Stretch-A-Sketch: a Dynamic Diagrammer

Mark D Gross
College of Architecture and Planning
University of Colorado
Boulder, Colorado 80309-0314
mdg@cs.colorado.edu

Abstract

Stretch-A-Sketch is a pen-based drawing program that combines recognition of hand-drawn glyphs with constraint based maintenance of spatial relations. The recognition program identifies hand drawn glyphs, spatial relations between them, and higher-level configurations, such as graph and tree diagrams and floorplan bubble diagrams. Stretch-A-Sketch then maintains the essential relations in these configurations as the user edits the diagram.

1. Introduction: Diagram editing and visual languages

Visual languages deal in diagrams. Dataflow graphs and Petri nets, analog and digital circuit diagrams, and Venn diagrams are all expressions of visual language. Traditionally, these expressions have been made with pen and paper, for communication between people. To employ diagrams in communicating with computers we need facilities for making, editing, and interpreting these diagrams.

A common way to support a particular visual language is with a structured graphical editor programmed with the syntax of the diagrams to be made. The editor offers a limited vocabulary of elements and it restricts the assembly of elements to syntactically correct spatial relations. For example, a structured flowchart editor provides blocks for imperative and conditional statements and the means to link entry and exit points with connecting lines. Users are thereby allowed to produce only valid language expressions; the structured editing also makes easy for the program to recognize and parse visual expressions for a language interpreter. One goal of visual language research, therefore, is to automatically generate structured editors from a grammar that describes the syntax of a class of diagrams.

Structured editors for visual languages have their problems, however. Users may be disinclined to use a structured editor in the early stages of thinking about a problem precisely because it imposes firm syntactic rules. A structured editor requires commitments to precision and detail that may be inappropriate for a user in the conceptual stages of problem-solving. Also, compared with pen and paper the structured editor is tedious for making a quick sketch. Even an ordinary draw program like MacDraw constrains the shapes of lines and elements and it imposes a menu and palette interface that is awkward and slow to use. Given these alternatives, users may prefer to explore a problem off-line, employing pen and paper because it is direct and unreactive, and switch to a graphical editor only after working out a conceptual scheme.

More generally, let us take visual language to include the diagrams, sketches, and drawings made for design, for example, mechanical and architectural design. Although the syntax of these design drawings may be less strictly defined than languages specifically designed for computation, we find the same course of development from rough sketch to precise and detailed specification. Early and conceptual sketches contain elements and relationships that persist throughout later more precise phases of designing. As with computational visual languages, certain configurations and spatial relationships are syntactically required or forbidden. However, in a design drawing other relationships are also employed by the designer, to achieve functional or stylistic goals.

The goal of Stretch-A-Sketch is to combine some of the advantages of structured editing, in particular knowledge of the domain syntax, with the ease, directness, and flexibility of drawing with pen and paper. With the pen based interface of Stretch-A-Sketch a user can begin to make diagrams without worrying about correctness or completeness. When the user is ready, the program tries to recognize configurations in the diagram according to previously defined syntactic rules, or spatial relations. Once Stretch-A-Sketch recognizes a configuration, the program maintains its essential relations during subsequent editing. For example if the program recognizes that the user has drawn a graph with nodes and arcs, then it maintains connections as the user moves the nodes around. Thus Stretch-A-Sketch aims to smooth the transition between the rough and sometimes ill-formed diagrams of early exploration and the more precise and well formed drawings that characterize the later phases of problem solving.

Stretch-A-Sketch employs two techniques from computer graphics: the recognition of hand drawn shapes and spatial relations and bottom up parsing into more complex configurations, and the maintenance of graphical constraints among drawing elements. In combining these two techniques, Stretch-A-Sketch shows how constraint based drawing, i.e. making diagrams behave according to rules, can be incorporated into a direct pen based interface. The remainder of the paper first reviews the two underlying techniques, then describes their combination in Stretch-A-Sketch. Section 2 presents the recognizer for hand drawn input used in Stretch-A-Sketch and section 3 briefly reviews constraint based graphic editing. Section 4 outlines the features of the Stretch-A-Sketch program and shows an example application, editing a flowchart. Section 5 concludes with a summary and discussion.

