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

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. With the increasing availability of tablet notebooks and pen based PDAs, sketch based interaction has gained attention as a natural interaction modality. 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 tasks 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. We also show how this early processing system can be used as part of a fast sketch recognition system with polynomial time segmentation and recognition algorithms.

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). This will, we believe, produce a sketching interface that feels much more natural, unlike Graffiti and other gesture-based systems (e.g., [9], [18]), 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 users 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 [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 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).

The rest of the paper is organized as follows: We start with a discussion of the sketch understanding task. In section 3, we describe our stroke approximation system that takes a raw stroke as input and returns its geometric approximation. In section 4, we present an evaluation of the system. Section 5 presents an application using our system for efficient sketch recognition to illustrate how our stroke approximation scheme can be used to build sketch recognizers. We conclude with a discussion of related and future work.

2. THE SKETCH UNDERSTANDING TASK

Sketch understanding overlaps in significant ways with the extensive body of work on document image analysis generally (e.g., [3]) and graphics recognition in particular (e.g., [20]), 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).

Also some domains such as the mechanical engineering design present 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), a Mimio whiteboard system and Tablet PCs. 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 two phases *approximation* and *beautification*. 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.

3.1 Vertex Detection and 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. 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

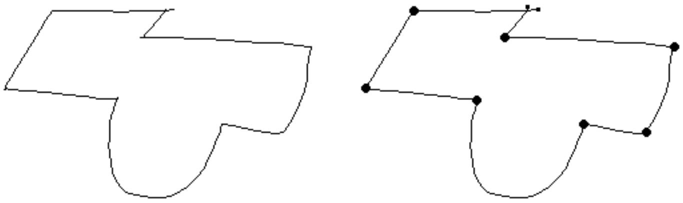


Figure 1: The stroke on the left contains both curves and straight line segments. The points we want to detect in the vertex detection phase are indicated with large dots in the figure on the right. The beginning and the end points of the stroke are indicated with smaller dots.

below). This is unlike the well studied problem of piecewise linear approximation [17].

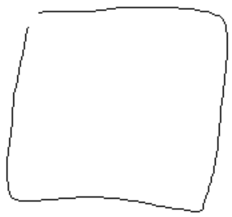


Figure 2: Stroke representing a square.

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.

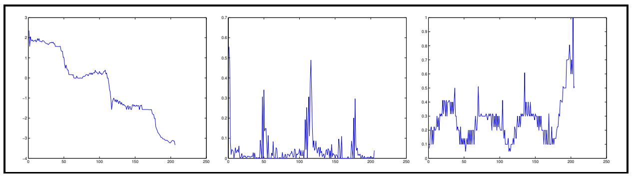


Figure 3: Direction, curvature and speed graphs for the stroke in Fig. 2. In all plots, the x axis indicates the indices of points in the stroke.

3.1.1 *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

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

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.

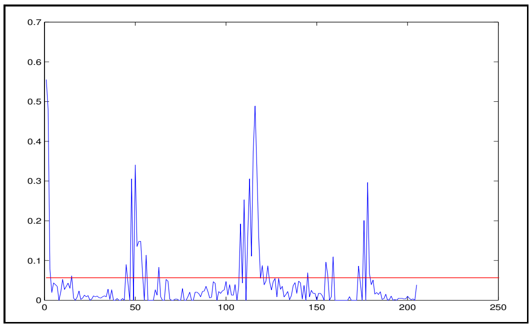


Figure 4: Curvature graph for the square in Fig. 2 with the threshold dividing it into regions.

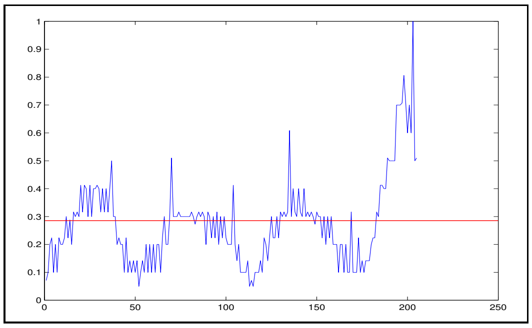


Figure 5: Speed graph for the stroke in Fig. 2 with the threshold dividing it into regions.

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]

Application to speed change

[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.

[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."

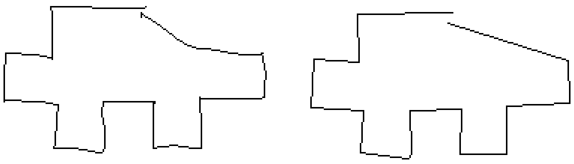


Figure 6: At left the original sketch of a piece of metal; at right the fit generated using only curvature data.

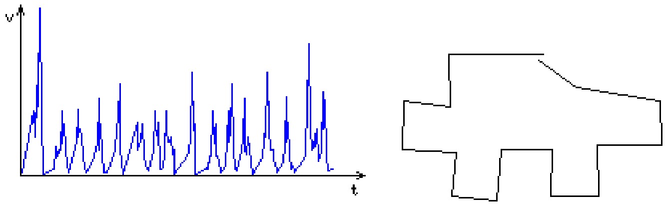


Figure 7: At left the speed graph for the piece; at right the fit based on only speed data.

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). It is important to note that speed data isn't sufficient by itself either. We discuss the issue of generating hybrid fits using information from both fits later in the paper.

