INTERACTIVE CONSTRAINT-BASED SEARCH AND REPLACE

David Kurlander
Steven Feiner

Department of Computer Science
Columbia University
New York, NY 10027

E-Mail: {djk, feiner}@cs.columbia.edu

ABSTRACT

We describe enhancements to graphical search and replace that allow users to extend the capabilities of a graphical editor. Interactive constraint-based search and replace can search for objects that obey user-specified sets of constraints and automatically apply other constraints to modify these objects. We show how an interactive tool that employs this technique makes it possible for users to define sets of constraints graphically that modify existing illustrations or control the creation of new illustrations. The interface uses the same visual language as the editor and allows users to understand and create powerful rules without conventional programming. Rules can be saved and retrieved for use alone or in combination. Examples, generated with a working implementation, demonstrate applications to drawing beautification and transformation.

KEYWORDS: Constraint specification, interactive techniques, demonstrational techniques, editor extensibility, graphical editing.

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. Previously we adapted this technique to the 2D graphical editing domain by building a utility called the MatchTool [8]. Using the MatchTool, we could search for all objects matching a set of graphical attributes, such as a particular fill color, line style, or shape, and change either these or a different set of attributes. 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

Reprinted from *CHI '92 Proceedings*, May 1992, Monterey, CA, 609-618.

these relationships precisely [17]. Myers' Peridot, an interactive system for designing new interface widgets, uses a rule set to find particular relationships between pairs of scene objects and establish new constraints among them [13]. Our system 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 [9]. Rules are defined by direct manipulation [19], 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 [14] 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 [2], and several recent systems allow new constraints to be entered via spreadsheets [5][11][15]. 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 [20] 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 creation, 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, a two-view constraint-based graphical editor, 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 [16]. Our 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 [22]. 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 [12]. 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 [21].

We have implemented constraint-based search and replace as part of MatchTool 2, an application that works in conjunction with the graphics and interface editing modes of the Chimera editor [10]. Here we discuss the motivation, interface, and implementation of constraint-based search and replace. We introduce the capabilities of our system and its interface through a series of examples. Next we discuss the algorithm and implementation. The last section presents our conclusions and planned future work. All figures in this paper are working examples, generated by Chimera's PostScript output facility.

EXAMPLE 1: MAKING A NEARLY RIGHT ANGLE RIGHT

Suppose that we would like to specify that all pairs of connected lines *nearly* 90 degrees apart should be *precisely* 90 degrees apart. Figure 1 shows MatchTool 2's interface for doing this.

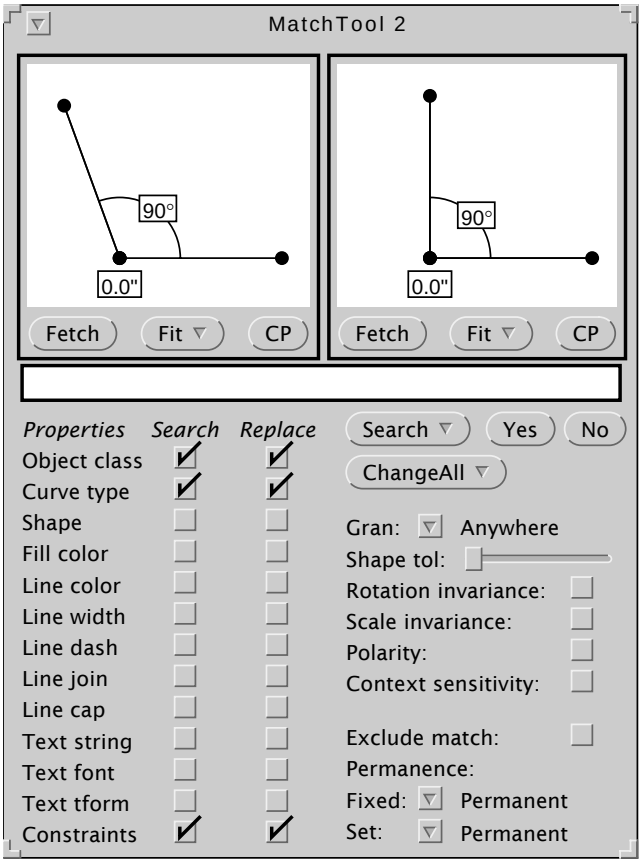

FIGURE 1. The MatchTool 2 interface.

