Ambiguous Intentions:
A paper-like interface for creative design

Gross, M.D. and E. Do.

Proceedings ACM Conference on User Interface Software Technology (UIST) '96

Seattle, WA. 183-192

1996

design machine group
University of Washington
Seattle WA USA 98195-5720
http://depts.washington.edu/dmachine

Ambiguous Intentions: a Paper-like Interface for Creative Design

Mark D Gross
College of Architecture and Planning and
Institute of Cognitive Science
University of Colorado at Denver and Boulder
Boulder, CO 80309-0314
mdg@cs.colorado.edu

Ellen Yi-Luen Do
College of Architecture
Georgia Institute of Technology
Atlanta, GA 30332-0155
ellendo@cc.gatech.edu

ABSTRACT

Interfaces for conceptual and creative design should recognize and interpret drawings. They should also capture users' intended ambiguity, vagueness, and imprecision and convey these qualities visually and through interactive behavior. Freehand drawing can provide this information and it is a natural input mode for design. We describe a pen-based interface that acquires information about ambiguity and precision from freehand input, represents it internally, and echoes it to users visually and through constraint based edit behavior.

KEYWORDS: Pen based systems, drawing, design environments, ambiguity and imprecision, graphical techniques.

INTRODUCTION

From mechanical engineering to the graphic arts, designers comprehensively reject the use of computers in the early, conceptual, creative phases of designing. For example, we observed real designers attempting to use two knowledge based design environments: Janus [5] and Archie II [4]. Although they valued each program's potential for providing useful information, designers complained that its stilted interface limited their ability to work and think fluidly. We conjecture (based on observation and interviews) that designers prefer to use paper and pencil because it supports ambiguity, imprecision, and incremental formalization of ideas as well as rapid exploration of alternatives. Paper and pencil is a direct manipulation interface *par excellence*—you draw what you want, where you want it, and how you want it to look. Structured mouse-menu interactions force designers into premature commitment, demand inappropriate precision, and are tedious to use compared with pencil and paper. Yet computers offer advantages: editing, 3D modeling, and rendering, as well as simulation, critiquing, case bases, and distance collaboration. Ideally we would like the best of both worlds.

UIST '96 Seattle Washington USA pp 183-192

An interface for design should capture users' intended ambiguity, vagueness, and imprecision and convey these qualities visually and through interactive behavior. Schön described design as a 'graphical conversation with the materials' [22]. The designer makes a drawing, which stimulates recall of similar forms, visual analogs, or rules and constraints; and the designer reacts in turn by making another drawing. If the drawing is vague or ambiguous, so much the better for stimulating a wider range of recall. Yet most internal CAD representations do not support ambiguity or imprecision; accordingly, neither do their user interfaces. Current design software, which restricts visual representations to precisely drawn geometric elements, stifles this graphical conversation. The CAD drawing eliminates the suggestive power of the sketch. Therefore, good designers defer using computers until they feel ready to be definite and precise. Then, in a virtually irreversible step, they transfer their work to the CAD environment. Typically this happens late in the design process, well after they have made many essential decisions. If designers could begin to design in a freehand drawing environment and move smoothly to more structured and precise representations, they could reap the benefits of computational assistance early without sacrificing the comforts of traditional media.

Although we would like eventually to support both diagramming and sketching, we concentrate here on design diagrams of the sort made on whiteboards and cocktail napkins. As distinct from sketches (though admittedly the distinction is blurry) diagrams are simple, often childish-looking constructions. For example, figure 1 shows a diagram made (on paper) during a discussion about Web page design. Like this one, most diagrams are composed of primitive elements chosen from a small universe of simple symbols—boxes, circles, blobs, lines, arrows. This simplifies the demands on low level recognition; however merely recognizing symbols is not enough. We want the design environment to interpret the drawing as well, to decide, for example, whether a box represents an integrated circuit chip, a desktop computer, a step in a process, or a house. Only if a program can represent the drawing's semantics can it support higher-level design advising. To take recognition further than identifying primitive elements, the program must also identify context.

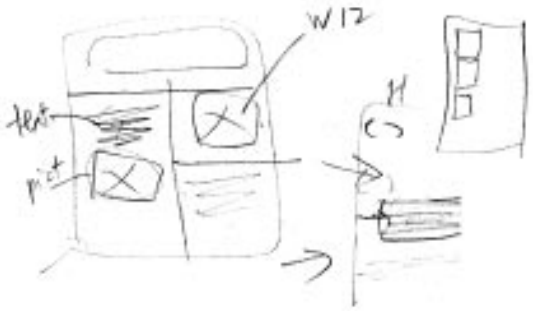

Figure 1. A diagram made during a discussion on Web page design.

In this paper we describe interface techniques used in the Electronic Cocktail Napkin, a pen based collaborative design environment. First we discuss how freehand drawing supports abstraction, ambiguity, and imprecision. Next we review related work on pen based interfaces. Then we describe the Electronic Cocktail Napkin, focusing on its support for abstraction, ambiguity, and imprecision: It supports abstraction through end user programming of graphical rewrite rules. It supports ambiguity by carrying alternative interpretations, using context to resolve unintended ambiguities. It supports imprecision with constraint based interactive behavior. Finally we discuss our formative evaluation approach that includes users in the ongoing interface design, and we conclude with directions for further work.