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 discuss next, our scale space based technique provides a better approach for dealing with noisy data without having to make a priori assumptions about the scale of relevant features.

3.1.2 Scale space approach

An inherent property of real-world objects is that they exist as meaningful entities over a range of scales. The classical example is a tree branch. A tree branch is meaningful at the centimeter or meter levels. However the concept of a branch looses its meaning at very small scales where cells, molecules or atoms make sense, or at very large scales where forests and trees make sense.

As humans we are good at detecting features at multiple scales, and we don't realize the difficulties posed by the multiscale nature of data interpretation. However, computers don't have a sense of scale. When we use computers to interpret sampled data, we have to take features at multiple scales into account, because digital data is degraded by noise and digitization.

In the case of stroke approximation, there are problems posed by noise and digitization. In addition, selecting an a priori scale has the problem of not lending itself to different scenarios where object features and noise may vary. There's a need to remove the dependence of our algorithms on preset thresholds.

A technique for dealing with features at multiple scales is to look at the data through multiple scales. The scale space representation framework introduced by Witkin [21] attempts to remove the dependence on constant thresholds and making a priori assumptions about the data. It provides us with a systematic framework for dealing with the kind of data we are interested in.

The virtues of scale-space approach are twofold. First, it enables multiple interpretations of the data. These interpretations range from descriptions with high detail to descriptions that capture only the overall structure of the stroke. Second, the scale space approach sets the stage for selecting a scale or a set of scales by looking at how the interpretation of the data changes and features move in the scale space as the scale is varied.

The intuition behind scale-space representation is generating successively higher level descriptions of a signal by convolving it with a filter that does not introduce new feature points as the scale increases.

We use the Gaussian defined as:

$$g(s,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-s^2/2\sigma^2}$$

where σ is the smoothing parameter that controls the scale. Higher σ means coarser scales describing the overall features of the data, while a smaller σ corresponds to finer scales containing the details. The Gaussian filter satisfies the restriction of not introducing new feature points. The uniqueness of the Gaussian kernel for use in scale-space filtering is discussed in [22] and [2].

In the continuous case, given a function $f(x)$, the convolution is given by:

$$F(x,\sigma) = f(x) * g(x,\sigma) = \int_{-\infty}^{\infty} f(u) \frac{1}{\sigma\sqrt{2\pi}} e^{(x-u)^2/2\sigma^2}\, du$$

We use the discrete counterpart of the Gaussian function which satisfies the property:

$$\sum_{i=0}^{n} g(i,\sigma) = 1$$

Given a Gaussian kernel, we convolve the data using the following scheme:

$$x_{(k,\sigma)} = \sum_{i=0}^{n} g(i,\sigma) x_{k-\lfloor n/2+1 \rfloor + i}$$

There are several methods for handling boundary conditions when the extent of the kernel is beyond end points. In our implementation we assume that for $k - \lfloor n/2+1 \rfloor + i < 0$ and $k - \lfloor n/2+1 \rfloor + i > n$ the data is padded with zeroes on either side.

Scale space provides a concise representation of the behavior of the data across scales, but doesn't provide a generic scale selection methodology. There is no known task independent way of deciding which scales are important when looking at the scale-space map for some data. On the other hand it is possible to formulate scale selection methods by observing the properties of the scale-space map for a given task such as edge detection or ridge detection. In the following subsections, we explain how we used the feature count for scale selection in shape approximation. Our goal is selecting a scale where the extrema due to noise disappear.

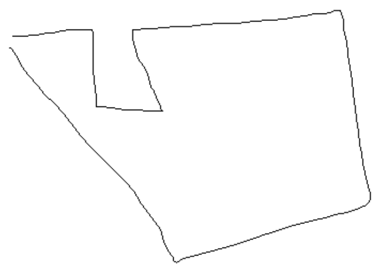


Figure 8: A freehand stroke.

Application to curvature data

As we did in the average based filtering, we start by deriving direction and curvature data. Next we derive a series of functions from the curvature data by smoothing it with Gaussian filters of increasing σ. Then we find the zero crossings of the curvature at each scale and build the scale-space.

Scale-space is the (x,σ)-plane where x is the dependent variable of function $f(.)$ [21]. We focus on how maxima of curvature move in this 2D plane as σ is varied.

Fig 8 shows a freehand stroke and Fig. 9 the scale space map corresponding to the features obtained using curvature data. The vertical axis in the graph is the scale index σ (increasing up). The horizontal axis ranging from 0 to 178 indicates the indices of the feature points in the scale space. The stroke in question contains 179 points. We detect the feature points by finding the negative zero-crossings of the derivative of absolute value of the curvature at a particular scale. We do this at each scale and plot the corresponding point (σ, i) for each index i in the scale space plot. An easy way of reading this plot is by drawing a horizontal line at a particular scale index, and then looking at the intersection of the line with the scale-space lines. The intersections indicate the indices of the feature points at that scale.

As seen in this graph, for small σ (bottom of the scale space graph), many points in the stroke end up being detected as vertices because at these scales the curvature data has many local maxima, most of which are caused by the noise in the signal. For increasing σ, the number of feature points decreases gradually, and for the largest scale σ_{max} (top of the scale space graph), we have only three feature points left, excluding the end points.