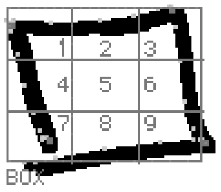
2. The Cocktail Napkin recognizer

This section provides an overview of the hand-drawn diagram recognizer in Stretch-A-Sketch -- the "electronic cocktail napkin" [7]. Recently much work has been done on recognizing hand-drawn input [2, 12, 14]. Recognition rates for these programs are around 95% and any of them might work well for Stretch-A-Sketch. The approach followed here is simple and easy to implement; it can be trained interactively during use; and it is sufficiently fast and accurate. The current version recognizes capital letters, numbers, and simple shapes (circles, triangles, boxes, arrows, etc.) without noticeable delay on a Quadra. Lines of various types (horizontal, vertical, dotted, dashed, and wiggly lines) are recognized as a special case. The algorithm can recognize glyphs drawn in 90 degree rotations, reflections, and reversal of the pen path in varying sizes and aspect ratios. Rotations between 90 degree multiples are handled by training the program with slightly rotated samples.

The diagram recognizer in Stretch-A-Sketch consists of three main parts: (1) a trainable low-level recognizer for hand drawn glyphs; (2) an analyzer that identifies spatial relations among glyphs; and (3) a graphical search and matching procedure to define higher level recognizers for configurations of glyphs in specific spatial relations. Each of these parts is summarized below.

2.1 A trainable recognizer for glyphs

The low-level recognizer reads a stream of x,y, and pressure values from a Wacom digitizing tablet. A glyph begins when the user brings the pen near the tablet and it ends when the pen is removed for longer than a certain time (1/5 second by default). The glyph input routine identifies pen up and pen down events, counts strokes, and finds the glyph's bounding box . The resulting 'raw glyph' is processed by a low-level routine that transforms the stream of point coordinate values into a pen path, a sequence of square numbers through which the pen moved (see figure 2). The low level routine also identifies corners where the pen slowed down and several points were sampled close together; and it records the glyph's approximate size and aspect ratio. The resulting structure is the input glyph, which is compared with a library of previously trained templates.



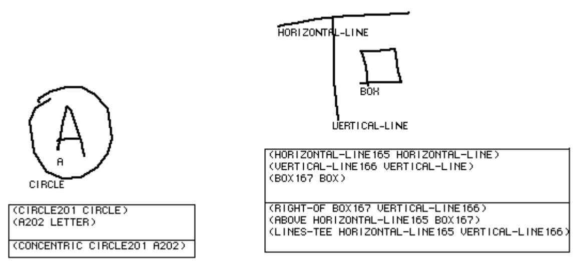
```
GLYPH -- BOX.1676
TYPE BOX
NSTROKES 1
NCORNERS 4
PEN-PATH (7 4 1 2 3 6 9 7)
SIZE LARGE
ASPECT RATIO SQUARE
LOCATION NIL
ROTATION NIL
```

Figure 2. Features of a Box glyph.

The templates in the library identify a set of allowable pen paths, stroke counts, corner counts, aspect ratios, sizes, and rotations for each glyph type. The recognizer compares these characteristics of the input glyph with the previously trained templates. If no candidate is found, matching criteria are relaxed and the match is tried again. If several candidates are found, then the templates most closely matching the input glyph are returned. The recognizer permits both 'no match' and 'multiple candidate' matches, and depending on user switch settings the program asks the user to immediately identify the input glyph or allows it to remain ambiguous. The user can add to the library of templates just by drawing and identifying several instances of the new glyph.

2.2 Identifying spatial relations

Once the recognizer has identified the diagram glyphs, it analyzes spatial relations among them. The program maintains a list of binary spatial relations, which it applies as predicates to the diagram elements. The relations are defined in terms of the bounding box, size, and starting and ending points of elements. Figure 3 shows spatial relations identified for two simple configurations.



```
(CIRCLE201 CIRCLE)
(A202 LETTER)

(CONCENTRIC CIRCLE201 A202)
```

```
(HORIZONTAL-LINE165 HORIZONTAL-LINE)
(VERTICAL-LINE166 VERTICAL-LINE)
(BOX167 BOX)

(RIGHT-OF BOX167 VERTICAL-LINE166)
(ABOVE HORIZONTAL-LINE165 BOX167)
(LINES-TEE HORIZONTAL-LINE165 VERTICAL-LINE166)
```

Figure 3. Spatial relations in simple configurations.