ABSTRACTION, AMBIGUITY & IMPRECISION

A freehand drawing indicates not only a designer's decisions but also the associated degrees of ambiguity, precision, and commitment. These are crucial characteristics of representations used in conceptual, creative design [6]. Relevant drawing techniques include overtracing and multiple lines, shading and hatching, and outline or formless blob shapes.

Abstraction

A common technique in conceptual design is graphical abstraction, in which a symbol takes the place of a more detailed configuration of parts, enabling a designer to work with components without specifying their internal structure. Graphical abstractions are employed in top-down design, where the designer draws a simple picture, then returns later to fill in detail. Alternatively, in bottom-up design, the designer may recognize a configuration of elements as an instance of a higher-level abstraction, and redraw the design showing a simple picture in place of the detailed configuration.

Ambiguity and Vagueness

An ambiguous representation is another way to postpone commitment yet retain a marker for a later decision. The designer may have several alternative interpretations in mind, but not necessarily a more detailed specification. One technique is to draw a formless or blob-like shape; another is to draw a shape that could be interpreted in one of several ways. Architects often draw a bubble diagram for a floor plan, in which semi-rectangular shapes indicate approximate sizes and positions of rooms. The bubble representation enables a designer to work with sizes and positions without worrying about exact shapes, walls, windows, and doors.

Designers often intentionally make drawings vague or ambiguous. In that case we want to support the ambiguity, and not try to resolve it. However, sometimes drawings are unintentionally ambiguous when read by others without enough contextual information. For example, the designer may draw a perfectly clear arrangement of chairs around a dining room table. Out of context though, a reader might interpret the drawing as houses around a lake. In general we want to resolve ambiguity, but it should not trouble us if we cannot always do it right away.

Imprecision

Imprecision is also valuable. Designers need only rough dimensions to decide on a basic layout. A soft fat pencil or marker ensures that a drawing retains the look and feel of quick, rough thinking and it discourages overly fine-grained decisions. Thus a designer can work with approximations or ranges, confident that more exact calculations may follow.

In summary, freehand drawing supports abstract, ambiguous, and imprecise representations. Abstraction permits postponing detailed specification and allows detailed configurations to be replaced by higher-level elements. Ambiguity permits entertaining several alternatives for the selection or identity of an element. Imprecision permits postponing decisions about exact dimensions and positions. These graphic techniques all serve to reassure the designer that certain decisions remain open—the design is still flexible—while providing explicit visual place holders for issues that remain to be resolved. Therefore an interface for early, conceptual design should provide (1) the means for users to express abstractions, ambiguity, and imprecision; (2) the means for the machine to represent these qualities internally; and (3) the means for the machine to express them in its output and interactive behavior.

RELATED WORK

Discussions of creative design (e.g. [24]) often stress the value and advantages of freehand sketching; yet they rarely mention computational support for these activities. First generation commercial products such as the Newton Message Pad, although promising, make it clear that further work is needed to develop effective pen-based interfaces for drawing. Some of this work is under way. For example, Lakin's work on visual grammars

highlights the need for both formal and informal treatment of drawings [14]. In particular, Lakin observes that a visual parser need not account for the entire drawing, i.e., a drawing may be partly parsable. Moran et al's approach to supporting group whiteboard discussions relies on implicit structures in the domain to parse free form documents, e.g., lists, text, tables, and outlines [17]. Their Tivoli program identifies these structures and provides gestural commands to manipulate them. Ishii and Kobayashi's Clearboard project, like the present work, supports informal graphical conversation with a gestural interface, but it does not recognize or interpret users' input [10]. Saund and Moran's PerSketch program [21] employs computer vision techniques toward many of the same goals we pursue here. Kramer's work on translucent patches [11] explores the uses of translucent trace layers in a freehand interface. Landay and Myers note the value of recognizing and interpreting freehand input in conceptual design, and they support incremental formalization from a sketched diagram to a working prototype [15]. However, their SILK program for the design of user interfaces relies on previously compiled parsing rules specific to interface design, and it does not maintain internal representations for ambiguous or imprecise drawings.

Constraints have a venerable history both in graphical user interfaces [1, 13, 19, 23] and in design [20]. Constraint based techniques used in programming by demonstration and example have been explored, particularly for user interface development [18]. However, the application of these techniques in freehand drawing environments and end-user programmable design environments has not been a focus of this work.

Computational representations of ambiguity and uncertainty used in AI [16] include Bayesian belief networks, fuzzy logic, and certainty weights attached to expert system production rules. However, user interface techniques for controlling these systems are primitive. They typically involve setting numerical values (probabilities) through text-based interfaces. Our work, which represents ambiguity by carrying multiple interpretations and imprecision, with constraints, focuses on obtaining information about imprecision and ambiguity that is implicit in users' input, and providing users with an interface that carries and reflects this information.

ELECTRONIC COCKTAIL NAPKIN

The Electronic Cocktail Napkin [9] is a freehand drawing environment for design, implemented in Macintosh Common Lisp, using Wacom digitizing tablets, Apple Newton PDAs, and the mouse or trackball for input. Its goal is to support the kind of informal drawing that designers do on the back of an envelope or a cocktail napkin during conceptual design. As 'intelligent paper', the Cocktail Napkin aims to support not only making, editing, and managing diagrams, but also using freehand drawings as the central medium for information retrieval, simulation, design critiquing, and collaborative work. We have used it as a prototype interface for other programs: retrieving images from visual databases, providing diagram bookmarks for a case based design aid, and as a front end to interactive simulations. Figure 2a shows the Cocktail Napkin drawing board, with command buttons (left), tabs to select and manipulate tracing layers (right), and previously set aside trace layers (top). Figure 2b shows the program's Sketchbook for storing and retrieving collections of drawings.

