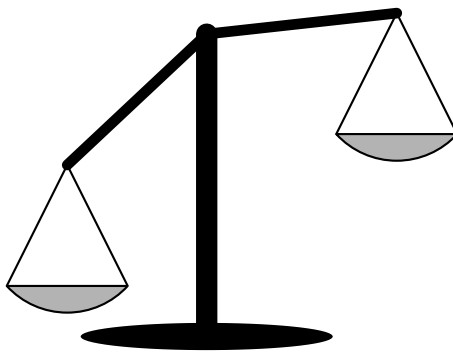


Figure 7 A third configuration of the balance, generated by the system, when one end of the arm is moved.

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 [Kurlander91]. Having constraints in an illustration make 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.

4.4 Editable Graphical Histories

Another approach to reduce 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 performed in a special recording mode. Other systems detect repetitions and automatically extract out the repeated elements themselves [Cypher91]. 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 discussed here, *editable graphical histories*, is a representation for commands in a graphical user interface.

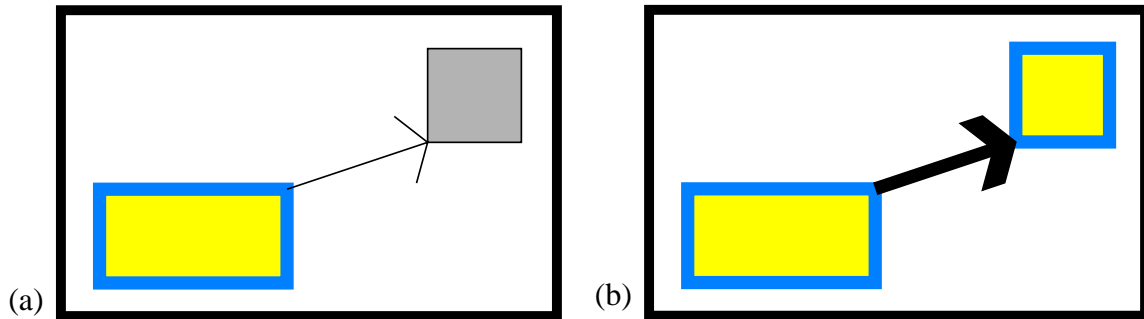


Figure 8 Two versions of a simple scene. (a) the original scene; (b) the scene after the manipulations discussed in this section.

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 8a containing two boxes and an arrow. The history generated during its construction is shown in Figure 9. Though Figure 9 appears to include two history windows, the figure really shows two successive scrolls of a single window.

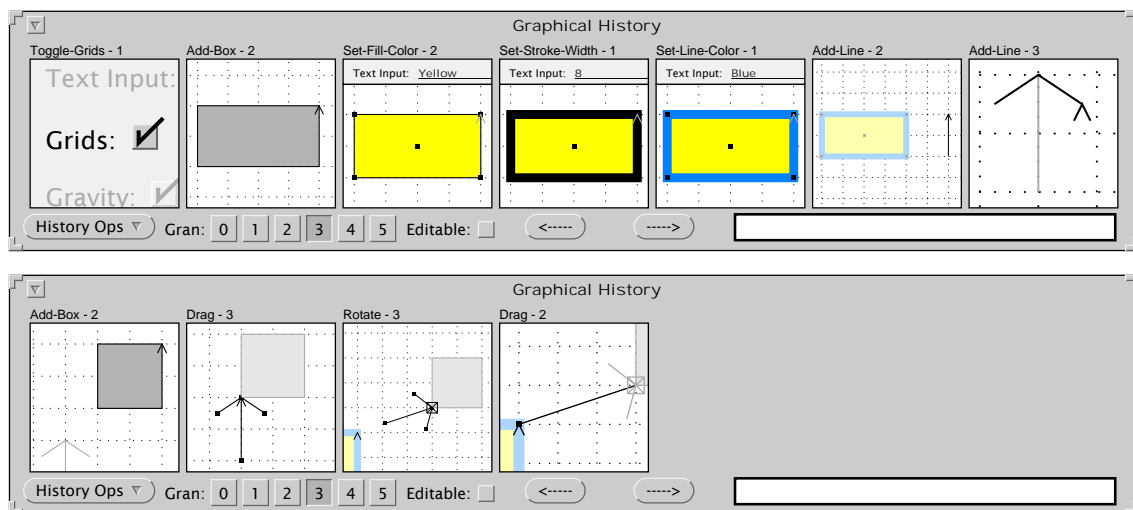


Figure 9 Editable graphical history that generated Figure 8a.