A potential problem that the analyzer must avoid is reporting too many relations, relations that are true but uninteresting. Several methods are used to screen uninteresting or redundant relations. First, spatial relations are organized in a hierarchy of specificity; for example, 'concentric' is more specific than 'contains'. The analyzer identifies <u>the most specific relations</u> that describe the configuration of elements. (The user can relax or modify this description by hand, as described in the following section). Second, spatial relations are also organized by the type of element they apply to. For example, 'connects,' 'intersects,' and 'tee-intersection' apply only to lines; others such as 'contains', or 'overlaps,' apply only to shapes. Finally, the analyzer also consults a table that lists the commutative and transitive properties of relations, which enables it to avoid reporting many redundant relationships. For example, if two elements overlap, the relation need only be reported once.

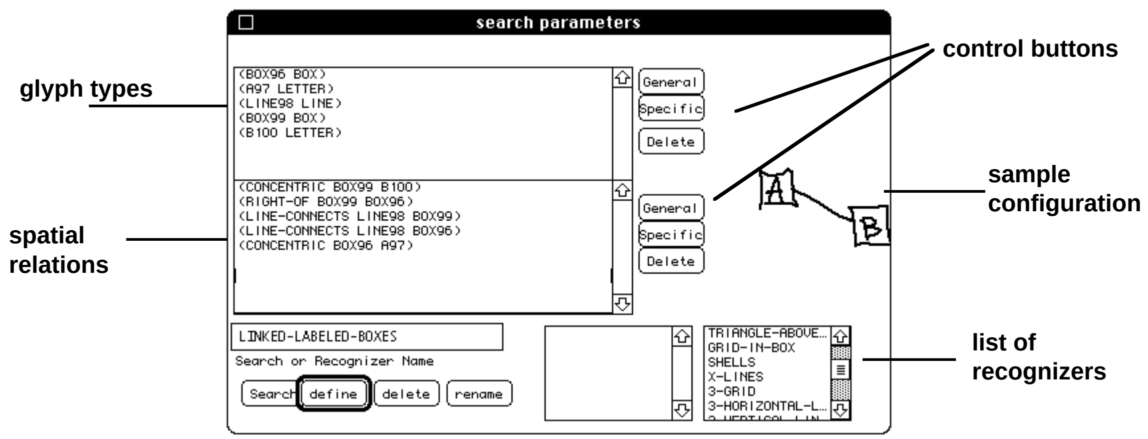
2.3 Defining higher level recognizers

The low level glyph recognizer and the spatial relations analyzer together provide the components for defining higher level recognizers that can identify configurations of diagram elements. The cocktail napkin program provides a dialog for users to define these recognizers. The user first draws an instance of the configuration to be recognized, then views a symbolic description of the glyphs and spatial relations that the program identifies, adjusts the symbolic description, and finally names it, adding the recognizer to a list that the program automatically applies.

After drawing an instance of the configuration, the user can view and adjust the symbolic description produced by the analyzer using a 'search parameters' dialog. The dialog (figure 4) displays the configuration, lists its elements with their types and the spatial relations among them, and provides control buttons for adjusting the description. Using the control buttons the user can (1) delete unwanted elements or relations from the description; (2) make element types more general or more specific; and (3) make spatial relations more general or specific.

The four buttons at the lower left of the dialog are used to test, define, and manage the list of higher level recognizers. The 'search' button allows the user to apply a recognizer to the diagram to search for instances of a particular configuration, which is useful for testing a symbolic description. The 'define' button adds the recognizer to the list that the program automatically applies. The 'delete' and 'rename' buttons enable the user to remove and rename the recognizers in the list.

For example, from the description shown in figure 4, the user can delete the unwanted 'right-of; relation, use the 'general' button to relax the two 'concentric' relations to the more general 'contains,' and to relax the boxes to the more general 'shape.' (Figure 5). Finally the user names the recognizer *linked-labeled-boxes* and adds it to the list of named recognizers.