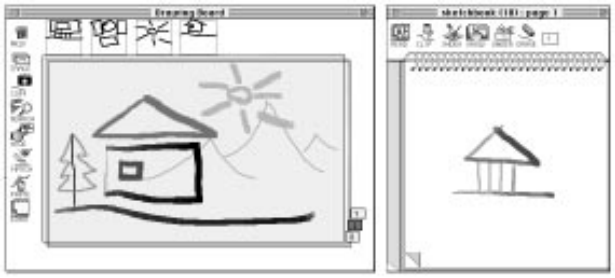

Figure 2. [a] Cocktail Napkin drawing area; [b] Sketchbook for storing interesting sketches.

The user draws on a digitizing tablet or other input device and marks appear in the Drawing Board or Sketchbook window. Each physical input device can be associated with a simulated instrument (pen, pencil, brush) and color, which may respond to varying pressures. For example, harder pressures cause the brush instrument to display a darker thicker line. At first, the program appears to be a simple paint program. But the Cocktail Napkin tries to recognize the glyphs the user draws, and it may (depending on switches) echo this recognition by displaying the name of the glyph (Box, Circle, Line). It may also remain silent as the user continues to draw. Unlike many pen based draw programs, the Cocktail Napkin retains and displays the as-inked representation rather than automatically cleaning up the drawing (although this option is available). The user can edit the diagram as with structured draw programs, selecting, moving, resizing, and rotating drawing elements. The Napkin also supports two or more users collaborating, drawing with two pens on the same tablet, two separate tablets, a tablet and a mouse or Newton, or two Napkins running on connected machines.

Recognizing Configurations

In addition to the primitive marks that make up a drawing at its lowest level, drawings typically have a structure of higher level configurations and a knowledgeable reader can parse primitive elements into these configurations. A configuration is a group of elements and while working a designer may choose to redraw it more abstractly. Alternately, she may just recognize the configuration without redrawing it.

The Napkin provides support for recognizing configurations in a drawing, with user defined patterns and

graphical rewrite rules. For example, a designer working on a floor plan might draw a configuration of four boxes around a larger box, which (an appropriately prepared) Napkin could recognize as an instance of a 'dining table' configuration. In the context of drawing floor plans of rooms, a small box is interpreted as a chair; a larger box as a table. When (again, in the context of rooms) the Napkin finds this spatial arrangement of these elements, it assembles them into a configuration. The user has provided a graphical abstraction (D in a box), so the program replaces the lower-level elements (the table and chairs) with the more abstract picture (figure 3). Later, drawing the abstract symbol can bring in the more detailed configuration of tables and chairs.

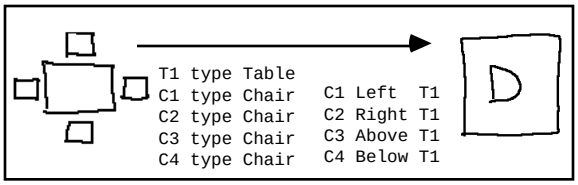

Figure 3. Individual Table and Chairs are replaced by Dining Table symbol.

The Napkin's configuration recognizers operate as dæmons. When the user pauses more than five seconds they run, looking over the drawing for their patterns. The five second delay ensures that the recognizer won't interrupt the user and prematurely parse a partially completed configuration. Together the configuration recognizers make up a set of production rules for parsing drawings.

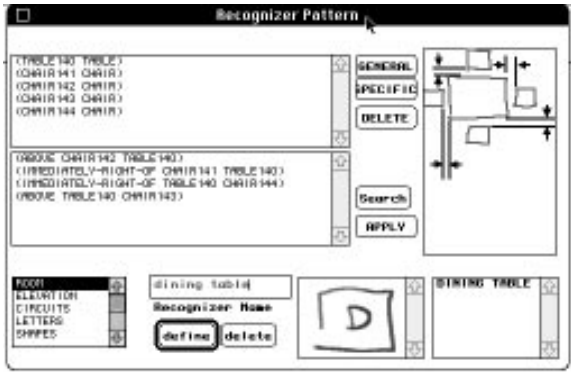

Figure 4. Defining a configuration by example: The Napkin shows relations, the user edits them.

Recognizers are user defined and context dependent. To define a configuration recognizer, the user draws examples. Defining a configuration is a special event, so the Napkin provides a special pattern definition window (figure 4) where it identifies the element types and spatial relations both graphically and as symbolic expressions. These form the pattern the Napkin will use for matching. The user can edit this pattern explicitly, deleting unwanted or incidental relationships and making the type and spatial relation constraints more general or specific, or adjust the pattern implicitly by drawing additional examples. The user names the configuration (Dining Table) and may provide an abstract view to replace the more detailed view of its parts. The user draws this abstract view directly over the parts using a semi-transparent tracing overlay.

Recognizing configurations depends on context. As the selection at the lower left of figure 4 shows, the Dining Table configuration is defined in the Room context. In another context the same configuration of rectangles might have a quite different interpretation. The context may restrict the diagnosis of spatial relations. For example, while in the Room context adjacencies and alignments are reported; however in the Circuits context, the pattern definition window only identifies connections.