At the top are two graphical editor panes in which objects can be drawn directly or fetched from any editor scene. The *Search Pane* on the left contains the search objects, and the *Replace Pane* on the right contains the replacement objects. Below these and to the left are two columns of checkboxes, the *Search* and *Replace Columns*. These specify which graphical properties of the objects in the Search and Replace Panes will be looked for or substituted into the match. Here we have selected "Object class", "Curve type" and "Constraints" from both the Search and Replace Columns. Starting at the middle right of the window is a set of buttons ("Search", "Yes", "No", and "ChangeAll") that invoke the search and replace operations.

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 1. The zero length distance constraint connects an endpoint of each line. It indicates that the MatchTool should look for two segments that touch at their endpoints. The 90 degree 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.

TOLERANCES BY EXAMPLE

We intend that the pattern should match all angles that are roughly 90 degrees, so we need a way to specify the tolerance of the search. We use a simple demonstrational technique. When system 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. In the above example, the angle match will be 90 ± 20 degrees, since the angle drawn is 110 degrees, and the constraint specifies 90 degrees. To represent an asymmetric range (e.g. 90 +0 –20 degrees), we can simply convert it into a symmetric range about a different value (e.g. 80 ± 10 degrees). In the search specified by Figure 1, the distance constraint must be satisfied exactly since the endpoints are coincident 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.

SEARCH AND REPLACE PARAMETERS

In the lower right hand corner of Figure 1, underneath the "ChangeAll" button, is a set of controls that affect how the search and replace is performed. Most of the parameters from the original MatchTool are still useful, but others have been added as well. The new controls appear at the bottom of Figure 1, starting with "Exclude match".

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 degrees, 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.

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 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.

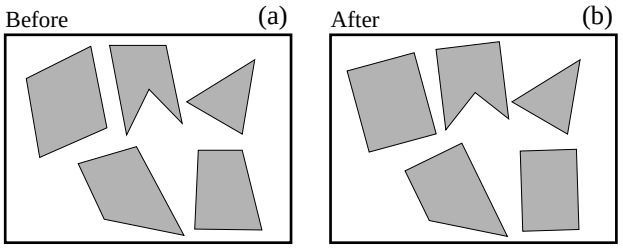

FIGURE 2. Application of the rule to make right angles out of nearly right angles. (a) The initial editor scene; (b) The modified scene.

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 2(a), 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 degree angles become true right angles, as shown in Figure 2(b).

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 3.

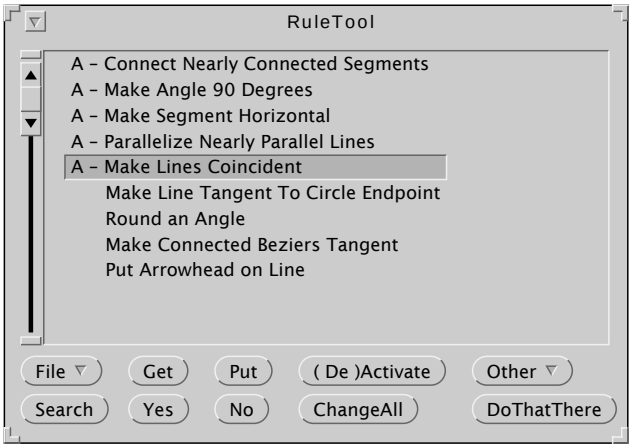

FIGURE 3. The RuleTool interface, a utility for building libraries of rules.