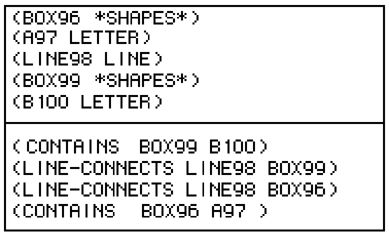
```
(BOX96 BOX)
(A97 LETTER)
(LINE98 LINE)
(BOX99 BOX)
(B100 LETTER)
```

```
(CONCENTRIC BOX99 B100)
(RIGHT-OF BOX99 BOX96)
(LINE-CONNECTS LINE98 BOX99)
(LINE-CONNECTS LINE98 BOX96)
(CONCENTRIC BOX96 A97)
```

```
LINKED-LABELED-BOXES
```

```
TRIANGLE-ABOVE...
GRID-IN-BOX
SHELLS
X-LINES
3-GRID
3-HORIZONTAL-L...
```

Figure 4. 'Search parameters' dialog to view, adjust, and define higher level recognizers.



```
(BOX96 *SHAPES*)
(A97 LETTER)
(LINE98 LINE)
(BOX99 *SHAPES*)
(B100 LETTER)

(CONTAINS  BOX99 B100)
(LINE-CONNECTS LINE98 BOX99)
(LINE-CONNECTS LINE98 BOX96)
(CONTAINS  BOX96 A97 )
```

Figure 5. Adjusted linked-labeled-boxes description

When the new recognizer is applied to a diagram, it identifies all instances of two shapes containing a letter connected by a line and replaces these elements by a new compound element *linked-labeled-boxes* whose parts are the shapes, letters, and line. Then Stretch-A-Sketch will maintain the relations, keeping the letters in the boxes and the boxes linked by the line.

2.4 Rectification considered harmful

Many pen-based systems turn crude sketches into neat drawings, straightening and latching lines, and transforming blobs into geometric shapes. Although rectified drawings make for better presentations, they are exactly wrong for conceptual thinking. They suggest a false commitment, definiteness, and precision, and they deprive the user of opportunities to see alternative interpretations in the rough sketches. Therefore in Stretch-A-Sketch, hand-drawn lines and shapes remain as initially drawn unless and until the user explicitly requests that a particular glyph be rectified. Even then, Stretch-A-Sketch retains original hand-drawn input so that the user can switch back to the hand-drawn form.

3. Constraint based graphic editing

One way to construct a graphic editor for a visual language is to use constraint programming language to describe and enforce the language's spatial syntax. Starting with Sketchpad [13], constraint based graphic editors have been built that maintain spatial relations among drawing elements e.g. [3, 11, 6]. Aligned objects stay aligned and objects stretch, squeeze, and move about to maintain sizes, proportions and spatial relations established by the user. These techniques can be applied to maintain the specific spatial relations of a visual language such as dataflow or logic circuit diagrams.

Generally, constraint based techniques enable the user to program the interactive edit behavior of a design drawing. In computer aided design, a constraint based graphic editor has a distinct advantage over a conventional draw program: a user can state desired relations and the machine ensures that they are maintained. The edit behavior of drawing elements can be programmed to simulate the behavior of real-world objects. For example, in Janus-Constraints, a kitchen design program [8], windows were constrained to occur only in walls; appliances to keep their backs to the wall; and sinks to center under windows.

Constraint techniques have been used in the automatic layout of diagrams for visual languages [5] and to adjust layouts during editing to minimize change [10]. But in most automatic layout schemes constraints are coded into the layout editor for a particular type of diagram; the constraints that Stretch-A-Sketch enforces are specified by its higher level recognizers. Also, most automatic layout editors use structured palette interfaces to create the initial diagram; Stretch-A-Sketch maintains constraints on a hand-drawn diagram.

3.1 Pen interface for constraind graphic editing

Empirical studies bear out the proposition that mouse based structured drawing tools hamper editing in a visual language. Citrin compared mouse-based structured drawing programs with a pen-based gestural recognizer, concluding that pen-gesture based interfaces have distinct advantages for visual language editing [4]. In another study Apte and Kimura showed that drawing graphic diagrams such as Petri nets and flowcharts with a pen is twice as fast as drawing them with a mouse [1].

Constraint based graphics editors typically employ a draw program interface with a tool palette of primitive shapes, operations, and constraints (alignments, etc.). This interface style imposes two barriers between the user and the drawing. First, each element must be selected from a menu, placed, and sized in the drawing. Second, each constraint must be explicitly applied by identifying elements and their spatial relation. The advantage of this interface is that there is no room for ambiguity -- the user is forced to be explicit about the elements and their relations. The disadvantage is that it can be quite tedious.

