Early Processing in Support of Sketch Understanding*

Tevfik Metin Sezgin+, Thomas Stahovich#, Randall Davis+
+MIT Artificial Intelligence Laboratory; # CMU Deptarment of Mechanical Engineering

Massachusetts Institute of Technology,
Cambridge MA 02139, USA,
`mtsezgin@ai.mit.edu`,
`stahov@andrew.cmu.edu`,
`davis@ai.mit.edu`,
Home page: `http://www.ai.mit.edu/people/mtsezgin/sezgin.html`
Home page: `http://www.me.cmu.edu/faculty1/stahovich/stahovich.html`
Home page: `http://www.ai.mit.edu/people/davis/davis.htm`

Abstract. Freehand sketching is a natural and crucial part of everyday human interaction, especially important in design, yet is unsupported by current design automation software. 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 design environment 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 the 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.

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 design behavior is largely unsupported by design software, which has for the most part aimed at providing services in the later phases of the design process. As a result designers either forgo tool use at this 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 want users to be able to draw in an unrestricted fashion, unlike

---

* In Submission to UBICOMP 2001

Graffiti. 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 is unlike Graffiti and other gesture-based systems such as [10], and [15] where constrained pen motions like an L-shaped stroke, or a rectangular stroke drawn in a particular fashion is used to indicate a rectangle. This will, we believe, produce a sketching interface that feels much more natural.

The work reported here is part of our larger effort aimed at providing natural interaction with design tools. That larger effort seeks to enable designers to interact with automated tools in much the same manner as they interact with each other: by informal, messy sketches, verbal descriptions, and gestures. That system uses a blackboard-style architecture [7], 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 to produce low level geometric descriptions such as lines, ovals, rectangles, arbitrary polylines, curves and their combinations. Converting 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., [17]), 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 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) [10].

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 we illustrate below. 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 over-fitting. 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]).

3.1 Stroke Approximation

Stroke processing starts by looking for vertices, i.e., points where there is a noticeable change in orientation. We use the example in Fig. 1 below as a motivating example of what should be done in vertex the detection phase.

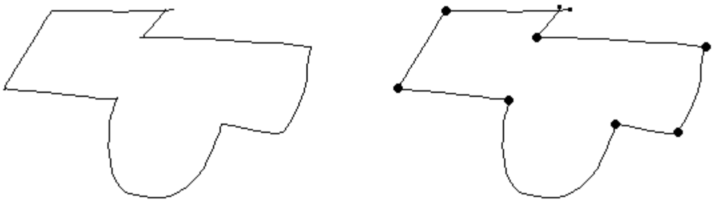


Fig. 1. The stroke on the left contains both curves and straight line segments. The points we want to detect in the vertex detection phase are indicated with large dots in the figure on the right. The beginning and the end points are indicated with smaller dots.

Finding Vertices Vertex localization is a frequent subject in the extensive literature on graphics recognition (e.g., [14] compares 21 methods). These methods are not applicable for our purposes because they produce a simple piecewise linear fit (i.e., a polygonalization) of the stroke. The algorithm in [5], for example, while fast and accurate for piecewise linear strokes, would produce a piecewise linear fit to all of Fig. 1, including the curved section. We want instead to fit line segments only to the straight sections of the sketch and fit curves where the stroke is curved.

We accomplish this by taking advantage of the interactive nature of sketching, combining information from both direction change and speed data.

Direction change

The first step in detecting vertices based on the direction change is computing the direction change. Point to point direction change is far too noisy to be informative (even in clean, scanned engineering drawings). Freehand sketches are more difficult still, because of the greater noise, the relative paucity of data points, and because, unlike scanned drawings, data points may be some distance from one another.[1] The relative paucity of data points in sketches as compared to scanned images (which may have upto thousands of points per inch) means that traditional noise filtering techniques as in [5] are less effective.

Given the noisy data, some form of smoothing has long been used, but the relative paucity of data points makes well known filtering techniques considerably less useful. Our approach is to compute the direction at a point by fitting an orthogonal distance regression (ODR) line to a small window of points centered on the point in question. (Orthogonal distance regression finds a line that minimizes the sum of the orthogonal distances from the points to the line, unlike linear regression, which minimizes only the y-distance.) For computational efficiency we use a discrete approximation to the ODR that is good to 0.5 degree.[2] Given the direction along the curve, direction change is simply the point-to-point difference in direction.

