Sketch Based Interfaces: Early Processing for Sketch Understanding

Tevfik Metin Sezgin
MIT AI Laboratory
Massachusetts Institute of Technology
Cambridge MA 02139, USA
mtsezgin@ai.mit.edu

Thomas Stahovich
CMU Deptarment of Mechanical Engineering
Pittsburgh, PA 15213
stahov@andrew.cmu.edu

Randall Davis
MIT AI Laboratory
Massachusetts Institute of Technology
Cambridge MA 02139, USA
davis@ai.mit.edu

ABSTRACT

Freehand sketching is a natural and crucial part of everyday human interaction, yet is almost totally unsupported by current user interfaces. We are working to combine the flexibility and ease of use of paper and pencil with the processing power of a computer, to produce a user interface for design that feels as natural as paper, yet is considerably smarter. One of the most basic steps in accomplishing this is converting the original digitized pen strokes in a sketch into the intended geometric objects. In this paper we describe an implemented system that combines multiple sources of knowledge to provide robust early processing for freehand sketching.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User interfaces; H.5.2 [**User Interfaces**]: Input Devices and strategies; J.6 [**Computer Aided Engineering**]: CAD

General Terms

Design, Human Factors

Keywords

freehand sketching, natural interaction, multiple sources of knowledge

1. INTRODUCTION

Freehand sketching is a familiar, efficient, and natural way of expressing certain kinds of ideas, particularly in the early phases of design. Yet this archetypal behavior is largely unsupported by user interfaces in general and by design software in particular, which has for the most part aimed at providing services in the later phases of design. As a result designers either forgo tool use at the early stage or end up having to sacrifice the utility of freehand sketching for the capabilities provided by the tools. When they move to a computer for detailed design, designers usually leave the sketch behind and the effort put into defining the rough geometry on paper is largely lost.

We are working to provide a system where users can sketch naturally and have the sketches understood. By "understood" we mean that sketches can be used to convey to the system the same sorts of information about structure and behavior as they communicate to a human engineer.

Such a system would allow users to interact with the computer without having to deal with icons, menus and tool selection, and would exploit direct manipulation (e.g., specifying curves by sketching them directly, rather than by specifying end points and control points). We also want users to be able to draw in an unrestricted fashion. It should, for example, be possible to draw a rectangle clockwise or counterclockwise, or with multiple strokes. Even more generally, the system, like people, should respond to how an object looks (e.g., like a rectangle), not how it was drawn. This will, we believe, produce a sketching interface that feels much more natural, unlike Graffiti and other gesture-based systems (e.g., [9], [14]), where pre-specified motions (e.g., an L-shaped stroke or a clockwise rectangular stroke) are required to specify a rectangular shape.

The work reported here is part of our larger effort aimed at providing natural interaction with software, and with design tools in particular. That larger effort seeks to enable user to interact with automated tools in much the same manner as they interact with each other: by informal, messy sketches, verbal descriptions, and gestures. Our overall system uses a blackboard-style architecture [6], combining multiple sources of knowledge to produce a hierarchy of successively more abstract interpretations of a sketch.

Our focus in this paper is on the very first step in the sketch understanding part of that larger undertaking: interpreting the pixels produced by the user's strokes and producing low level geometric descriptions such as lines, ovals, rectangles, arbitrary polylines, curves and their combinations. Conversion from pixels to geometric objects is the first step in interpreting the input sketch. It provides a more compact representation and sets the stage for further, more abstract interpretation (e.g., interpreting a jagged line as a symbol for a spring).

2. THE SKETCH UNDERSTANDING TASK

Sketch understanding overlaps in significant ways with the extensive body of work on document image analysis generally (e.g., [2]) and graphics recognition in particular (e.g., [16]), where the task is to go from a scanned image of, say, an engineering drawing, to a symbolic description of that drawing.

Differences arise because sketching is a realtime, interactive process, and we want to deal with freehand sketches, not the precise diagrams found in engineering drawings. As a result we are not analyzing careful, finished drawings, but are instead attempting to respond in real time to noisy, incomplete sketches. The noise is different as well: noise in a freehand sketch is typically not the small-magnitude randomly distributed variation common in scanned documents. There is also an additional source of very useful information in an interactive sketch: as we show below, the timing of pen motions can be very informative.

Sketch understanding is a difficult task in general as suggested by reports in previous systems (e.g., [9]) of a recognition rate of 63%, even for a sharply restricted domain where the objects to be recognized are limited to rectangles, circles, lines, and squiggly lines (used to indicate text).