Our goal at this stage is to choose a scale where the false positives due to noise are filtered out and we are left with the real vertices of the data. We want to achieve this without having any particular knowledge about the noise[4] and without having preset scales or constants for handling noise.

The approach we take is to keep track of the number of

[4]The only assumption we make is that the noise is smaller in magnitude than the feature size.

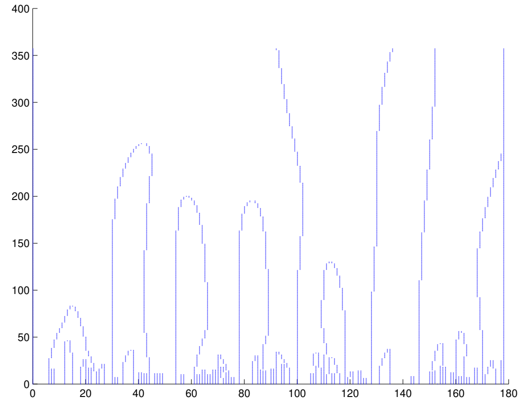


Figure 9: The scale-space for the maxima of the absolute curvature for the stroke in Fig. 8. This plot shows how the maxima move in the scale space. The x axis is the indices of the feature points, the y axis is the scale index.

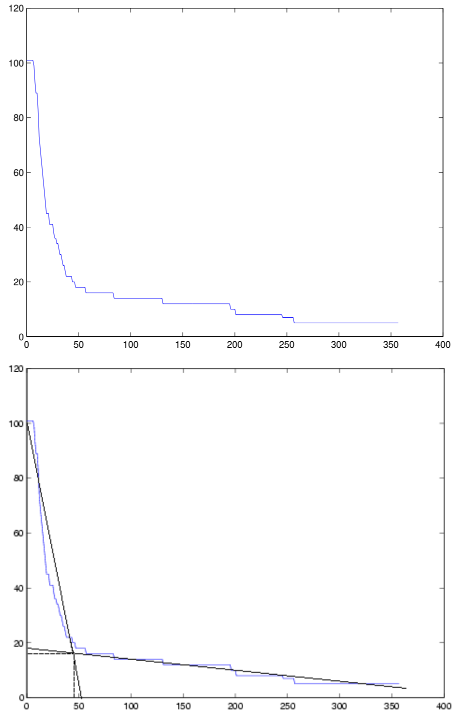


Figure 10: The plot on the left shows the drop in feature point count for increasing σ. The plot at right shows the scale selected by our algorithm (in both, the y axis is the feature count, x is the scale index).

feature points as a function of σ and find a scale preserving the tradeoff between choosing a fine scale where the data is too noisy and introduces many false positives, and choosing a coarse scale where true feature points are filtered out. For example, the stroke in Fig. 8, has 101 feature points for $\sigma = 0$. On the coarsest scale, we are left with only 5 feature points, two of which are end points. This means 4 actual feature points are lost by the Gaussian smoothing. Because the noise in the data and the shape described by the true feature points are at different scales, it becomes possible to detect the corresponding ranges of scales by looking at the feature count graph.

For this stroke, the feature count graph is given in Fig. 10. In this figure, the steep drop in the number of feature points that occurs for scale indices $[0, 40]$ roughly corresponds to scales where the noise disappears, and the region $[85, 357]$ roughly corresponds to the region where the real feature points start disappearing. Fig. 11 shows the scale space behavior during this drop by combining the scale-space with the feature-count graph. In this graph, the x, y, axis z, respectively correspond to the feature point index [0,200], σ [0,400], and feature count [0,120]. We read the graph as follows: given σ, we find the corresponding location in the y axis. We move up parallel to the z axis until we cross the first scale space line[5]. The z value at which we cross the first scale space line gives the feature count at scale index σ. Now, we draw an imaginary line parallel to the x axis. Movements along this line correspond to different feature indices, and its intersection with the scale space plot corresponds to indices of feature points present at scale index σ. In this figure, the steep drop in the number of feature points that occurs for scale indices $[0, 40]$ roughly corresponds to scales where the noise disappears, and the region $[85, 357]$ roughly corresponds to the region where the real feature points start disappearing. Fig. 11 shows the scale space behavior during this drop by combining the scale-space with the feature-count graph. In this graph, the x, y, axis z, respectively correspond to the feature point index [0,200], σ [0,400], and feature count [0,120]. The steep drop in the feature count is seen in both Fig. 10 and Fig. 11.

[5]The first scale space line corresponds to the zeroth point in our stroke, and by default it is a feature point and is plotted in the scale space plot. This remark also applies to the last point in the stroke.

Our experiments suggest that this phenomena (i.e., the drop) is present in all hand drawn curves. For scale selection, we make use of this observation. We model the feature count - scale graph by fitting two lines and derive the scale using their intersection. Specifically, we compute a piecewise linear approximation to the feature count - scale graph with only two lines, one of which tries to approximate the portion of the graph corresponding to the drop in the number of feature points due to noise, and the other that approximates the portion of the graph corresponding to the drop in the number of real feature points. We then find the intersection of these lines and use its x value (i.e., the scale index) as the scale. Thus we avoid extreme scales and choose a scale where most of the noise is filtered out.