Ambiguity and Context

Inevitably users will make marks the Cocktail Napkin cannot identify. The marks may be intentionally vague or ambiguous, or they may not. The program cannot tell whether (1) the user has drawn something intentionally ambiguous, (2) the ambiguity is temporary, about to be resolved, or (3) its own recognition algorithm has failed. Therefore, the program maintains a representation for unknown and ambiguous elements and configurations until it can resolve their identities. It carries alternative interpretations of a drawing element or configuration until it finds additional information elsewhere in the drawing or later in time, or until the user explicitly resolves the ambiguity.

Suppose the user draws a blob-like glyph the Napkin can only recognize as an instance of either 'circle' or 'box' (figure 5a). Rather than trying to decide immediately, the Napkin retains both identifications as alternatives. The ambiguity can be resolved later in one of two ways: The user can identify the element by drawing over it, this time more clearly as a box (figure 5b). When the user overtraces an glyph, the new glyph replaces the old (and if tracing paper is in use, the old one is copied to a trace layer). Or, the program may find that the glyph is part of a known configuration, in a position that resolves the ambiguity (figure 5c).

Figure 5. (a) A blob shape matches both Circle and Box. (b) The user resolves it as a Box by drawing over it, or (c) the Napkin resolves it as a Circle from its position in a configuration.

Figure 5c shows how local context can resolve an unintended ambiguity—the blob must be a Circle because it plays the role of a wheel in a wagon. As noted earlier, context is essential to understanding drawings. If the program knows that a drawing is a flow chart, then a box is a process step, and lines connecting boxes indicate

flow-sequence. If on the other hand, it's a map, then the box is a building and lines are roads.

Figure 6. A curlicue maps to different domain interpretations.

Thus, specific contexts map primitive elements to different interpretations. For example, in a child's drawing a curlicue indicates hair; in a circuit diagram, a coil; in a mechanical diagram, a spring (figure 6). Suppose the Napkin finds a curlicue. Until it finds additional information in the drawing to help choose between the various contexts, the program carries all interpretations and identifies the curlicue (ambiguously) as 'either a spring, a coil, or hair.'

Of course the user can state the context explicitly: "This is an analog circuit" and many ambiguous elements may be resolved. Often, however, the program can identify context when it finds a single unique symbol or configuration. For example, a resistor symbol, found only in the circuits context, identifies a drawing as a circuit diagram. This directs further recognition to search first in the circuits context, then outward in more general contexts. Alternately, the face configuration (two eyes above nose above mouth, centered) could identify the context of the child's drawing. In summary: Context plays a dual role: once the Napkin knows a drawing's context, it can better interpret elements and configurations. On the other hand, a single unique element or configuration can often determine context.

Representing Imprecision with Constraints

The Cocktail Napkin maintains an internal, constraint representation that provides interactive edit behavior. For example, after recognizing a diagram as a graph of nodes and arcs, the Napkin establishes connection constraints. The user can drag the nodes and the graph connectivity remains intact. The constraint representation also allows for imprecision. For example, after recognizing a diagram as a room in floor plan, the Napkin establishes adjacency and alignment constraints on the positions of the rooms, but also approximate constraints on their sizes. Within these size constraints, the user can edit the floor plan; dragging and stretching the rooms. The Napkin can display constraints explicitly as superimposed drawing annotations (as in [7]; see also figure 4), and the user can edit constraints by selecting these annotations.

The user does not apply constraints explicitly; rather, the Napkin identifies spatial relations among drawing elements, such as 'immediately above', 'contains', 'connects', and asserts them as constraints on the drawing. The Napkin filters the spatial relations it finds in the drawing, so not every relation becomes a constraint. Schemes to infer graphical constraints from drawings must determine which relationships are *incidental* (it happens the drawing was made that way) and which are *intended*. One technique is to observe the user's editing moves, and use this dynamic information to identify intended constraints [12]. Instead, we employ a contextual and domain-oriented approach. Only context relevant constraints are asserted; for example, if the user is drawing in the Circuits context, then only connections and not spatial layout constraints will be recognized and asserted. When the Napkin recognizes a configuration, only the relations that belong to its definition are asserted as constraints.

Any scheme to infer constraints from a drawing won't always get it right. It can either tend to over-constrain or under-constrain the drawing. Our scheme errs in the direction of over-constraint—it tends to assert unwanted incidental constraints in the drawing. We think this is likely to provide a better fit to the designers' intentions. It is easier to eliminate unwanted constraints than to determine new constraints and apply them. Of course, after the Napkin has tried to infer the intended constraints on a drawing users can add, delete, and modify them.

Incremental Refinement and Formalization

Ambiguity and imprecision are all very well during early and conceptual design, but ultimately the aim of design is to make definite decisions. Therefore it is important to support incremental refinement and formalization of designs, from the ambiguous imprecise early sketch to the well-defined, highly articulated hard line drawing. At some point the designer will move from sketchy diagrams of conceptual design to more rigid drawings of schematic design (figure 7).

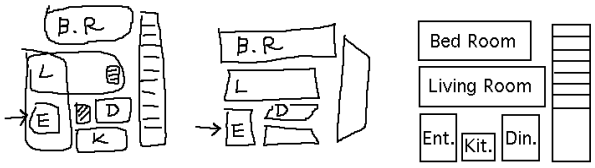

Figure 7. Design drawings undergo a process of gradual refinement.

Although the Napkin does not support this entire process smoothly, the representations we use can be employed. For example, the user can gradually tighten constraints by making more specific decisions, and switch from as-drawn to rectified views of drawing elements.