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 shows the results of executing the activated rules of Figure 3 on a rough drawing of a house. Figure 4(a) is the initial drawing, and Figure 4(b) is the neatened version. The

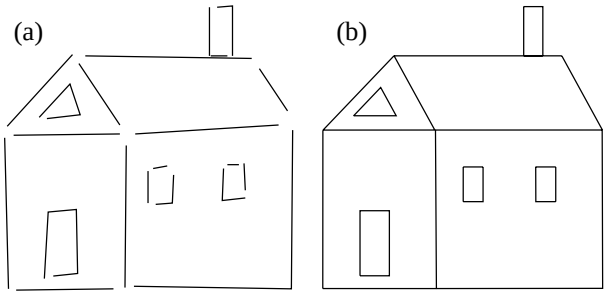

FIGURE 4. Renovating a house. (a) The original house; (b) The house after application of the activated rules in the rule set of Figure 3.

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. In the next two examples we demonstrate more sophisticated tasks that introduce other capabilities of the system.

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 [4]. An example of this is included in [1]. 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 the letters before us and presumably would have written the macros. One particular task that we found difficult was making a line tangent to a circle through a particular point. 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 5(a). 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 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 degrees. (Actually, there are two lines tangent to a circle through a point, and we should be looking for –90 degree 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 5(b).

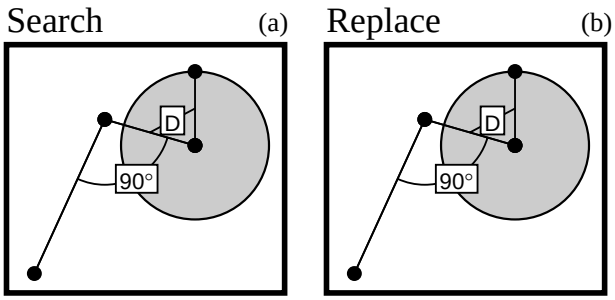

FIGURE 5. Search Pane (a) and initial Replace Pane (b) for line-endpoint and circle tangency.

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 adapt 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 6. 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.

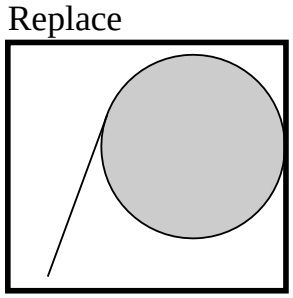

FIGURE 6. A test reveals a problem in the replacement specification.

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 degrees. 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 the 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 interfaces for specifying constraints, either of which may be used in a MatchTool 2 pane or in an arbitrary scene. The user can select commands from the FIX menu to fix relationships at their current value, or alterna-

tively 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. 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 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 7(a), our rule makes all line endpoints that are nearly tangent to circles, truly tangent, resulting in the scene shown in Figure 7(b).

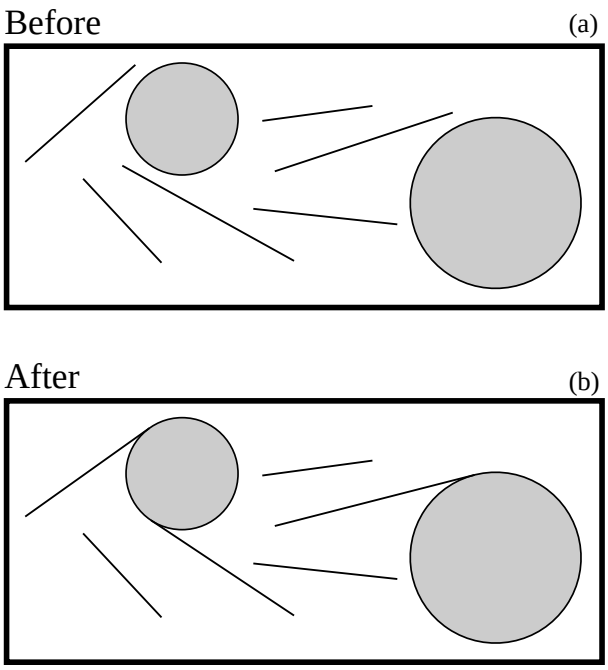

FIGURE 7. Applying the tangency rule. (a) a scene containing lines and circles; (b) nearly tangent lines made tangent.