Our domain–mechanical engineering design–presents the additional difficulty that there is no fixed set of shapes to be recognized. While there are a number of traditional symbols with somewhat predictable geometries (e.g., symbols for springs, pin joints, etc.), the system must also be able to deal with bodies of arbitrary shape that include both straight lines and curves. As consequence, accurate early processing of the basic geometry–finding corners, fitting both lines and curves–becomes particularly important.

3. SYSTEM DESCRIPTION

Sketches can be created in our system using any of a variety of devices that provide the experience of freehand drawing while capturing pen movement. We have used traditional digitizing tablets, a Wacom tablet that has an LCD-display drawing surface (so the drawing appears under the stylus), and a Mimio whiteboard system. In each case the pen motions appear to the system as mouse movements, with position sampled at rates between 30 and 150 points/sec, depending on the device and software in use.

In the description below, by a single stroke we mean the set of points produced by the drawing implement between the time it contacts the surface (mouse-down) and the time it breaks contact (mouse-up). This single path may be composed of multiple connected straight and curved segments (see, Fig. 1).

Our approach to early processing consists of three phases *approximation*, *beautification*, and *basic recognition*. Approximation fits the most basic geometric primitives–lines and curves–to a given set of pixels. The overall goal is to approximate the stroke with a more compact and abstract description, while both minimizing error and avoiding overfitting. Beautification modifies the output of the approximation layer, primarily to make it visually more appealing without changing its meaning, and secondarily to aid the third phase, basic recognition. Basic recognition produces interpretations of the strokes, as for example, interpreting a sequence of four lines as a rectangle or square. (Subsequent recognition, at the level of mechanical components, such as springs, and pin joints is accomplished by another of our systems [1]).

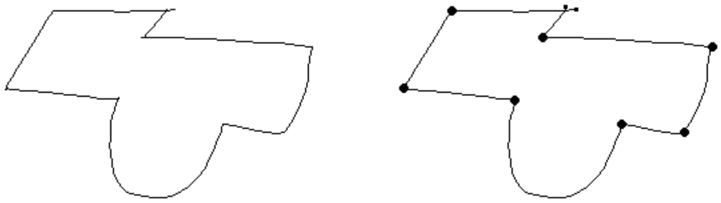


Figure 1: The stroke on the left contains both curves and straight line segments. The points we want to detect in the vertex detection phase are indicated with large dots in the figure on the right. The beginning and the end points of the stroke are indicated with smaller dots.

3.1 Stroke Approximation

Stroke processing consists of detecting vertices at the endpoints of linear segments of the stroke, then detecting and characterizing curved segments of the stroke.

3.1.1 *Vertex detection*

We use the sketch in Fig. 1 as a motivating example of what should be done in the vertex detection phase. Points marked in Fig. 1 indicate the corners of the stroke, where the local curvature is high.

Note that the vertices are marked only at what we would intuitively call the corners of the stroke (i.e., endpoints of linear segments). There are, by design, no vertices marked on curved portions of the stroke because we want to handle these separately, modeling them with curves (as described below). This is unlike the well studied problem of piecewise linear approximation [13].

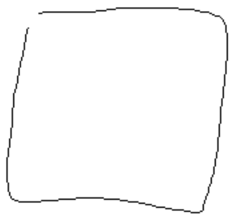


Figure 2: Stroke representing a square.

Our approach takes advantage of the interactive nature of sketching, combining information from both stroke direction and speed data. Consider as an example the square in Fig. 2; Fig. 3 shows the direction, curvature (change in direction with respect to arc length) and speed data for this stroke. We locate vertices by looking for points along the stroke that are minima of speed (the pen slows at corners) or maxima of the absolute value of curvature.[1]

While extrema in curvature and speed typically correspond to vertices, we cannot rely on them blindly because noise in the data introduces many false positives. To deal with this we use average based filtering.

[1]From here on for ease of description we use curvature to mean the absolute value of the curvature data.

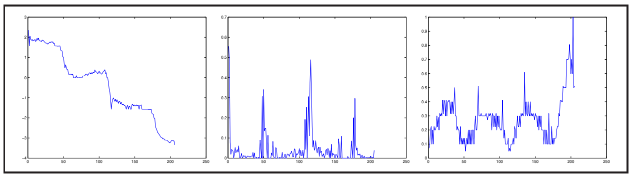


Figure 3: Direction, curvature and speed graphs for the stroke in Fig. 2

Average based filtering
We want to find extrema corresponding to vertices while avoiding those due to noise. To increase our chances at doing this, we look for extrema in those portions of the curvature and speed data that lie beyond a threshold. Intuitively, we are looking for maxima of curvature only where the curvature is already high and minima of speed only where the speed is already low. This will help to avoid selecting false positives of the sort that would occur say, when there is a brief slowdown in an otherwise fast section of a straight stroke.