The pen based interface of Stretch-A-Sketch avoids these barriers at the risk of incorrectly guessing the user's intentions. The user draws directly without selecting from menus, placing, or sizing, and the program's glyph recognizer is responsible for identifying the diagram elements. Whereas with a menu interface, the user must choose to draw either a circle or a rectangle, a pen based interface allows the user to draw ambiguous shapes and identify them later. A shape can be identified as 'either a circle or square,' or it can remain unidentified. Likewise, the user need not explicitly select and apply constraints from a menu; instead Stretch-A-Sketch identifies relevant spatial relations in the diagram.

A known difficulty with identifying constraints from drawings is that in addition to the intended relations, any particular drawing also exhibits other relations that are entirely incidental. For example, a diagram intended to indicate only that one element contains another may happen to represent the elements as circles, misleadingly suggesting that the shape of the objects is also important. One solution is to observe which relations the user does not change during editing and promote them as constraints [9].

Stretch-A-Sketch uses contextual information to identify which constraints in a diagram are relevant. For example, in a floorplan bubble diagram the relevant relations are size, shape, and adjacency of the bubbles; in a graph the relevant relations are connections between nodes. After identifying elements and spatial relations in the user's diagram, the recognizer tries to identify parts of the drawing as instances of previously defined configurations. The configuration recognizers identify which properties and relations are essential and which are incidental, and then Stretch-A-Sketch asserts the essential relations as constraints. (In the event that no configuration is recognized Stretch-A-Sketch asserts all the relations it finds as constraints). In any case, the user can manually adjust the constraints, adding, deleting, and modifying them as desired.

4. Diagrams with behavior

Stretch-A-Sketch is built on top of the Cocktail Napkin recognizer program. The user draws on the Wacom digitizing tablet to make marks in the diagram window. Except for a few gestural commands (e.g. erase, clear screen, overtrace), every mark made with the pen is added to the drawing. The Stretch-A-Sketch extension enables the user to manipulate the hand drawn diagram, adding conventional graphics editing operations to the paper-like interface. The user can select diagram elements and delete, move, resize, and rotate them with the pen. To distinguish these editing commands from drawing new glyphs the user must press a button on the pen barrel. However, Stretch-A-Sketch goes beyond conventional pen-based graphic editors (such as the draw program on the Apple Newton, or InkWare's NoteTaker program) by maintaining spatial relations among the elements of the drawing. For example, in a diagram consisting of connected nodes and arcs, Stretch-A-Sketch maintains the connections as the user adjusts the positions of nodes. This specific edit behavior (maintaining connectivity) is not built in to Stretch-A-Sketch; rather it is obtained from diagram recognizers that can be programmed by users.

4.1 Constraints

Constraints on elements in Stretch-A-Sketch are of two basic types -- constraints on the attributes of individual elements, and constraints on the spatial relations between pairs of elements. Global relations such as "no two nodes may overlap," or "the minimize the distance between nodes" are not supported.

Constraints on element attributes are obtained from the library of glyph templates. Each glyph template describes an allowable range for each property of the element (used for recognition), and these ranges constitute constraints to be maintained during editing. For example, the template provides each element a size constraint based on its bounding box area, a shape constraint based on its aspect ratio, and a constraint that specifies which rotations are legal for the element.

Constraints on spatial relations between elements are obtained from the spatial relations analyzer or from the description of a recognized configuration. For example, when a configuration is recognized as a graph, Stretch-A-Sketch establishes constraints on the diagram that keep the arcs connected to the nodes.

4.2 Propagation of constraint

As Stretch-A-Sketch asserts constraints among diagram elements, each element maintains a list of the relations it is engaged in and the other elements to which it is related. After an element is moved, resized, or reoriented constraint management routines adjust the sizes and positions of related elements to maintain the constraints on the diagram. The routines that propagate constraint among related elements can do it in various ways, resulting in different edit behaviors. For example, if the user tries to move one of the nodes in a graph, several things could happen. The node could snap back to its original position, the arcs connecting it with the rest of the graph could stretch, or the entire graph configuration could move.

Stretch-A-Sketch decides how to propagate constraints using a precedence table of element types. For each pair of element types the table contains an entry describing default editing behavior. For example, the (circle, line) entry of the table indicates that when a circle propagates constraint to a line, the line should be stretched rather than moved. A relation can also be declared a one way constraint. For example, the 'contains' relation can be set to keep the contained element inside the containing one, rather than allowing the containing element to grow or move.