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 high, and by slowing down the editor. An extension to the current system that would allow tolerances to be scaled down 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 the 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.

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 degree 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 degree 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 beautifiers.

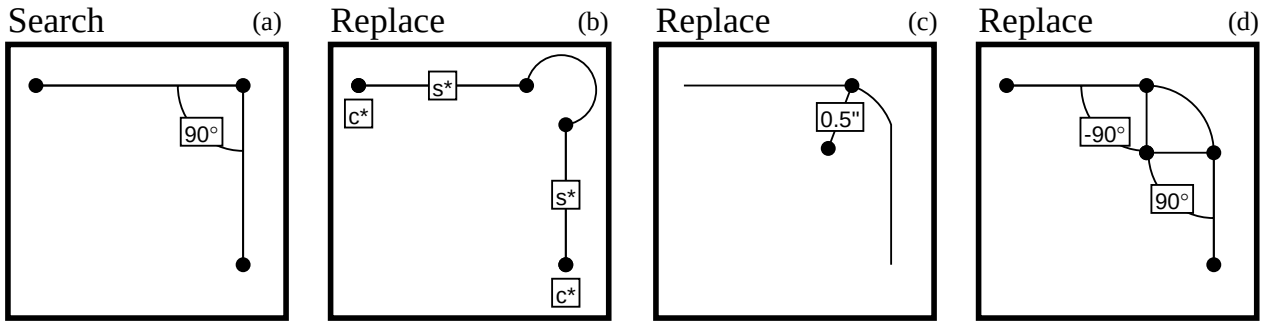

FIGURE 8. Rounding corners. (a) Search Pane; (b) Replace Pane after splicing in an arc, and adding fixed location and fixed slope constraints (labeled c* and s*, respectively); (c) Replace Pane after adding a 1/2 inch distance constraint on the arc's radius (the figure has been reduced); (d) Replace Pane after adding two more angle constraints.

The search pattern shown in Figure 8(a), matches on two segments, meeting at 90 degrees. 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 representative match that we will, step by step, transform into its replacement. Figure 8(b-d) 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 8(b).

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 8(c). 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 degrees, and the corresponding angle formed with the other line is set to 90 degrees. These final constraints are shown in Figure 8(d) (Note that for each of these figures we have turned off the display of constraints not added by the step.).

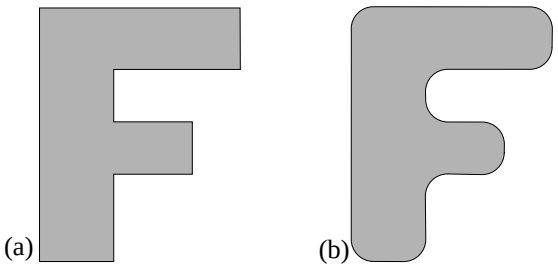

FIGURE 9. Rounding right angles in an F. (a) The unrounded version; (b) After application of our rounding rule. (This figure has been reduced.)

The representative match has now been completely transformed into the desired replacement. After applying the rule to the "F" in Figure 9(a), the corners are correctly rounded, producing Figure 9(b).

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 degrees, provided it can be rounded. We need only add a 90 degree tolerance to the search pattern, making the entire rule work for angles between 0 and 180 degrees. Given two lines that meet at an angle ABC, either this angle or its reverse, angle CBA, is between 0 and 180 degrees. 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 10(a), the rule rounds all the angles, producing Figure 10(b).