Despite the smoothing done by using a window of points, both direction and direction change data are noisy (see Fig. 2). We want to select as candidate vertices extrema points of the direction change curve, but in doing so want to avoid local extrema, while (of course) finding more than just the single global extreme. To accomplish this we select only the extrema of the function above a threshold. To avoid the problems posed by choosing a fixed threshold, we compute the threshold from the direction change data itself, selecting only those extrema that are greater than twice the average direction

---

[1] The pen typically travels numerous pixels between samplings, because while digitizing tablets have sub-millimeter accuracy of pen placement, they are typically not sampled fast enough to provide a data point every time the pen moves from one pixel to the next in a freehand sketch.

[2] Principal component analysis solves the same problem: The direction at a point is given by the eigenvector corresponding to the largest eigenvalue of the covariance matrix for the window of points surrounding the point in question. But this is computationally more expensive than our ODR approximation, which is more than accurate enough for our purposes. There are also gradient descent methods for ODR, but these don't provide any significant computational improvement.

change.[3] Intuitively, the average-based thresholding process partitions the stroke into regions of high and low change in direction.

Fig. 2, with its relatively careful hand sketch of a square, shows that in this case the peaks (here the minima) of the direction change function are good indicators of corners.

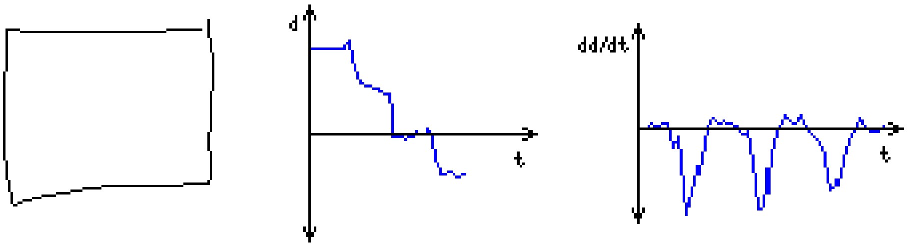


Fig. 2. A hand-drawn square; graphs of direction (d) and direction change (dd/dt).

Speed data

Our experience is that direction change data alone rarely provide sufficient reliability. Noise is one problem, but variety in angle changes is another. Fig. 3 illustrates how direction fit misses a vertex (at the upper right) because the direction change there was too small to be detected in the context of the other, larger direction changes. We solve this problem by incorporating the speed data into our decision as an independent source of guidance.

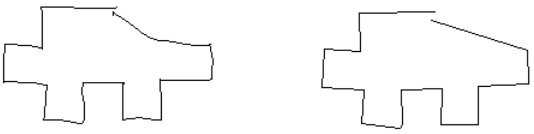


Fig. 3. At left the original sketch of a piece of metal; at right the fit generated using only direction change data.

We measure instantaneous pen speed by measuring the distance pen travels per unit time. Then we look for speed minima. The intuition here is simply that pen speed drops when going around a corner in the sketch. Fig. 4 shows (at left) the speed data for the sketch in Fig. 3, along with the polygon drawn from the speed-detected vertices (at right).

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

---

[3] This self scaling frees us to some extent from built-in threshold, but as we point out below, the scale space theory seems to provide a better methodology for choosing thresholds.

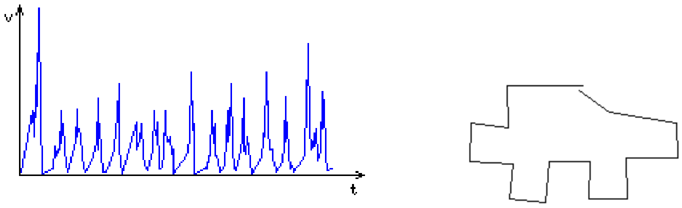


Fig. 4. At left the speed graph for the piece; at right the fit based on only speed data.

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.

Our solution is to use information from both sources, and generating hybrid fits by combining the candidate set from direction change F_d with the candidate set from speed information 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 direction change candidate vertex v_i is the scaled magnitude of the direction change in a local neighborhood around it expressed by $|d_{i-k} - d_{i+k}|/d_{max}$. Here d_{max} is the largest direction change anywhere in the vertices of the approximation 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 anywhere in the vertices of the approximation.

As is traditional both of these metrics produce values in [0,1], though with different scales (as the metrics are used only for ordering within each set, they need not be numerically comparable). 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 are then generated by appending to H_i the highest scoring direction change 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 direction change 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 (ie., 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 direction change 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.