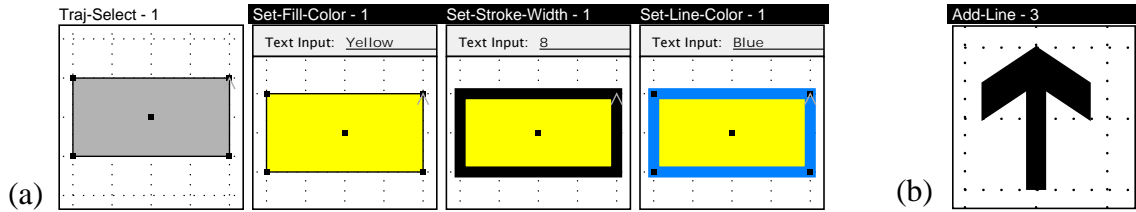


Figure 10 Using graphical histories to change a scene. (a) the three latter panels selected for redo; (b) new operations added in place, in the history.

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 in the editor scene using the Add-Box command. In the third panel the box is selected, a color (yellow) was 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 shows changes to the rectangle's stroke width and line color, a line being added beside the rectangle, and two lines being added above the first to create a handdrawn 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. For example, the third panel contains two operations: one to select a scene object, and the other to change the fill color of selected objects. Each panel's label indicates the number of commands that it represents. We can expand high-level panels into lower-level ones and vice versa. This panel expands into the first two shown in Figure 10a. So that the history panels will be less cluttered, each panel shows only those objects that participate in its 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 are important, and they 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, we would like to apply to the upper rectangle the commands that set the fill color, stroke width, and line color of the lower one. We select this rectangle, find the relevant panels in the history, and select them too. These are the last three panels of Figure 10a, and panel selections are indicated by white labels on a black background. Next we execute the Redo-Selected-Panels command, and the top rectangle changes appropriately.

Editable graphical histories reduces 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 9 to draw a fancier arrowhead and change the width of the arrow's base. The new panel is shown in Figure 10b. We modify this history panel rather than the editor scene directly, since at this point in time the arrow is still aligned with the grid axes and later the change would be more difficult. After propagating these changes

into the history, a new scene results as shown in Figure 8b. [Kurlander90] contains a more detailed treatment of this history representation..

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 example*, and one of the major challenges involves generalizing the demonstrated commands to work in different contexts. Another challenge is in providing a visual representation of these programs. CHIMERA includes a programming by example 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 rectangles. The steps are captured in the graphical history of Figure 11. Initially we create the two rectangles (panels 1 and 2). Next we turn on 0 and 90 degree slope alignment lines (panels 3 and 4), and select the upper left corner of the bottom rectangle (panel 5) and lower right corner of the top rectangle (panel 6) to generate these lines. Finally we select the top rectangle, and drag it until it snaps to the appropriate intersection of two alignment lines (panel 7).

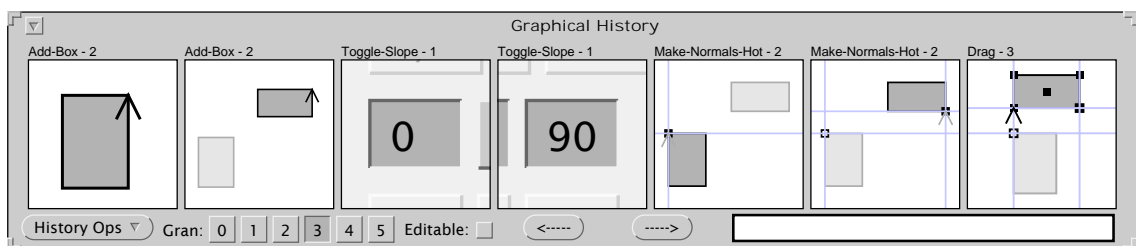


Figure 11 Graphical history showing the creation of two rectangles, and the left-alignment of the top rectangle with the bottom one.