To avoid the problems posed by choosing a fixed threshold, we set the threshold based on the mean of each data set.[2] We use these thresholds to separate the data into regions where it is above/below the threshold and select the global extrema in each region that lies above the threshold.

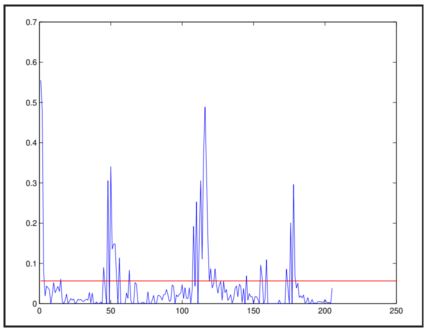


Figure 4: Curvature graph for the square in Fig. 2 with the threshold dividing it into regions.

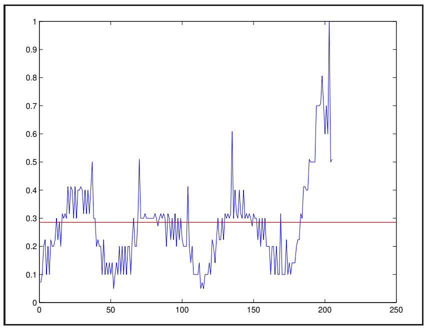


Figure 5: Speed graph for the stroke in Fig. 2 with the threshold dividing it into regions.

[2]The exact threshold has been determined empirically; for curvature data the threshold is the mean, while for the speed the threshold is 90% of the mean.

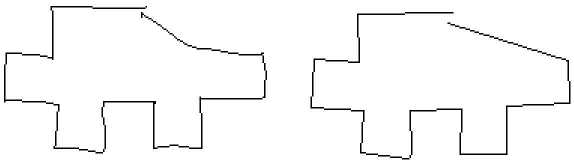


Figure 6: At left the original sketch of a piece of metal; at right the fit generated using only curvature data.

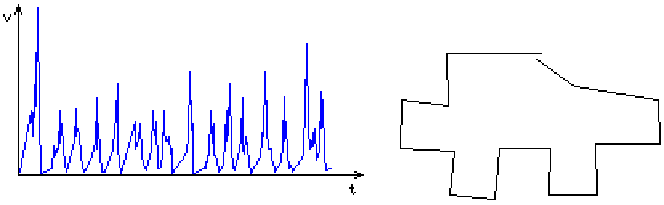


Figure 7: At left the speed graph for the piece; at right the fit based on only speed data.

Application to curvature data

Fig. 4 shows the curvature graph partitioned into regions of high and low curvature. Note that this reduces but doesn't eliminate the problem of false positives introduced by noise in the stroke. We deal with the false positives using the hybrid fit generation scheme described below.[3]

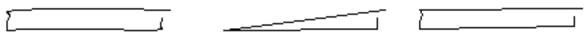
While average based filtering performs better than simply comparing the curvature data against a hard coded threshold, it is still clearly not free of empirical constants. As we explain when considering future work, scale space provides a better approach for dealing with noisy data without having to make a priori assumptions about the scale of relevant features.

Application to speed change
Our experience is that curvature data alone rarely provides sufficient reliability. Noise is one problem, but variety in angle changes is another. Fig. 6 illustrates how curvature fit alone misses a vertex (at the upper right) because the curvature around that point was too small to be detected in the context of the other, larger curvatures. We solve this problem by incorporating speed data into our decision as an independent source of guidance.

Just as we did for the curvature data, we reduce the number of false extrema by average based filtering, then look for speed minima. The intuition here is simply that pen speed drops when going around a corner in the sketch. Fig. 7 shows (at left) the speed data for the sketch in Fig. 6, along with the polygon drawn from the speed-detected vertices (at right).

[3]An alternative approach is to detect consecutive almost-collinear edges (using some empirical threshold for collinearity) and combine them into one edge, removing the vertex in between. Our hybrid fit scheme deals with the problem without the need to decide what value to use for "almost-collinear."

Using speed data alone has its shortcomings as well. Polylines formed from a combination of very short and long line segments can be problematic: the maximum speed reached along the short line segments may not be high enough to indicate the pen has started traversing another edge, with the result that the entire short segment is interpreted as the corner. This problem arises frequently when drawing thin rectangles, common in mechanical devices. Fig. 8 illustrates this phenomena. In this figure, the speed fit misses the upper left corner of the rectangle because the pen failed to gain enough speed between the endpoints of the short vertical segment. The curvature fit, by contrast, detects all corners, along with some other vertices that are artifacts due to hand dynamics during freehand sketching. This illustrates the utility of having both fits available.