Handling curves The approach described 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. This is done by comparing the Euclidean distance l_1 between each pair of consecutive vertices u, v in H_f to the accumulated arc length l_2 between the corresponding 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:

$$c_1 = k\hat{t_1} + v$$

$$c_2 = k\hat{t_2} + u$$

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

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 roots [13], 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.

---

[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 [16].

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.

One example of the capability of our approach is shown in Fig. 5, a hastily-sketched mixture of lines and curves.

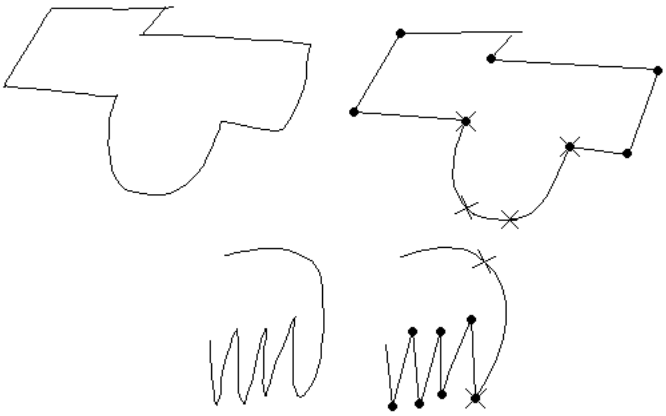


Fig. 5. Two examples of arbitrary stroke approximation. Boundaries of Bézier curves are indicated with crosses and detected vertices are indicated with dots.

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. 6 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. 8.

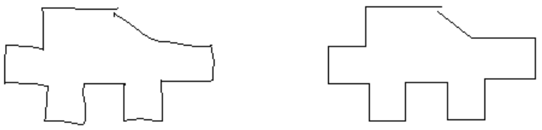


Fig. 6. At left the original sketch of a piece of metal revisited, and the final beautified output at right.

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

Fig. 8 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.[5]

At this point the only evaluation is an informal comparison of the raw sketch and the system's approximations, determining 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.

It is worth noting that among the examples in Fig. 8 the number of loops in the spring strokes, and the number of shading lines in the ground objects are not necessarily equal to their counterparts in the recognized versions, because in each case, the raw strokes are replaced with a generic object of the recognized type. These generic objects only carry the important attributes of the strokes forming them (such as the end points of the spring, and the bounding boxes of both the spring and the ground object). On the other hand, features such as the number of loops in the spring stroke, and the number of shading lines in the ground object need not to be preserved (unless we face a scenario where they are considered to contain information). These choices are made by the higher level recognition system.

---

[5] These examples also show some higher level domain specific recognition. 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.

We have also begun to deal with overtracing, one of the (many) things that distinguishes freehand sketches from careful diagrams. Fig. 7 illustrates one example of the limited ability we have thus far embodied in the program.

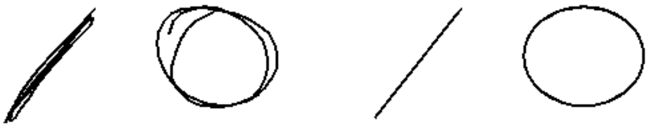


Fig. 7. An overtraced oval and a line along with and the system's output.

4 Related work

The Phoenix sketching system [16] 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 [6] 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 by user lifting 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 [8] 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. [9] 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.

5 Future Work

Future directions for this work include user studies to measure the degree to which the system is both natural and accurate, i.e., supplies the feeling of freehand sketching while still successfully interpreting the strokes.