Fig. 10 illustrates the scale selection scheme via fitting two lines l_1, l_2 to the feature count - scale graph. The algorithm to get the best fit simply finds i that minimizes $OD(l_1, \{P_j\}) + OD(l_2, \{P_k\})$ for $0 \leq j < i$, $i \leq k < n$. $OD(l, \{P_m\})$ is the average orthogonal distance of the points P_m to the line l, P is the array of points in the feature count - scale graph indexed by the scale parameter and $0 \leq i < n$ where n is the number of points in the stroke. Intuitively, we divide the feature count - scale graph into two regions, fit an ODR line to each region, and compute the orthogonal least squares error for each fit. We search for the division that minimizes the sum of these errors, and select the scale

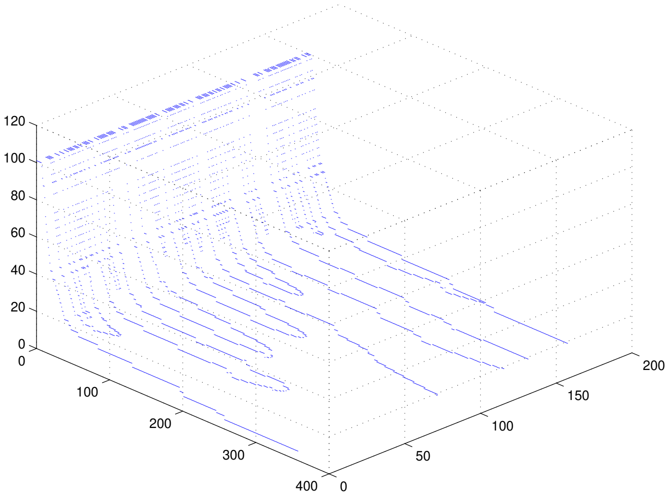


Figure 11: Joint scale-space feature-count graph for the stroke in Fig. 8. This plot simultaneously shows the movement of feature points in the scale space and the drop in feature point count for increasing σ. Here the z axis is the feature count [0,120], the x axis is the feature point index [0,200], and the y axis is the scale index [0,400].

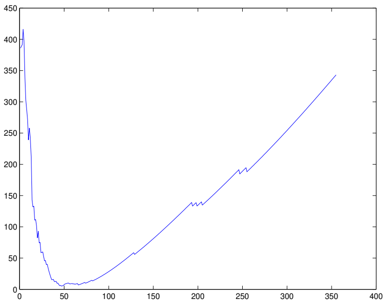


Figure 12: The summed error for the two lines fit to Fig. 10 during scale selection for the stroke in Fig. 8.

corresponding to the intersection of the lines for which the division is optimal (i.e., has minimum error).

Interestingly enough, we have reduced the problem of stroke approximation via feature detection to fitting lines to the feature count graph, which is similar in nature to the original problem. However, now we know how we want to approximate the data (i.e., with two lines). Therefore even an exhaustive search for i corresponding to the best fit becomes feasible. As shown in Fig. 12 the error as a function of i is a U shaped function. Thus, if desired, the minima of the summed error can be found using gradient descent methods by paying special attention to not getting stuck in the local minima. For the stroke in Fig. 8, the scale index selected by our algorithm is 47.

While we try to choose a scale where most of the false maxima due to noise are filtered out, feature points at this scale we may still contain some false positives. The problem of false extrema in the scale space is also mentioned in [15], where these points are filtered out by looking at their separation from the line connecting the preceding and following feature points. They filter these points out if the distance is less than one pixel.

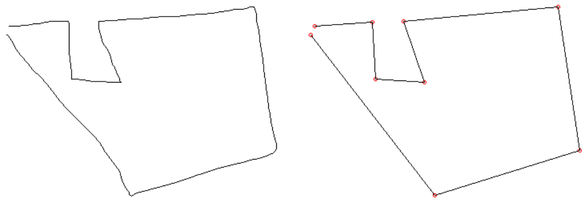


Figure 13: The input stroke (above) and the features detected by looking at the scale space of the curvature (below).

The drawback of the filtering technique in [15] is that the scale-space has to be built differently. Instead of computing the curvature for $\sigma = 0$ and then convolving it with Gaussian filters of larger σ to obtain the curvature data at a particular scale, they treat the stroke as a parametric function of a third variable s, path length along the curve. The x and y components are expressed as parametric functions of s. At each scale, the x and y coordinates are convolved with the appropriate Gaussian filter and the curvature data is computed. It is only after this step that the zero crossings of the derivative of curvature can be computed for detecting feature points. The x and y components should be convolved separately because filtering out false feature points requires computing the distance of each feature point to the line connecting the preceding and following feature points, as explained above. This means the Gaussian convolution, a costly operation, has to be performed twice in this method, compared to a single pass in our algorithm.

Because we convolve the curvature data instead of the x and y coordinates, we can't use the method mentioned above. Instead we use an alternate 2-step method to remove the false positives. First we check whether there are any vertices that can be removed without increasing the least squares error between the generated fit and the original stroke points[6]. The second step in our method takes the generated fit, detects consecutive collinear[7] edges and combines these edges into one by removing the vertex in between. After performing these operations, we get the fit in Fig. 13.

One virtue of the scale space approach is that works extremely well in the presence of noise. In Fig. 14 we have a very noisy stroke. Figures 15 and 16 show the feature-count and scale-space respectively. Fig. 17 combines these two graphs, making it easier to see simultaneously what the feature count is at a particular scale, and what the scale-space behavior is in that neighborhood.