At some later time 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 rectangle from a panel, name it “fixed” since it doesn’t move, and execute Make-Argument. We do the same for the other rectangle, but call it “moved” since this is the rectangle that was translated. Figure 12 shows the resulting macro builder window, with the argument declaration panels just created.

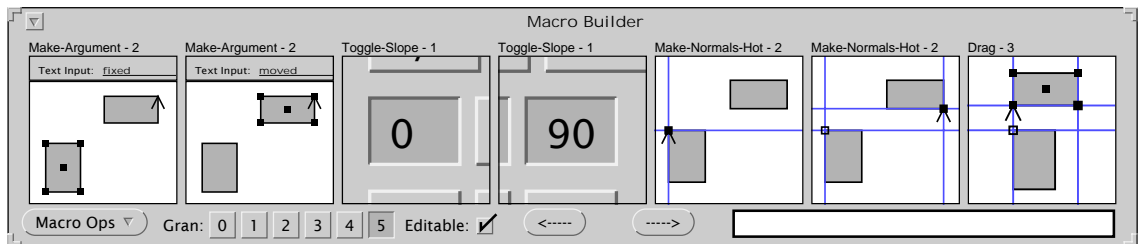


Figure 12 Macro builder window, containing a macro to left-align to rectangles.

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 rectangles. A macro invocation window appears, as shown in Figure 13 that allows us to set and view the arguments. We test this

macro on a sample scene, and it works as expected. A later chapter in this book discusses the many ways that editable graphical histories support the macro definition process.

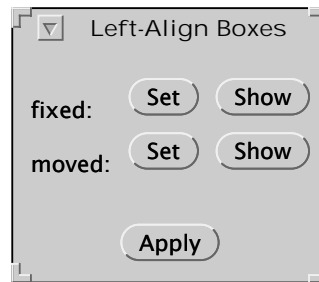


Figure 13 A macro invocation window.

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 with synergistic relationships. Figure 14 is a graph with the five techniques represented as nodes, and edges that 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.

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.*

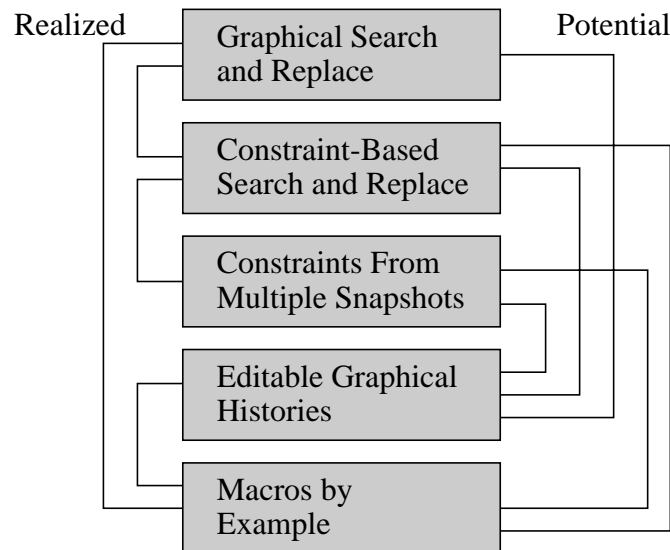


Figure 14 Relationships between the five techniques. Edges on the left represent existing relationships, and those on the right potential relationships.

- *Graphical search provides an iteration construct for macros by example. The system can execute a macro for all objects fitting a given graphical description.*

Listed next are relationships that could potentially link the 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 14 would be a complete graph if it were not for the missing edge between graphical search and replace and constraints from multiple snapshots. Those suggesting a plausible relationship will earn my undying gratitude.