(a) Input, 63 points

(b) Using speed data, 4 vertices

(c) Using curvature data, 7 vertices

Figure 8: Average based filtering using speed data misses a vertex. The curvature fit detects the missed point (along with vertices corresponding to the artifact along the left edge of the rectangle).

We use information from both sources, generating hybrid fits by combining the set of candidate vertices derived from curvature data F_d with the candidate set from speed data F_s, taking into account the system's certainty that each candidate is a real vertex.

Generating hybrid fits

Hybrid fit generation occurs in three stages: computing vertex certainties, generating a set of hybrid fits, and selecting the best fit.

Our certainty metric for a curvature candidate vertex v_i is the scaled magnitude of the curvature in a local neighborhood around the point, computed as $|d_{i-k} - d_{i+k}|/l$. Here l is the curve length between points S_{i-k}, S_{i+k} and k is a small integer defining the neighborhood size around v_i. The certainty metric for a speed fit candidate vertex v_i is a measure of the pen slowdown at the point, $1 - v_i/v_{max}$, where v_{max} is the maximum pen speed in the stroke. The certainty values are normalized to $[0, 1]$.

While both of these metrics are designed to produce values in $[0, 1]$, they have different scales. As the metrics are used only for ordering within each set, they need not be numerically comparable across sets. Candidate vertices are sorted by certainty within each fit.

The initial hybrid fit H_0 is the intersection of F_d and F_s. A succession of additional fits is then generated by appending to H_i the highest scoring curvature and speed candidates not already in H_i.

To do this, on each cycle we create two new fits: $H_i' = H_i + v_s$ (i.e., H_i augmented with the best remaining speed fit candidate) and $H_i'' = H_i + v_d$ (i.e., H_i augmented with the best remaining curvature candidate). We use least squares error as a metric of the goodness of a fit: the error ε_i is computed as the average of the sum of the squares of the distances to the fit from each point in the stroke S:

$$\varepsilon_i = \frac{1}{|S|} \sum_{s \in S} ODSQ(s, H_i)$$

Here $ODSQ$ stands for *orthogonal distance squared*, i.e., the square of the distance from the stroke point to the relevant line segment of the polyline defined by H_i. We compute the error for H_i' and for H_i''; the higher scoring of these two (i.e., the one with smaller least squares error) becomes H_{i+1}, the next fit in the succession. This process continues until all points in the speed and curvature fits have been used. The result is a set of hybrid fits.

In selecting the best of the hybrid fits the problem is as usual trading off more vertices in the fit against lower error. Here our approach is simple: We set an error upper bound and designate as our final fit H_f, the H_i with the fewest vertices that also has an error below the threshold.

3.1.2 Handling curves

The approach described thus far yields a good approximation to strokes that consists solely of line segments, but as noted our input may include curves as well, hence we require a means of detecting and approximating them.

The polyline approximation H_f generated in the process described above provides a natural foundation for detecting areas of curvature: we compare the Euclidean distance l_1 between each pair of consecutive vertices in H_f to the accumulated arc length l_2 between those vertices in the input S. The ratio l_2/l_1 is very close to 1 in the linear regions of S, and significantly higher than 1 in curved regions.

We approximate curved regions with Bézier curves, defined by two end points and two control points. Let $u = S_i$, $v = S_j$, $i < j$ be the end points of the part of S to be approximated with a curve. We compute the control points as:

$$c_1 = k\hat{t}_1 + v$$

$$c_2 = k\hat{t}_2 + u$$

$$k = \frac{1}{3} \sum_{i \le k < j} |S_k - S_{k+1}|$$

where $\hat{t}_1$ and $\hat{t}_2$ are the unit length tangent vectors pointing inwards at the curve segment to be approximated. The 1/3 factor in k controls how much we scale $\hat{t}_1$ and $\hat{t}_2$ in order to reach the control points; the summation is simply the length of the chord between S_i and S_j.[4]

As in fitting polylines, we want to use least squares to evaluate the goodness of a fit, but computing orthogonal distances from each S_i in the input stroke to the Bézier curve segments would require solving a fifth degree polynomial. (Bézier curves are described by third degree polynomials, hence computing the minimum distance from an arbitrary point to the curve involves minimizing a sixth degree polynomial, equivalent to solving a fifth degree polynomial.) A numerical solution is both computationally expensive and heavily dependent on the goodness of the initial guesses for

[4]The 1/3 constant was determined empirically, but works very well for freehand sketches. As we discovered subsequently, the same constant was independently chosen in [15].