The output of the scale-space based algorithm is in Fig. 18. This output contains only 9 points. For comparison purposes, the output of the average based feature detection algorithm based on curvature is also given in Fig. 18. This fit contains 69 vertices. (The vertices are not marked for the

[6]It is also possible to relax this criteria and remove points if the increase in the least squares error of the segment they belong to remains within some percentage of the original error.

[7]The collinearity measure is determined by the task in hand. In our system, lines intersecting with an angle of $\pi/32$ or less are considered to be collinear.

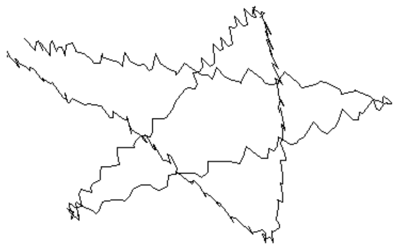


Figure 14: A very noisy stroke.

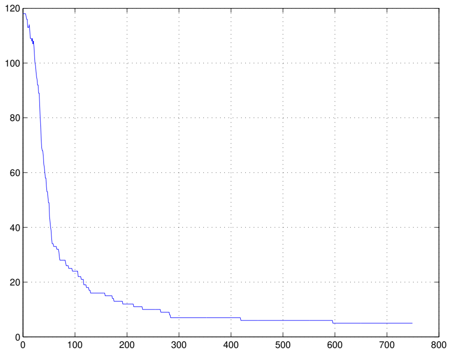


Figure 15: This plot shows the drop in feature point count for increasing σ. y axis is the feature count, and the x axis is the scale index. Even in the presence of high noise, the behavior in the drop is the same as it was for 8. Fig. 17 combines this graph with the feature count graph to illustrate the drop in the feature count and the scale-space behavior.

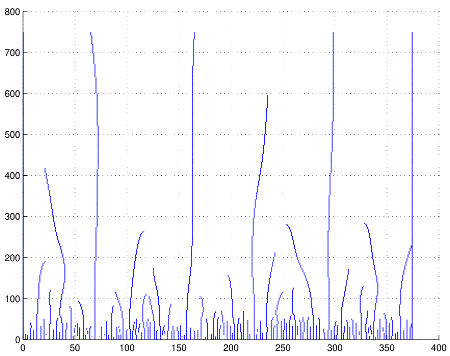


Figure 16: The scale space map for the stroke in Fig. 14. Fig. 17 combines this graph with the feature count graph to illustrate the drop in the feature count and the scale-space behavior.

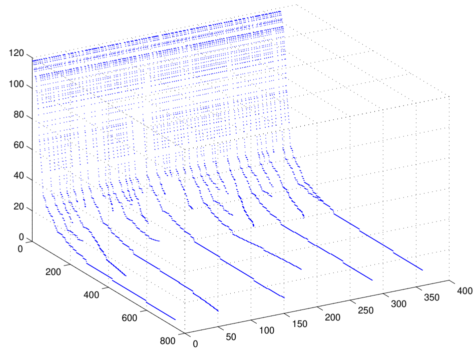


Figure 17: Joint feature-count scale-space graph obtained for the noisy stroke in Fig. 14 using curvature data. This plot simultaneously shows the movement of feature points in the scale space and the drop in feature point count for increasing σ. Here the z axis is the feature count [0,120], the x axis is the feature point index [0,400], and the y axis is the scale index [0,800].

sake of keeping the figure clean.)

Application to speed change

We applied the scale selection technique mentioned above on speed data. The details of the algorithm for deriving the scale-space and extracting the feature points are similar to that of the curvature data, but there are some differences. For example, instead of looking for the maxima, we look for the minima.

Fig. 19 has the scale-space, feature-count and joint graphs for the speed data of the stroke in Fig. 14. As seen in these graphs, the behavior of the scale space is similar to the behavior we observed for the direction data. We use the same method for scale selection. In this case, the scale index picked by our algorithm was 72. The generated fit is in Fig. 18 along with the fit generated by the average based filtering method using the speed data.

For the speed data, the fit generated by scale-space method has 7 vertices, while the one generated by the average based filtering has 82. In general, the performance of the average based filtering method is not as bad as this example may suggest. For example, for strokes as in Fig. 8, the performance of the two methods are comparable, but for extremely noisy data as in Fig. 14, the scale-space approach pays off. This remark also applies to the results obtained using curvature data. Because the scale-space approach is computationally more costly[8], using average based filtering is preferable for data that is less noisy. There are also scenarios where only one of curvature or speed data may be more noisy. For example, in some platforms, the system generated timing data for pen motion required to derive speed may not be precise enough, or may be noisy. In this case, if the noise in the pen location is not too noisy, one can use the faster average based method for generating fits from the curvature data

[8]Computational complexity of the average based filtering is linear with the number of points where the scale space approach requires quadratic time if the scale index is chosen to be a function of the stroke length.