IMPLEMENTATION

Three main mechanisms support the features of the Cocktail Napkin described in the preceding section: (1) the

low level recognition of hand drawn glyphs, (2) the higher level recognition of configurations, and (3) the maintenance of constraints on and spatial relations among diagram elements. These three mechanisms are employed with respect to a simple representation for context. The implementation of these mechanisms and their interactions with contexts are described below.

Low Level Recognition

The low level recognizer's job is to identify the best match(es) in a given context for an input glyph along with a certainty value for the match (1-5 where 1 is most certain). Recognition of each glyph is typically performed immediately after input, but it may also be postponed for later batch processing. A glyph is time-delimited by when the user lifts the pen for longer than a certain time out period. Therefore a glyph can include more than one stroke made in rapid succession. Raw data from the digitizing tablet is stored as a sequence of x,y points, pressures, and pen-down marks. In the first processing step, the glyph's bounding box is identified, its aspect ratio and size classified into a set of 'fuzzy' categories (e.g., tall, square, wide; tiny, normal, gigantic), and the point coordinates rectified to a 3x3 grid inscribed within the bounding box. The rectified coordinates are coded to a sequence of numbers (1-9) that indicate the grid squares of the pen path. The sequence only discerns crude shape (for example, [1 2 3 6 9 8 7 4] describes both a clockwise circle and a box); therefore a corner count helps to distinguish shapes with the same pen path. Corners are identified when the change in heading of segments between raw points exceeds 45 degrees (see figure 8).

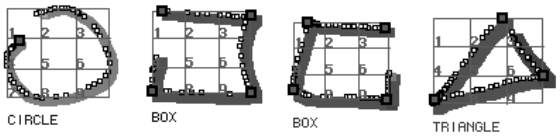

Figure 8. Features of simple glyphs: the 3x3 path grid, raw x, y points, and corners.

These features—pen path, aspect ratio and size of the bounding box, stroke and corner count—describe a glyph. After analyzing the input glyph, the recognizer compares these features against a library of previously stored glyph templates, each of which records a range or list of allowable values. Where an individual glyph records a single value (e.g., a corner count of 4), the glyph template records a constraint on acceptable values (e.g., a set of possible corner counts {3 4 5}). Templates also contain a slot that indicates which transformations of a glyph are permitted; for example, unlike boxes and circles, alphanumerics cannot be drawn in reflection.

The match proceeds as follows: First, any templates that match the input glyph's pen path sequence are selected as candidates. (A hash table of templates keyed on pen paths makes this first comparison fast). If any templates match the input glyph's pen path, their other features are compared; candidates that match all features are selected. If this stage of matching finds only one candidate it is assigned a certainty value of 1 (most certain). If several candidates matched the input glyph's pen path as well as all its other features, the set of successful candidates is assigned a certainty value of 2 (pretty good match).

Next, the input glyph's pen path is permuted, to determine whether the input glyph matches a template drawn backwards, rotated, or reflected. For example, the square-circle path [1 2 3 6 9 8 7 4] is permuted to [3 2 1 4 7 8 9 6], which describes a circle drawn counterclockwise from top right (see figure 8a, b, and c). Any matching permutations are filtered for other features as above. Successful candidates are then checked for whether they permit rotation and reflection. Candidates found at this stage of matching are also assigned a certainty value of 1 and 2.

Straight lines and dots are treated as special cases. Any very small glyph is a dot; straight lines are identified by examining the deviation of raw points from the segment connecting the first and last points. Lines and dots identified in this way are assigned a certainty value of 1.

If the pen path or its permutations cannot be found, the recognizer attempts a weaker match. First, candidates with otherwise matching features whose pen path differs by only one or two squares are sought. A single match is assigned a certainty value of 3 (okay match); a multiple match, a certainty value of 4 (weak match). Finally, if no templates can be found whose pen path is close to the input glyph's, any templates with otherwise matching features are assigned a certainty value of 5 (very weak match).

Training the low level recognizer The user can introduce new templates on the fly, by drawing examples of the glyph to be trained and entering its name. The Napkin constructs a new template for the glyph, adds the features of the training samples to the corresponding slots of the new template, and adds the pen path to the hash key for the new template. If a template already exists, features from the sample are added to slots of the existing template. For example, a user may add a 4-stroke Box to a training set containing only 1-stroke Boxes. The template's strokes slot would be expanded from the list [1] to the list [1 4]. Thus, constructing and refining training sets may occur at any time during use. Features from the new samples are used to adjust the low level recognition constraints.

Recognizing Configurations

A configuration is a set of elements (glyphs or configurations) of certain types arranged in certain spatial relations. To identify configurations, the program runs recognizer functions over all pairs of appropriately typed elements in the diagram to look for these patterns. A small number of hand coded binary predicates (e.g., overlaps, contains, connects, right-of, above) support this

recognition. When a recognizer finds an instance of the pattern it is looking for, it replaces the individual elements by a configuration whose parts are the individuals. For example, a recognizer for 'labeled graph node' configurations looks for patterns of a Letter contained by a Circle. When it finds this pattern, it replaces the letter and the circle by a new 'labeled graph node' configuration. The new configuration may then be parsed as an element in a larger 'graph' configuration.

The configuration recognizers can resolve (coerce) the types of previously ambiguous glyphs. After low level matching, some glyphs may still not be uniquely identified; the matching routines may have only been able to narrow them down to a list of candidates. Then, these ambiguous glyphs may be identified by the role they play in a configuration. For example, a glyph initially identified as either Box or Letter D may be resolved by the labeled-graph-node recognizer as the Letter D, if it occurs inside a circle.