4.3 Modifying constraints

The user can escape the constraint maintenance that automatically enforces existing diagram relations, by holding down a modifier key on the keyboard (the Shift key) before selecting elements to edit. This removes any constraints that apply to the element. This is useful, for example, to cut out a piece of a graph and move it to the side of the diagram for later use.

The user can also selectively add, delete, and modify constraints to the initial set that Stretch-A-Sketch identifies and applies to the diagram. First the user selects a set of elements whose constraints are to be edited, and calls on the analyzer to show the relations presently in effect. Using the 'search parameters' dialog (figure 4), the user can edit them. Stretch-A-Sketch then applies the new relations as constraints on the selected diagram elements.

4.4 A flowchart example

The simple flowchart in Figure 6a was drawn using Stretch-A-Sketch, and recognized as an instance of a flowchart. As the individual glyphs were drawn, the low level recognizer identified them as boxes, a diamond, a circle, lines, and letters. Three higher level recognition rules were then used:

- a "labeled-shape" recognizer identifies any shape containing a letter;
- "poly-line" identifies sets of line segments that connect or tee (as between C, D, and E), and
- "flowchart" identifies labeled shapes that are connected to a line segment or to a poly-line.

The labeled-shape recognizer asserts a containment constraint between the shape and the letter and allows the user to treat the combined element as a unit. The poly-line recognizer asserts connection constraints between its line segments. Finally, the flowchart recognizer asserts the connections between labeled shapes and lines as constraints.

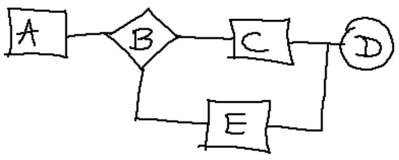


Figure 6a. Flowchart as initially drawn.

In Figure 6b, the user has selected and moved the elements C and D to the right and E to the left, in preparation for adding another element after E in the conditional branch. The line between B and C, and the tee'd poly-line have adjusted by stretching to maintain their connections.

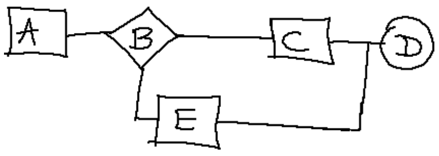


Figure 6b. User moves C & D to right; E to left.

In Figure 6c, the user has (1) deleted the segment of the poly-line leading from E; (2) drawn a new element F; and (3) drawn lines to connect F with E and with the fragment of the previous poly-line. At this point, the poly-line recognizer notices that the new segment can be added to the poly-line, and the flowchart recognizer asserts as constraints the connections between E and F, and between the tee poly-line and F.

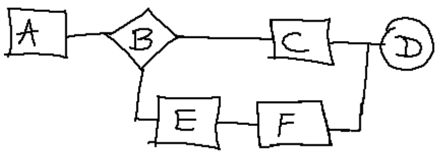


Figure 6c. Element F is inserted between E and D.

Finally, in Figure 6d the diagram is displayed in rectified form.

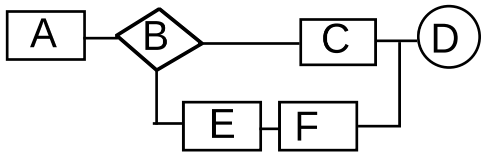


Figure 6d. The flowchart in rectified form.

Note that between Figures 6b and c the user has had to delete and redraw some connections in order to add element F. However, this extra work is small compared to what it would take to perform this same editing task using a normal draw program with a palette interface. To draw each element, the user would have had to choose the appropriate shape from the palette, drag it to its location, and possibly size it; then choose a text tool and enter the label on the keyboard. The shape and the text would need to be grouped together. Then to link the elements the user would choose a line or poly-line tool from the palette and touch both elements being connected. After moving elements the user would have to adjust their connections. To be sure, a palette based graphic editor that is specifically designed for flowchart editing could avoid these problems. Linked elements would be programmed to stay connected; when a shape is chosen, the user would supply a text label, and so forth. But with Stretch-A-Sketch the user can obtain a similar effect by programming the recognizers with the appropriate constraints.