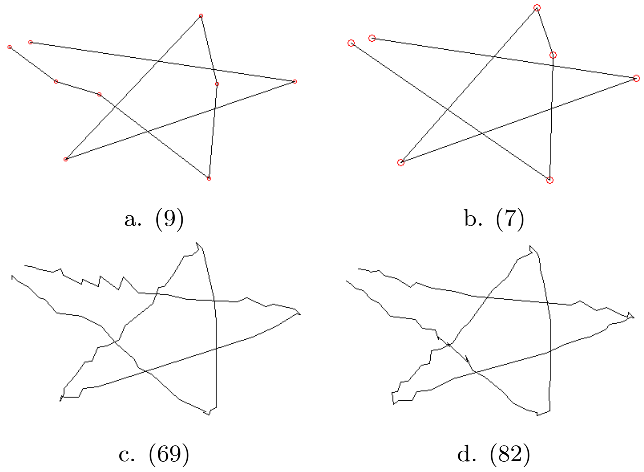


Figure 18: Above, curvature (a) and speed (b) fits generated for the stroke in Fig. 14 with scale space filtering. Below, fits generated using average based filtering (c,d). For each fit, the number of vertices is given in parenthesis.

and the scale-space method for deriving the speed fit. This is a choice that the user has to make based on the accuracy of hardware used to capture the strokes, and and the computational limitations.

3.1.3 Generating hybrid fits

Above we introduced methods for vertex detection using curvature and speed data. As we pointed out, curvature data itself was not sufficient to detect all vertices, motivating our use of speed data. However, using speed data alone has its shortcomings as well. Polylines formed by a combination of very short and long line segments can be problematic: the maximum speed reached along the short segments may not be high enough to indicate the pen has started traversing another edge, causing the entire short segment to be interpreted as a corner. This problem arises frequently when drawing thin rectangles, common in sketches of mechanical devices. Fig. 20 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 corresponding short segment. Fig. 21 shows pen speed for this rectangle. The curvature fit, by contrast, detects all corners, along with some other vertices that are artifacts due to hand dynamics during freehand sketching.

Because both curvature and speed data alone are insufficient for generating good fits in certain scenarios, a method to combine these two information sources is needed. We use information from both sources, and generate hybrid fits by combining the candidate set F_c obtained using curvature data with the candidate set F_s obtained using speed information, taking into account the system's certainty that each candidate is a real vertex.

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 neighbor-

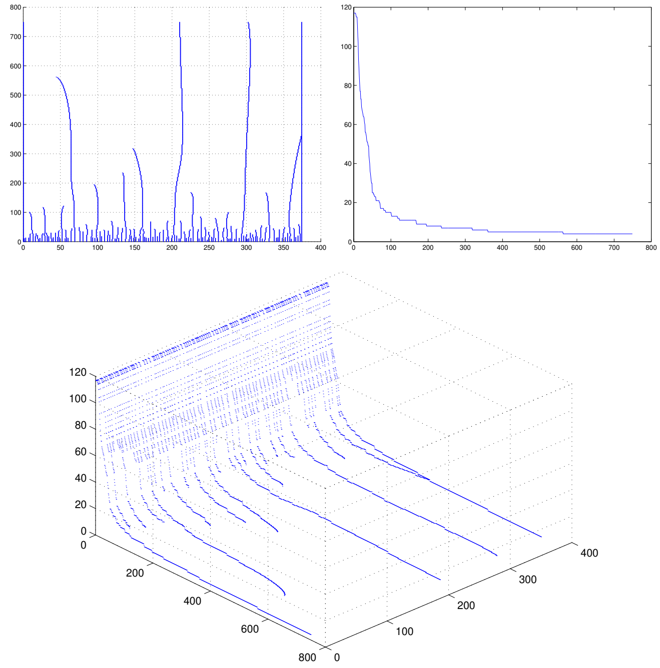


Figure 19: The scale-space, feature-count and joint graphs for the speed data of the stroke in Fig. 14. In this case, the scale selected by our algorithm is 72.

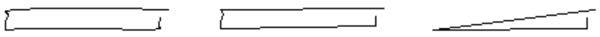
hood around the point, computed as $|d_{i-k} - d_{i+k}|/l$. Here l is the curve length between points S_{i-k}, S_{i+k} and k is a small integer defining the neighborhood size around v_i. The certainty metric for a speed fit candidate vertex v_i is a measure of the pen slowdown at the point, $1 - v_i/v_{max}$, 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



(a) Input, 63 points

(b) Using curvature data, 7 vertices

(c) Using speed data, 4 vertices

Figure 20: 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 short edge of the rectangle on the left).

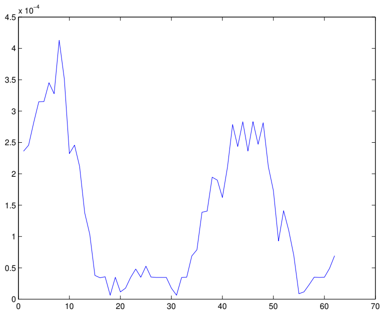


Figure 21: The speed data for the rectangle in Fig. 20.

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.