Constructing configuration recognizers. As with the glyph recognizer, the user trains the configuration recognizer by showing examples, from which the Napkin extracts constraints on the types of elements and their spatial relations. Each recognizer is a Lisp function, which the program builds in response to the designer's examples and subsequent adjustments, then compiles for rapid execution. For example, given a circle containing the letter A as an example of a labeled-graph-node, the Napkin builds this recognizer function:

```
(defun labeled-graph-node (circle1 letter2)
        (and    (glyph-type circle1 'circle)
                (glyph-type letter2 'letter)
                (contains circle1 letter2)))
```

Examples are combined disjunctively. For example, if the user draws an additional example (say, a box containing a letter), a similar predicate is constructed, and the two predicates are combined with an OR.

```
(defun labeled-graph-node (glyph1 letter2)
        (or     (and    (glyph-type glyph1 'circle)
                        (glyph-type letter2 'letter)
                        (contains glyph1 letter2))
                (and    (glyph-type glyph1 'box)
                        (glyph-type letter2 'letter)
                        (contains glyph1 letter2))))
```

The Napkin also provides a more explicit way to adjust the configuration recognizers, the pattern definition interface shown above in figure 4 (with the Dining Table example). Here, the designer can view each clause of the recognizer predicate, and explicitly adjust the types and spatial relations in each clause. The interface shows the example for each clause, with its elements and spatial relations displayed both graphically and textually. The user can select an element or relation in either the text or the graphic view.

The General, Specific, and Delete buttons operate on the selected element or spatial relation. If the example inadvertently includes extraneous elements or undesired incidental spatial relations, the user can delete these from the recognizer predicate. The General and Specific buttons allow the user to adjust the element type and spatial relations parts of the predicate. For example, performing the General operation on the type constraint (Circle1 Circle) converts the description to (Glyph1 Shape). Conversely, performing the Specific operation on (Glyph1 Shape) allows the user to choose a specific type from the various Shape glyphs.

The user can also adjust the relations descriptors. Suppose in the example for labeled-graph-node that the letter happens to have been drawn concentric with the circle. The spatial relation will then read (Concentric Circle1 Letter2). The General operation will change this to (Contains Circle1 Letter2), and the Specific operation would change it back to Concentric. Spatial relations belong to classes; for example 'topology' constraints include line connections and containments; 'spatial layout' constraints include the above, below, right and left-of relations. Relations are also organized by the types of objects they apply to. For example, some relations only apply between lines; others only to shapes. Transitivity and commutativity properties are recorded for each relation, and these are used to ensure a non redundant relations diagnosis. Finally, the relations are ordered by degree of specificity. For example, Concentric is a more specific version of Contains which in turn is more specific than Overlaps. When identifying spatial relations among drawing elements, the Napkin chooses the most specific version that applies.

Contexts

The Napkin maintains a list of all known contexts, a 'current context' and a 'current context chain.' Recognizers look first in the current context for glyph templates and configurations. The current context chain specifies a sequence of other contexts that may contribute to recognition, most specific contexts listed first. This is the search path for contextual recognition. The Napkin sets and resets the current context and the context chain as it identifies glyphs and configurations that belong to a specific domain.

A context has four components: glyph templates, configuration recognizers, spatial relations, and mappings of glyphs from other contexts. For example, glyph templates in the Circuits context include resistors and capacitors. The recognizers in this context include series and parallel resistor configurations. The spatial relations are limited to forms of connection, because only connections are relevant in circuit diagrams. Finally, the mappings indicate that Line glyphs (from the general Lines context) should be interpreted as Wire glyphs. Mapping enables a glyph to be interpreted differently in different contexts, without duplicating the template or the recognition effort. In the general context a line is just a

line. In the Circuits context it is interpreted as a Wire and in the Floor plans context it is interpreted as a Wall.

Contextual recognition. All recognition—of both simple glyphs and configurations— takes place with respect to context. The first recognition attempt is in the current context, but all contexts in the current context chain are consulted. Only glyphs unique to a context are stored there, so for example, the circuit context stores only glyphs for electronic components. Other glyphs that may appear in a circuit diagram (but also in other kinds of diagrams), for example shapes and letters, are found in more general contexts further along the context chain.

The matching procedure for simple glyphs described above is performed against the template libraries in each of the active contexts in the order prescribed by the current context chain. After each match-in-context attempt, mappings from the current context are applied (e.g., changing Line to Wire). This results in a list of matches and associated certainty values for each context; only the most certain matches are retained. If as often happens there is only one match, it is returned. If there are several matches they are returned as a list of alternatives, in the order their contexts appear in the context chain.

Recognizing context. The Napkin's initial context is the most general one, which includes only shapes, alphanumeric characters, and lines; these are basic elements of all diagrams. The initial current context chain contains only this one context. As soon as the user draws a glyph or makes a configuration that the Napkin can identify as belonging to a more specific context, the Napkin adjusts the current context and the current context chain. For example, if the Napkin identifies a Resistor, the current context is set to Circuits, and the current context chain becomes (Circuits General). The current context can be changed by recognizing a glyph or configuration that is unique to a context. Once set, the context will not change until the window is cleared, or until the user begins to draw glyphs and configurations that indicate a new domain.

Constraints

The features slot values in the glyph template establish unary constraints on the properties of each glyph. For example, a Bed may be restricted to certain size (small) and aspect ratio (tall or wide) constraints. The Napkin maintains these constraints, and permits users to size the Bed only within this size and aspect ratio range. If the user tries to directly resize the bed beyond its constrained limits, the bed does not accept the new size, but snaps back to its previous dimension.