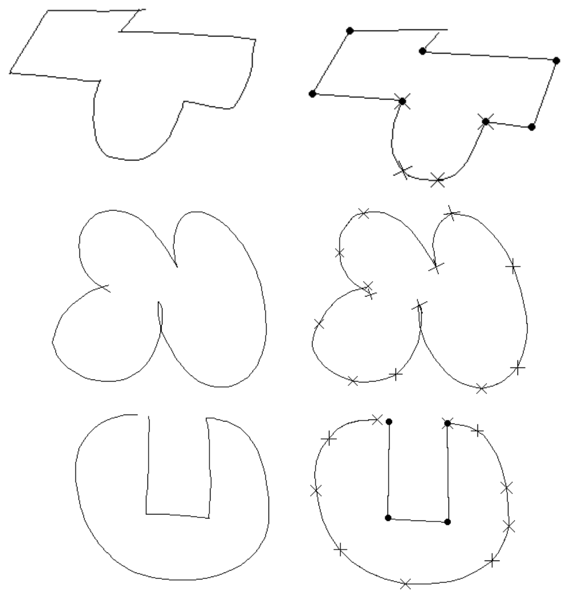


Figure 9: Examples of arbitrary stroke approximation. Boundaries of Bézier curves are indicated with crosses, detected vertices are indicated with dots.

roots [12], hence we resort to an approximation. We discretize the Bézier curve using a piecewise linear curve and compute the error for that curve. This error computation is $O(n)$ because we impose a finite upper bound on the number of segments used in the piecewise approximation.

If the error for the Bézier approximation is higher than our maximum error tolerance, the curve is recursively subdivided in the middle, where middle is defined as the data point in the original stroke whose index is midway between the indices of the two endpoints of the original Bézier curve. New control points are computed for each half of the curve, and the process continues until the desired precision is achieved.

Examples of the capability of our approach is shown in Fig. 9, a hastily-sketched mixture of lines and curves. Note that all of the curved segments have been modeled curves, rather than the piecewise linear approximations that have been widely used previously.

3.2 Beautification

Beautification refers to the (currently minor) adjustments made to the approximation layer's output, primarily to make it look as intended. We adjust the slopes of the line segments in order to ensure the lines that were apparently meant to have the same slope end up being parallel. This is accomplished by looking for clusters of slopes in the final fit produced by the approximation phase, using a simple sliding-window histogram. Each line in a detected cluster is then rotated around its midpoint to make its slope be the weighted average of the slopes in that cluster. The (new) endpoints of these line segments are determined by the intersections of each consecutive pair of lines. This process (like any neatening of the drawing) may result in vertices being moved; we chose to rotate the edges about their midpoints because this produces vertex locations that are close to those detected, have small least square errors when measured against the original sketch, and look right to the user. Fig. 10 shows the original stroke for the metal piece we had before, and the output of the beautifier. Some examples of beautification are also present in Fig. 13.

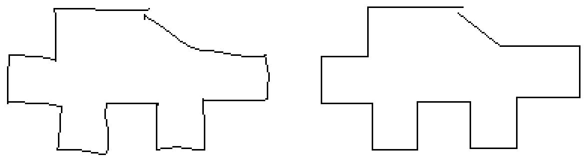


Figure 10: At left the original sketch of a piece of metal revisited, and the final beautified output at right.

3.3 Basic Object Recognition

The final step in our processing is recognition of the most basic objects that can be built from the line segments and curve segments produced thus far, i.e., simple geometric objects (ovals, circles, rectangles, squares).

Recognition of these objects is done with hand-tailored templates that examine various simple properties. A rectangle, for example, is recognized as a polyline with 4 segments all of whose vertices are within a specified distance of the center of the figure's bounding box; a stroke will be recognized as an oval if it has a small least squares error when compared to an oval whose axes are given by the bounding box of the stroke.

3.4 Evaluation

We have conducted a user study to measure the degree to which the system is perceived as easy to use, natural and efficient. Study participants were asked to create a set of shapes using our system and Xfig, a Unix tool for creating diagrams. Xfig is a useful point of comparison because it is representative of the kinds of tools that are available for drawing diagrams using explicit indication of shape (i.e., the user indicates explicitly which parts of the sketch are supposed to be straight lines, which curves, etc.) As in other such tools, XFig has a menu and toolbar interface; the user selects a tool (e.g., for drawing polygons), then creates the shapes piece by piece.

Thirteen subjects participated in our study, including computer science graduate students, computer programmers and an architecture student. Subjects were given sufficient time to get familiar with each system and then asked to draw a set of 10 shapes (examples given in Fig 11). All of the subjects reported our system being easier to use, efficient and more natural feeling. The subjects were also asked which system they would prefer when drawing these sort of informal shapes on a computer. All but one subject preferred our system; the sole dissenter preferred a tablet surface that had the texture and feel of paper.