3.1.4 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:

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

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

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

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.[9]

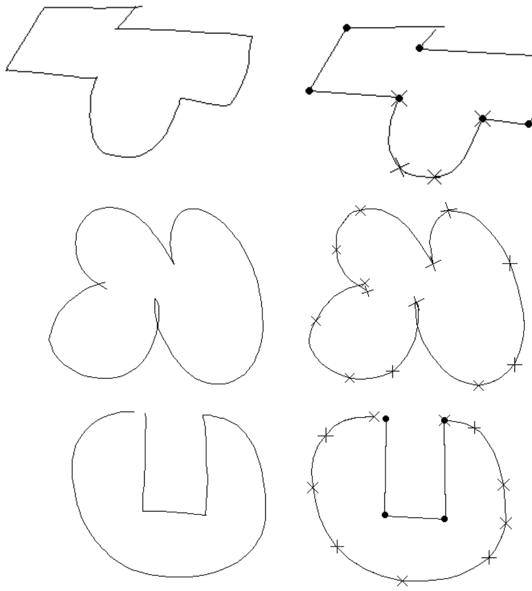


Figure 22: Examples of arbitrary stroke approximation. Boundaries of Bézier curves are indicated with crosses, detected vertices are indicated with dots.

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 [16], hence we resort to an approximation. We discretize the Bézier curve using a piecewise linear curve and compute the error for that curve.

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. 22, 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 Handling Circular Arcs and Ovals

Handling arcs require a circular arc to be fit to the input data. We do this writing the equation for a circle as $(x_i -$

[9] 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 [19].

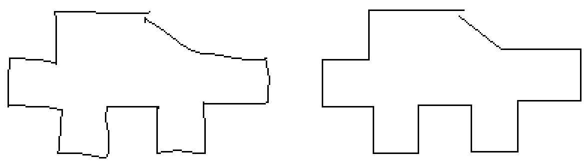


Figure 23: At left the original sketch of a piece of metal revisited, and the final beautified output at right.

$c_x)^2 + (y_i - c_y)^2 = r^2$ where (c_x, c_y) is the center of the circular arc and r is the radius of the arc. We then find the least squares solution for:

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ . & . & . \\ . & . & . \\ 2x_n & 2y_n & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ r' \end{bmatrix} = \begin{bmatrix} {x_1}^2 + {y_1}^2 \\ {x_2}^2 + {y_2}^2 \\ . \\ . \\ {x_n}^2 + {y_n}^2 \end{bmatrix}$$

Here $r'^2 = r^2 - {c_x}^2 - {c_y}^2$. Finally we calculate the starting angle and the extent for the arc.

Oval fit is generated by finding the bounding box of the input stroke and computing the parameters for the oval with the same bounding box. Oval fit is generated separately from the arc fit because the arc fit is strictly circular. As we also mention later, our approximation methods for arcs and ovals work even when objects are drawn using overtracing (See Fig. 25).

3.3 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. 23 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. 30.

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.

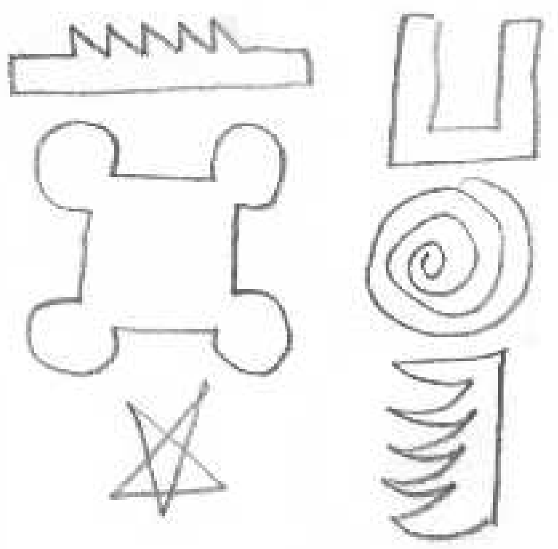


Figure 24: Examples of the shapes used in the user study.

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 24). 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 determine 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, for evaluation purposes, we tested our system by integrating it with the first generation sketch recognition system written earlier in our group for the domain of mechanical engineering drawings. The higher level recognizer took the geometric descriptions generated by our system and combined them into domain specific objects.

Fig. 30 shows the original input and the program's analysis for a variety of simple but realistic mechanical devices

Figure 25: Examples of overtracing. Overtraced objects on the left, system's output on the right.

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. 25 illustrates one example of the limited ability we have thus far embodied in the program.

5. APPLICATION: FAST SKETCH RECOGNITION

So far, we have described a stroke approximation scheme to be used for sketch recognition, but we have not yet described how these approximations can be used. In this section we present an application that uses the approximations generated by our system to achieve fast sketch recognition. As we confirmed with user studies, people have personal sketching styles. They sketch objects with predictable stroke orderings. The novel approach we present in this section shows how viewing sketching as an interactive process allows us to model sketching styles of users using Hidden Markov Models (HMMs). This method enables us to have polynomial time algorithms for structural sketch recognition and segmentation, unlike the conventional methods with exponential complexity.

5.1 Sketch recognition

We characterize the sketch recognition task in terms of three processes:

- *Segmentation:* The task of grouping strokes so that those constituting the same object end up in the same group. At this point it is not known which object the strokes form. For example, in Fig. 26, the correct segmentation gives us four groups of strokes.
- *Classification:* Classification follows segmentation; it is the task of determining which object each group of strokes represent. For Fig. 26, recognition would indicate that the first object in the sketch is a stick figure.
- *Labeling:* The task of labeling components of a recognized object (i.e., the head, the torso, the legs and the arms in the stick figure in Fig. 26).