5. Summary

Stretch-A-Sketch combines two techniques from interactive computer graphics—recognition of hand-drawn diagrams and constraint based drawing—to make an environment that supports construction and editing of hand drawn diagrams that behave according to certain rules or constraints. The program identifies elements and spatial relations in the hand-drawn input and enforces and maintains these constraints during editing. When Stretch-A-Sketch can recognize the diagram as an instance of a particular type (e.g. a graph), it maintains only the essential relations in the diagram. When it cannot, all relations in the diagram are made into constraints and the user must adjust them during editing. The editing environment enables the designer to move, resize, and reorient diagram elements, to work with them either in original hand-drawn forms or in ‘cleaned up’ rectified form, and to interactively add, change, and delete constraints.

The current version of Stretch-A-Sketch is a prototype; it has several clear shortcomings. Perhaps most important for visual programming language editing, Stretch-A-Sketch does not handle global spatial constraints, only local ones. In addition, the control of propagation of constraint among the diagram elements is crude and new user-defined element types must be added to the precedence table, which may be inconvenient. The process of modifying constraints using the search parameters dialog is also quite clumsy. It would be easier to edit the constraints graphically, on the diagram itself, rather than resorting to the symbolic description in a separate dialog.

Stretch-A-Sketch suggests a way to add constraint based editing behaviors to hand drawn diagrams that is both cognizant of particular syntactic rules from the domain and forgiving of diagrams that do not conform to the syntax. It makes entering visual language expressions in the early stages of thinking by supporting unstructured input from the pen easier, and it provides a path to more structured and parsable representations needed for further language processing.

Acknowledgments

Discussions with Wayne Citrin and Ellen Do were helpful in developing the ideas behind Stretch-A-Sketch. Trenchant remarks from the anonymous reviewers led to a substantial revision of the paper. Support from the National Science Foundation Grant DMI-9313186 has been instrumental for constructing the constraint programming environment.

References

1. A. Apte, D. Kimura. “A Comparison Study of the Pen and the Mouse in Editing Graphic Diagrams”, 1993 IEEE Symposium on Visual Languages, pp. 352- 357.
2. A. Apte, V. Vo, T.D. Kimura. “Recognizing Multistroke Geometric Shapes: An Experimental Evaluation”, ACM conference on User Interface and Software Technology (UIST) 1993, pp. 121-128.
3. A. Borning., “Programming Language Aspects of ThingLab” ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, 1981, pp. 353-387.
4. W. Citrin. “Requirements for Graphical Front Ends for Visual Languages”, 1993 IEEE Symposium on Visual Languages, pp. 142-149.
5. E. Dengler, M. Friedell, J. Marks. “Constraint-Driven Diagram Layout”, 1993 IEEE Symposium on Visual Languages, pp. 330-335.
6. M.D. Gross. “Graphical Constraints in CoDraw”, 1992 IEEE Workshop on Visual Languages, pp. 81-87.
7. M.D. Gross, “Recognizing and Interpreting Diagrams in Design”, Advanced Visual Interfaces ‘94, Edited by T. Catarci, M.F. Costabile, S. Levialdi, G. Santucci, ACM Press (to appear).
8. M.D. Gross, C. Boyd. “Constraints Provide Domain Behavior in a Construction Kit”, University of Colorado Computer Science Technical Report CU-CS-583-92, 1992.
9. D. Kurlander. “Graphical Editing by Example”, Proc. Human Factors in Computing (InterCHI) 1993, Addison Wesley / ACM Press, pp. 529.
10. M. Minas, G. Viehstaedt. “Specification of Diagram Editors Providing Layout Adjustment with Minimal Change”, 1993 IEEE Symposium on Visual Languages, pp. 324-329.
11. G. Nelson., “Juno — A Constraint-based Graphics System” Computer Graphics, Vol. 19, No. 3, 1985, pp. 235-243.
12. D. Rubine., “Specifying Gestures by Example” Computer Graphics, Vol. 25, No. 4, 1991, pp. 329-337.
13. I. Sutherland. Sketchpad - a Graphical Man-Machine Interface [Ph.D. Dissertation]. M.I.T., 1963.
14. R. Zhao. “Incremental Recognition in Gesture-Based and Syntax-Directed Diagram Editors”,Proc. Human Factors in Computing (InterCHI) 1993, ACM / Addison Wesley, pp. 95-100.