Overall users praised our system because it let them draw shapes containing curves and lines directly and without having to switch back and forth between tools. We have also observed that with our system, users found it much easier to draw shapes corresponding to the gestures they routinely draw freehand, such as a star.

While the central point of this comparison was to deter-

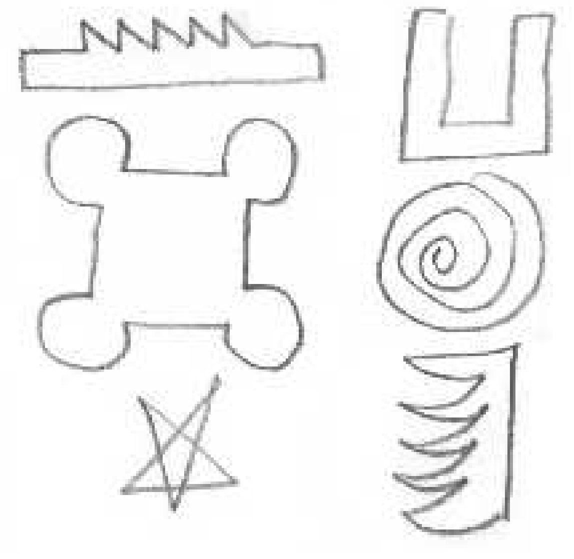


Figure 11: Examples of the shapes used in the user study.

mine how natural it felt to use each system, we also evaluated our system's ability to produce a correct interpretation of each shape (i.e., interpret strokes appropriately as lines or curves). Overall the system's identification of the vertices and approximation of the shapes with lines and curves was correct 96% of the time on the ten figures.

In addition to the user studies we have conducted, we wrote a higher level recognizer for evaluation purposes. The higher level recognizer takes the geometric descriptions generated by the basic object recognition module of our system and combines them into domain specific objects.

Fig. 13 shows the original input and the program's analysis for a variety of simple but realistic mechanical devices drawn as freehand sketches. The last two of them are different sketches for a part of the direction reversing mechanism for a tape player. Recognized domain specific components include gears (indicated by a circle with a cross), springs (indicated by wavy lines), and the standard fixed-frame symbol (a collection of short parallel lines). Components that are recognized are replaced with standard icons scaled to fit the sketch.

An informal comparison of the raw sketch and the system's approximations shows whether the system has selected vertices where they were drawn, fit lines and curves accurately, and successfully recognized basic geometric objects. While informal, this is an appropriate evaluation because the program's goal is to produce an analysis of the strokes that "looks like" what was sketched.

We have also begun to deal with overtracing, one of the (many) things that distinguishes freehand sketches from careful diagrams. Fig. 12 illustrates one example of the limited ability we have thus far embodied in the program.

4. RELATED WORK

In general, systems supporting freehand sketching lack one or more of the properties that we believe a sketching system should have:

Figure 12: An overtraced oval and a line along with and the system's output.

- It should be possible to draw arbitrary shapes with a single stroke, (i.e., without requiring the user to draw objects in pieces).
- The system should do automatic feature point detection. The user should not have to specify vertex positions by hand.
- The system should not have sketching modes for drawing different geometric object classes (i.e., modes for drawing circles, polylines, curves etc.).
- The sketching system should feel natural to the user.

The Phoenix sketching system [15] had some of the same motivation as our work, but a more limited focus on interactive curve specification. While the system provided some support for vertex detection, its focus on curves led it to use Gaussian filters to smooth the data. While effective for curves, Gaussians tend to treat vertices as noise to be reduced, obscuring vertex location. As a result the user was often required to specify the vertices manually.

Work in [5] describes a system for sketching with constraints that supports geometric recognition for simple strokes (as well as a constraint maintenance system and extrusion for generating solid geometries). The set of primitives is more limited than ours: each stroke is interpreted as a line, arc or as a Bézier curve. More complex shapes can be formed by combinations of these primitives, but only if the user lifts the pen at the end of each primitive stroke, reducing the feeling of natural sketching.

The work in [3] describes a system for generating realtime spline curves from interactively sketched data. They focus on using knot removal techniques to approximate strokes known to be composed only of curves, and do not handle single strokes that contain both lines and curves. They do not support corner detection, instead requiring the user to specify corners and discontinuities by lifting the mouse button, or equivalently by lifting the pen. We believe our approach of automatically detecting the feature points provides a more natural and convenient sketching interface.