Each binary spatial relation that the Napkin can diagnose also contains code to implement one-way constraints for the relation. For example, the Immediately-Right-Of constraint contains code to adjust the position of the right edge of the left element (if the right element is moved), or the position of the left edge of the right element (if the left element is moved). Each relation keeps track of the most recent direction of propagation. Each element contains a 'stretchy-p' flag that tells the constraint propagator whether its size or position is to be adjusted in response to changes.

The Napkin's constraint routines are limited to value propagation, though more sophisticated methods could obviously be added. It handles conflict resolution in two ways. First, recently changed values are more fixed than older values, though the user can anchor any value. Second, the classes of elements and their attributes can be ordered. For example, properties of *shapes* can be set to take precedence over those of *lines*, or *dimensions* are more fixed than *positions*.

DISCUSSION

Evaluation with Users

Usability studies of the Archie case based design aid [3] and informal observation of users of Janus and other CAD environments were an initial impetus for the Cocktail Napkin. It became clear that for designers, drawing is a non-negotiable demand. We also conducted several pilot studies of architects' drawing conventions, to explore the feasibility of machine understanding of freehand drawings. In one study [8], fifty undergraduate students produced diagrams from slides of buildings and drawings. In a second study [8], twenty architects and design instructors produced diagrams of famous buildings from memory. In a third study [2] sixty-two design students were asked to draw diagrams to illustrate design problems and responses and to write textual explanations of diagrams. These studies encouraged us to pursue recognition and interpretation of freehand drawing.

Throughout development of the Cocktail Napkin we engage undergraduate students and design instructors at the University of Colorado's architecture program to test the interface. A training period of at least half an hour is needed for a new evaluator to become comfortable using a digitizing tablet while looking at a monitor. Typically we ask users to carry out a sequence of simple tasks and observe them, noting difficulties they encounter; individual evaluation sessions have ranged in duration from 15 minutes to 2 hours. We also ask evaluators for an open-ended response to their experience and often we ask specific questions about pieces of the interface we are testing. Suggestions from users have strengthened the interface details, as well as provided some novel ideas. (For example, one user's comments led us to implement a selection decay option, where selected items gradually deselect if not operated on). This formative evaluation technique enables us to focus user testing on specific aspects of the program we are currently working on, and we can learn a great deal by observing only a small number of users.

Most of our experience has been using the program with a digitizing tablet, rather than a LCD digitizing display, which we would prefer. LCD digitizing hardware for Macintosh hardware has been scarce and expensive. (The MCL programming environment commits us to the Mac.) However, separation of drawing surface and display imposes an annoying extra hand-eye coordination penalty on users. That was the initial attraction of the Newton; although low resolution, it provided an input-output drawing device. Until LCD digitizing displays become readily available, we are exploring alternatives, such as using touch screen overlays to the Powerbook LCD screen.

Many users sit down at the Cocktail Napkin and begin to draw, ignoring its recognition facilities. Their primary aim is to make a drawing. For these users, low level interface details are exceedingly important: for example, response time from putting the pen down to making a mark must be minimized; in earlier versions users accidentally selected previous marks by touching them quickly; the time-out delay after drawing a glyph caused problems, and even the feel of the pen and tablet are important concerns. Observing these users has resulted in a great deal of low level tuning and tweaking.

Designers often ask "why does it tell me I drew a box?" Designers know what they draw, so echoing recognition seems gratuitous. This suggests setting the Cocktail Napkin with low level echoing and rectification turned off, so as not to distract the user. However, most designers have understood and appreciated the need for end-user training of symbols and contextual definitions of configurations.

DISCUSSION & FUTURE WORK

We have chosen to explore a general purpose drawing environment that end users can make more specific. Rather more effort than we would like is still needed to construct the domain specific recognizers. However, we favor this approach over building custom applications because we believe there is inevitably a high degree of idiosyncrasy, even within the drawing conventions of a specific design domain.

Diagrams and Sketches

In future work, we would like to support true sketching, in addition to the more symbolic diagram-making described here. Sketching includes more overtracing, non symbolic figures that explore shape, hatching, cross hatching, and shape filling gestures. First steps will be to recognize these graphical gestures and to filter sketches for identifiable shapes and eliminate noise. For example, the architectural sketch in figure 9a can be simplified to the diagram in figure 9b, by substituting simple figures for their overtraced equivalents. Or, the sketch plan in figure 10a can be filtered to identify the key shapes in figure 10b. We have begun to explore these ideas but are still at a very early stage of implementing them.

Figure 9. Sketch for a museum and amphitheater, architect Ping Xu

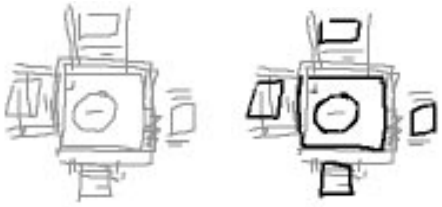

Figure 10. Sketch plan of Palladio's Villa Capra.