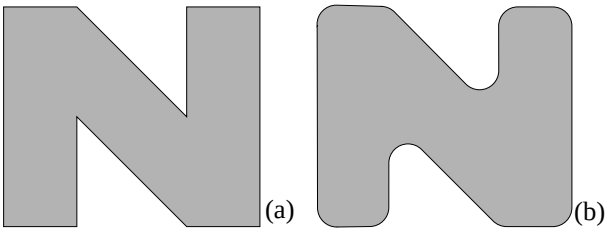

FIGURE 10. Rounding arbitrary angles in an N. (a) The unrounded version; (b) After application of the generalized rule. (This figure has been reduced.)

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 [3]. 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 presorted 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, in advance of searching. We have two mechanisms to do 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, we also copy 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.

IMPLEMENTATION

MatchTool 2 and the RuleTool are both implemented as part of the Chimera editor. Chimera is written mainly in Lucid COMMON LISP (using the Common Lisp Object System), with a little C code thrown in for numerically intensive tasks and window system communications. It runs on Sun Workstations under the OpenWindows 3.0 window system. Chimera uses Levenberg-Marquadt iteration to solve systems of constraints [18], and can currently enforce about 10 different types of geometric relations.

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 7 take 0.11 seconds on a Sun Sparcstation 2 (28.5 MIPS), including the time spent solving constraints.

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, 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 editor. If it cannot, we print 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 [6][7]. 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 degrees 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.

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.

ACKNOWLEDGMENTS

Eric Bier originally suggested the idea of graphical search and replace, and participated in the development of MatchTool 1. Michael Elhadad's comments lead to an improved paper. This work was partially funded by a grant from IBM.

REFERENCES

1. Bier, E. A., and Stone, M. C. Snap-Dragging. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18-22, 1986) In *Computer Graphics 20*, 4 (August 1986). 233-240.

2. Borning, A. 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.

3. Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA. 1979.

4. Goines, D. L. *A Constructed Roman Alphabet.* David R. Godine, publisher. 306 Dartmouth St., Boston, MA 02116. 1982.

5. Hudson, S. E. An Enhanced Spreadsheet Model for User Interface Specification. Report TR 90-33. Univ. of Arizona. Computer Science. October 1990.

6. Hudson, S. E., and Yeatts, A. 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.

7. Karsenty, S., Landay, J. A., and Weikart, C. Inferring Graphical Constraints with Rockit. Research Report. DEC Paris Research Laboratory. In preparation.

8. Kurlander, D., and Bier, E. A. Graphical Search and Replace. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4 (August 1988). 113-120.

9. Kurlander, D. Editor Extensibility: Domains and Mechanisms. Technical Report CUCS-516-89. Columbia University, Computer Science. May 1989.

10. Kurlander, D. *Graphical Editing by Example*. Ph.D. Thesis. Columbia University. Computer Science. In preparation.

11. Lewis, C. NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery. In E. Glinert, *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press, Los Alamitos, CA. 526-546.

12. Maulsby, D. L., Witten, I. H., and Kittlitz, K. 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.

13. Myers, B. A. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

14. Myers, B. A. Demonstrational Interfaces: A Step Beyond Direct Manipulation. Technical Report CMU-CS-90-162. Carnegie Mellon University, School of Computer Science. August 1990.

15. Myers, B. A. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. In CHI '91 Proceedings (New Orleans, LA, April 27-May 2, 1991). ACM, New York. 1991. 243-249.

16. Nelson, G. 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.

17. Pavlidis, T. and Van Wyk, C. 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.

18. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

19. Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer 16*, 8 (August 1983), 57-69.

20. Sutherland, I. E. Sketchpad: A Man-Machine Graphical Communication System. AFIPS Conference Proceedings, Spring Joint Computer Conference. 1963. 329-346.

21. 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.

22. Weitzman, L. DESIGNER: A Knowledge-Based Graphic Design Assistant. ICS Report 8609. University of California, San Diego. July 1986.