Zeleznik [7] describes a mode-based stroke approximation system that uses simple rules for detecting the drawing mode. The user has to draw objects in pieces, reducing the sense of natural sketching. Switching modes is done by pressing modifier buttons in the pen or in the keyboard. In this system, a click of the mouse followed by immediate dragging signals that the user is drawing a line. A click followed by a pause and then dragging of the mouse tells the system to enter the freehand curve mode. Our system allows drawing arbitrary shapes without any restriction on how the user draws them. There is enough information provided by the freehand drawing to differentiate geometric shapes such as curves, polylines, circles and lines from one another, so we believe requiring the user to draw things in a particular fashion is unnecessary and reduces the natural feeling of sketching. Our goal is to make computers understand what the user is doing rather than requiring the user to sketch in a way that the computer can understand.

Among the large body of work on beautification, Igarashi et al. [8] describes a system combining beautification with constraint satisfaction, focusing on exploiting features such as parallelism, perpendicularity, congruence and symmetry. The system infers geometric constraints by comparing the input stroke with previous ones. Because sketches are inherently ambiguous, their system generates multiple interpretations corresponding to different ways of beautifying the input, and the most plausible interpretation is chosen among these interpretations. The system is interactive, requiring the user to do the selection, and doesn't support curves. It is, nevertheless, more effective then ours at beautification, but beautification is not the main focus of our work and is present for the purposes of completeness.

The works in [15] and [3] describe methods for generating very accurate approximations to strokes known to be curves with precision several orders of magnitude below the pixel resolution. The Bézier approximations we generate are less precise but are sufficient for approximating free-hand curves. We believe techniques in [15] and [3] are excessively precise for free-hand curves, and the real challenge is detecting curved regions in a stroke rather than approximating those regions down to the numerical machine precision.

5. FUTURE WORK

We are working to link this early processing to other work in our group that has focused on recognition [1] of higher level mechanical objects. This will provide the opportunity to add model-based processing of the stroke, in which early operations like vertex localization may be usefully guided by knowledge of the current best recognition hypothesis.

In addition, incorporating ideas from scale space theory looks like a promising way of detecting different scales inherent in the data and avoiding *a priori* judgments about the size of relevant features. In the pattern recognition community [4], [11] and [10] apply some of the ideas from scale space theory to similar problems. We are currently working on ways of applying these techniques to speed and curvature data. We believe this may allow us to deal more effectively with sketches that contain relevant details at a variety of scales. There is no guaranteed way of deciding which scales are important at the geometric level, so using constraints and/or information provided by the domain of application may help in scale selection.

Humans naturally seem to slow down when they draw things carefully as opposed to casually, so another interesting research direction would be to explore the degree to which one can use the time it takes to draw a stroke as an indication of how careful and precise the user meant to be.

6. CONCLUSION

We have built a system capable of using multiple sources of information to produce good approximations of freehand sketches. Users can sketch on an input device as if drawing on paper and have the computer detect the low level geometry, enabling a more natural interaction with the computer, as a first step toward more natural user interfaces generally, and toward earlier use of automated tools in the design cycle in particular.

7. REFERENCES

[1] C. Alvarado. A natural sketching environment: Bringing the computer into early stages of mechanical design. Master's thesis, Massachusetts Institute of Technology, 2000.

[2] H. S. Baird, H. Bunke, and K. Yamamoto. Structured document image analysis. Springer-Verlag, Heidelberg, 1992.

[3] M. Banks and E. Cohen. Realtime spline curves from interactively sketched data. In *SIGGRAPH, Symposium on 3D Graphics*, pages 99–107, 1990.

[4] A. Bentsson and J. Eklundh. Shape representation by multiscale contour approximation. *IEEE PAMI 13, p. 85–93, 1992.*, 1992.

[5] L. Eggli. Sketching with constraints. Master's thesis, University of Utah, 1994.

[6] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12:213–253, 1980. Reprinted in: *Readings in Artificial Intelligence*, Bonnie L. Webber and Nils J. Nilssen (eds.)(1981), pp 349-389. Morgan Kaufman Pub. Inc., Los Altos, CA.

[7] A. Forsberg, K. Herndon, and R. Zeleznik. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH'96*, pages 163–170, 1996.

[8] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Interactive beautification: A technique for rapid geometric design. In *UIST '97*, pages 105–114, 1997.

[9] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer, vol. 34, no. 3, March 2001, pp. 56-64.*

[10] T. Lindeberg. Edge detection and ridge detection with automatic scale selection. *ISRN KTH/NA/P–96/06–SE, 1996.*, 1996.

[11] A. Rattarangsi and R. T. Chin. Scale-based detection of corners of planar curves. *IEEE Transactionsos Pattern Analysis and Machine Intelligence*, 14(4):430–339, Apr. 1992.

[12] N. Redding. Implicit polynomials, orthogonal distance regression, and closest point on a curve. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 191–199, 2000.