Commitment and Certainty

Ambiguity, vagueness, and imprecision: all these correlate in some way with the users' certainty or commitment to decisions. We believe a drawing also indicates the designer's degree of certainty and commitment, but we have not tried to assess this. We intend to start by looking at the drawing pressure and speed of the input data, which so far we have ignored, and overtracing. The Napkin echoes pen pressure visually but it does not interpret this information. Likewise, because we record pen strokes as a series of points in time, we can determine pen speed. We can also detect overtracing, in which the user repeatedly draws the same or a similar form. We expect we can extract information about the user's degree of certainty and commitment, as well as intended precision from drawing speed and pressure and overtracing. For example, a rapid light drawing might indicate a rough but sure set of decisions, while a slow heavy drawing might indicate an uncertain attempt to be precise. To explore this territory, we plan to carry out empirical studies of pressure and speed in drawing, for which we propose to use the Cocktail Napkin as a recording instrument.

Drawing as a Front End to Everything

We have described our work with drawing as a design medium, but we would like to expand the range of application for freehand drawing input. We believe it is a natural medium for human users, who first learn to hold a crayon in kindergarten. While most computer interfaces today are text-based, we wonder how far we can push pen based, freehand additions. We expect the greatest gains will be in augmenting text based interfaces with optional, alternate modes of interaction based on drawing.

ACKNOWLEDGMENTS

We wish to acknowledge the interest and patience of students and instructors of Environmental Design who participated in evaluating the Cocktail Napkin. The

anonymous reviewers provided valuable suggestions for improving the paper, which we hope we have succeeded in following. NSF grant DMII 93-13186 provided essential material assistance.

REFERENCES

1. A. Borning, R. Duisberg., "Constraint-based tools for building user interfaces" ACM Transactions on Graphics, Vol. 5, No. 4, 1986, pp. 345-374.
2. E.Y.-L. Do, "What's in a Diagram (that a computer should understand)", Computer Aided Architectural Design Futures '95, Edited by M. Tan, Singapore, 1995, pp. 469-480.
3. E.Y.-L. Do, S.-W.D. Or, D.M. Carson, C.. Chang, W.C. Hacker. "*Usability Study of A Case-based Design Aid Archie*", Georgia Tech, 1994.
4. E.A. Domeshek, J.L. Kolodner, C.M. Zimring, "The Design of A Tool Kit for Case-based Design Aids", *Artificial Intelligence in Design '94*, Edited by J. Gero, Kluwer Academic Publishers, Dordrecht, 1994.
5. G. Fischer, A. Girgensohn. "End-User Modifiability in Design Environments", CHI'90), ACM Press, pp. 183-191.
6. V. Goel, Sketches of Thought, MIT Press, Cambridge MA, 1995.
7. M.D. Gross, "Graphical Constraints in CoDraw", IEEE Workshop on Visual Languages, Edited by S. Tanimoto, IEEE Press, Seattle, 1992, pp. 81-87.
8. M.D. Gross, "Indexing visual databases of designs with diagrams", Visual Databases in Architecture, Edited by A. Koutamanis, H. Timmermans, I. Vermeulen, Avebury, Aldershot, UK, 1995, pp. 1-14.
9. 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, 1994, pp. 89-94.
10. H. Ishii, M. Kobayashi, "Clearboard: A seamless medium for shared drawing and conversation with eye contact", CHI '91, Monterrey, CA, 1991, pp. 525-532.
11. A. Kramer, "Translucent Patches - dissolving windows", ACM Symposium on User Interface Software and Technology, ACM Press, Marina del Rey, CA, 1994, pp. 121-130.
12. D. Kurlander. "Graphical Editing by Example", Human Factors in Computing (InterCHI), Addison Wesley / ACM Press, pp. 529.
13. D. Kurlander, S. Feiner. "Interactive Constraint Based Search and Replace", CHI '92, ACM Press, pp. 609-618.
14. F. Lakin, J. Wambaugh, L. Leifer, D. Cannon, C. Steward., "The electronic notebook: performing medium and processing medium" Visual Computer, Vol. 5, No. 1989, pp. 214-226.
15. J.A. Landay, B.A. Myers, "Interactive Sketching for the Early Stages of Interface Design", CHI '95 ACM Press, Denver, Colorado, 1995, pp. 43-50.
16. R.L.d. Mantaras, D. Poole, ed. Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence. San Mateo, CA: Morgan Kaufmann, 1994.
17. T.P. Moran, P. Chiu, W.v. Melle, G. Kurtenbach, "Implicit Structures for Pen-Based Systems within a Freeform Interaction Paradigm", CHI '95, ACM Press, Denver, Colorado, 1995, pp. 487-494.
18. B. Myers, Creating User Interfaces by Demonstration, Academic Press, Boston, 1988.
19. G. Nelson., "Juno — A Constraint-based Graphics System" Computer Graphics, Vol. 19, No. 3, 1985, pp. 235-243.
20. M. Sapossnek. " Research on Constraint-Based Design Systems.", Proc. 4th Intl. Conf. Applications of AI in Engineering, Cambridge, England.
21. E. Saund, T.P. Moran. "A Perceptually-Supported Sketch Editor", ACM Symposium on User Interface Software and Technology, ACM Press, pp. 175-184.
22. D. Schön., "Designing as Reflective Conversation with the Materials of a Design Situation" Knowledge Based Systems, Vol. 5, No. 3, 1992.
23. I. Sutherland. Sketchpad - a Graphical Man-Machine Interface [Ph.D. Dissertation]. M.I.T., 1963.
24. M. Tscheligi, S. Houde, R. Kolli, A. Marcus, M. Muller, K. Mullet. "Creative Prototyping Tools: What Interaction Designers Really Need to Produce Advanced Interaction Concepts", CHI '95 ACM Press, pp. 170-171.