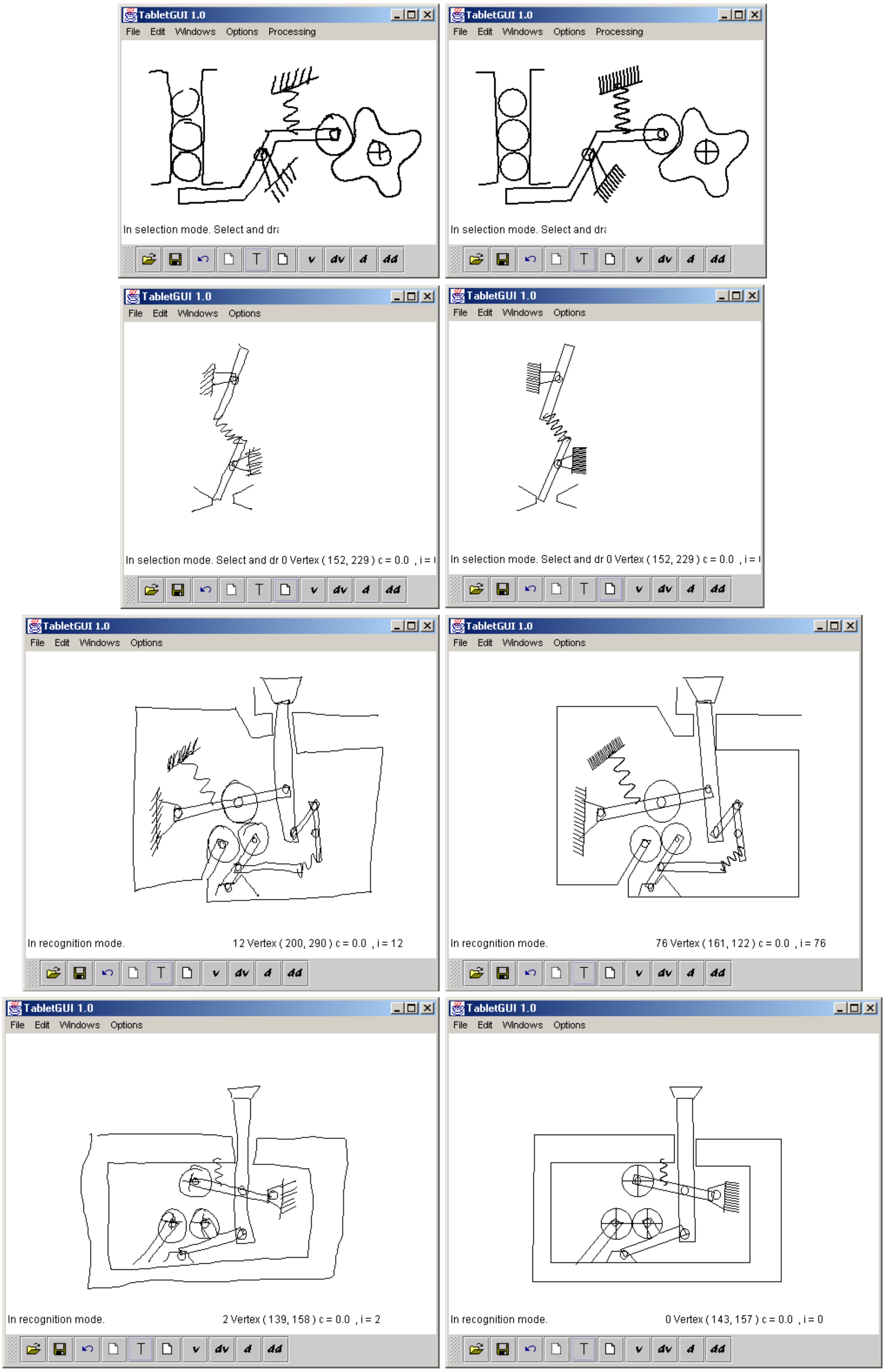
We are also working to link this early processing to other work in our group that has focused on recognition [1] of higher level mechanical objects. This will provide the opportunity to add model-based processing of the stroke, in which early operations like vertex localization may be usefully guided 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* judgements about the size of relevant features. In the pattern recognition community [4], [12] and [11] 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 direction change data. We believe this may allow us to deal with sketches that contain relevant details at a variety of scales more effectively. There is no 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. Combining this idea with machine learning methods may result in interesting results.

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 use of computers far earlier in the design cycle.



In selection mode. Select and dr

In selection mode. Select and dr

In selection mode. Select and dr 0 Vertex (152, 229) c = 0.0 , i =

In selection mode. Select and dr 0 Vertex (152, 229) c = 0.0 , i =

In recognition mode. 12 Vertex (200, 290) c = 0.0 , i = 12

In recognition mode. 76 Vertex (161, 122) c = 0.0 , i = 76

In recognition mode. 2 Vertex (139, 158) c = 0.0 , i = 2

In recognition mode. 0 Vertex (143, 157) c = 0.0 , i = 0

Fig. 8. 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.

References

[1] Christine 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. David. *Perceptual Organization and Visual Recognition*. Kluwer Academic Publishers, 1985.

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

[7] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj 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.

[8] R. Zeleznik et al. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH'96*, pages 163–170, 1996.

[9] T. Igarashi et. al. Interactive beautification: A technique for rapid geometric design. In *UIST '97*, pages 105–114, 1997.

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

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

[12] 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, April 1992.

[13] 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.

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

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

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

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