[13] R. Rosin. Techniques for assessing polygonal approximations of curves. *7th British Machine Vision Conf., Edinburgh*, 1996.

[14] D. Rubine. Specifying gestures by example. *Computer Graphics*, 25(4):329–337, 1991.

[15] P. Schneider. Phoenix: An interactive curve design system based on the automatic fitting of hand-sketched curves. Master's thesis, University of Washington, 1988.

[16] K. Tombre. Analysis of engineering drawings. In *GREC 2nd international workshop*, pages 257–264, 1997.

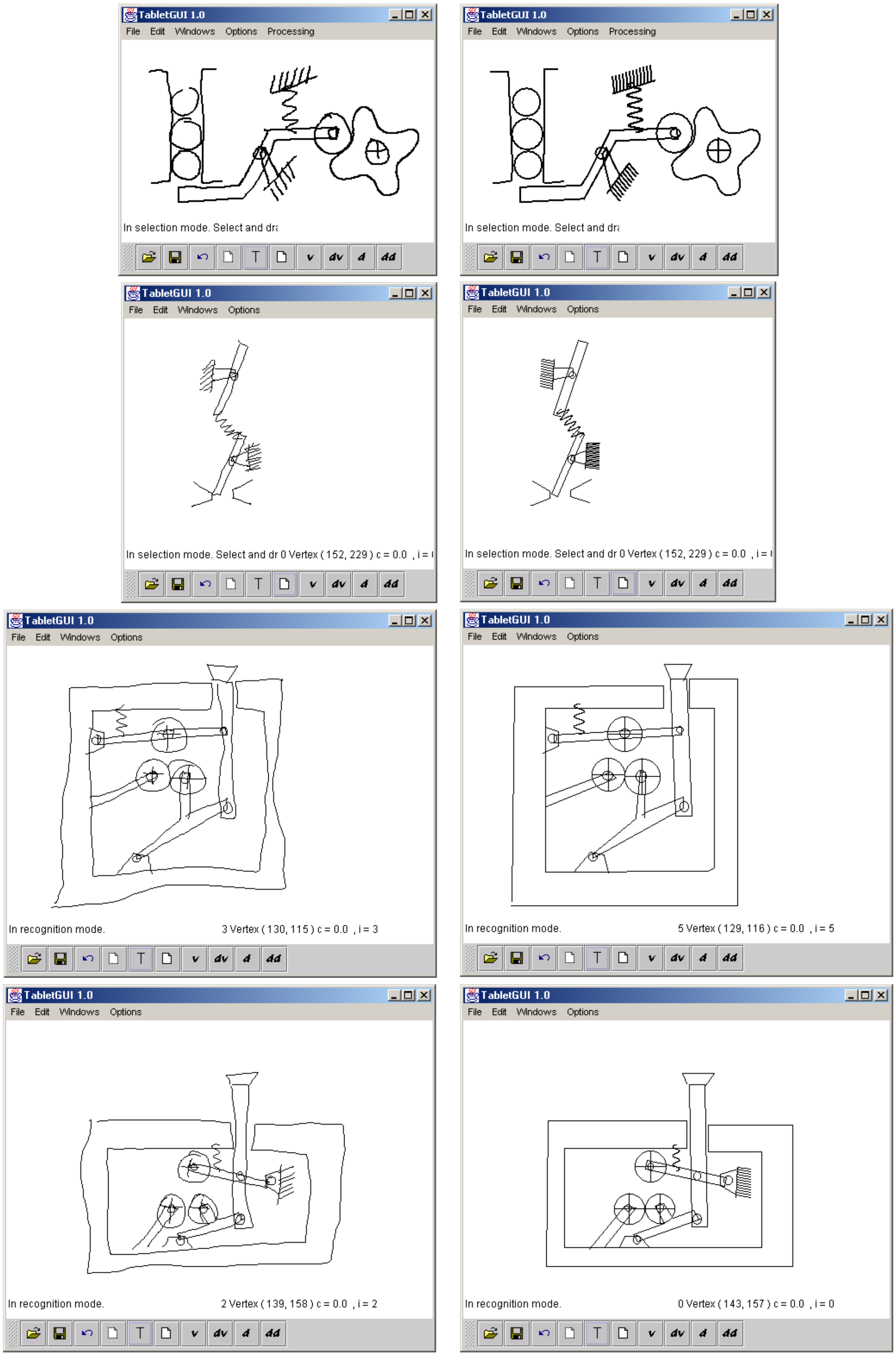


Figure 13: Performance examples: The first two pair are sketches of a marble dispenser mechanism and a toggle switch. The last two are sketches of the direction reversing mechanism in a tape player.