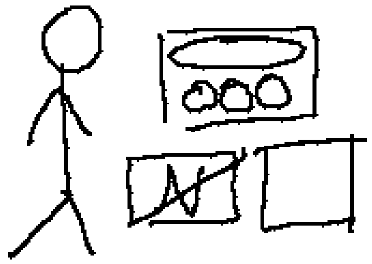


Figure 26: An example sketch.

Current sketch recognition systems treat sketches as images and employ well known object recognition methods to recognize sketches. These systems have two major limitations: First, they are indifferent to who is using the system, employing the same recognition routines for all users eventhough sketching styles for users vary. Second, recognition algorithms employed in these systems suffer from the exponential nature of object recognition. As noted in [12], treating sketches as images requires recognition algorithms with exponential time complexities. There is no known generic method for symbolic object recognition that works in polynomial time. Some of the methods used for symbolic object recognition include subgraph isomorphism based methods with exponential time complexities, or decision-tree based approaches that have exponential storage requirements [13].

We exploit characteristics of sketches separating them from images to build a sketch recognition framework that allows recognition in polynomial time. This framework also makes it possible to learn user sketching styles allowing the recognition parameters to be set based on the current user. We now review characteristics separating sketches from images.

5.2 Characteristics of sketching

Sketches have a number of static properties. Unlike formal drawings, they are messy, and are usually iconic (e.g., a stick figure or a house icon). Sketches are often compositional, for example, a house is formed by composing an isosceles triangle with a rectangle with the triangle above the rectangle.

When viewed as a dynamic process, sketching can be characterized as incremental, with strokes put on the sketching surface one at a time. Sketch recognition can be thought as an interactive process, because in a sense there is a two way communication channel: from the user to the computer in terms of strokes drawn and the editing operations; and from the computer to the user in terms of computer's interpretation of the strokes and editing operations. This interactive nature of sketching is an important source of knowledge when it comes to confirming the correctness of a system's interpretation: The longer the user lets an interpretation exist, the more certain the system can be about its interpretation [1]. Finally, sketching is a highly stylized process; people have strong biases in the way they sketch. This property of sketching forms the basis for our approach to addressing the limitations of traditional approaches to sketch recognition, so we conducted a user study to assess its validity.

5.3 User studies

We ran user studies to assess the degree to which people have sketching styles, by which we mean the way in which they draw an item. For example, if one starts drawing a stick figure with the head, then draws the torso, the legs and the arms respectively, we regard this as a style different from the one where the arms are drawn before the legs. Our user study asked users to sketch various icons, diagrams and scenes from six domains. Example tasks given to the participants included drawing:

- Finite state machines performing simple tasks such as recognizing a regular language.
- Unified Modeling Language (UML) diagrams depicting the design of simple programs.
- Scenes with stick figures playing certain sports.
- Course of Action Diagram symbols used in the military to mark maps and plans.
- Digital circuit diagrams that implement a simple logic expression.
- Emoticons expressing happy, sad, surprised and angry faces.

We asked 10 subjects to sketch three instances from each of the six domains, a total of 18 sketches each. Users were asked to sketch scenes in an arbitrary order to intersperse domains and reduce the correlation between sketching styles used in different instances of sketches from the same domain. Sketches were captured using a digitizing LCD tablet.

Our analysis of the sketches involved constructing multiple sketching style diagrams for each user, one for each object in our domains. *Sketching style diagrams* provide a concise, graphical way of representing how different instances of the same object were drawn. Nodes of a sketching style diagram correspond to partial drawings of an object; nodes are connected by arcs that correspond to strokes. Fig. 27 illustrates the sketching style diagram for the stick figure example described above. Our inspection of the style diagrams revealed that:

- People sketch objects in a highly stylized fashion. In drawing the stick figure, for example, one of our subjects always started with the head and the torso, and finished with the arms or the legs (Fig. 27).
- Sketching styles for individual users agree across sessions.
- Subjects preferred an order (e.g., left-to-right) when drawing symmetric objects (e.g., the two arms) or arrays of similar objects (e.g., three collinear circles).
- Enclosing objects were usually drawn first (e.g., the outer circle in happy faces).

The user study confirmed our conjecture about the stylized nature of sketching. In order to capitalize on this structure we used Hidden Markov Models (HMMs) to model different sketching styles. HMMs are appropriate for this task because sketching can be seen as generating a sequence of strokes (observations), and HMMs have successfully been used to analyze sequence data. Next we briefly review HMMs and explain how we applied them to the problem at hand.

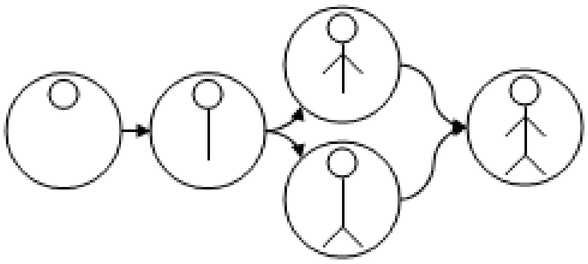


Figure 27: Sketching style diagram for a stick figure illustrating two different ways of drawing stick figures: In both cases the stick figure was drawn using four strokes, but the order in which the legs and arms were drawn is different.

5.4 Overview of HMMs

An HMM $\lambda(A, B, \pi)$ is a doubly stochastic process for producing a sequence of observed symbols. Intuitively, an HMM is a finite state machine with probabilities attached to node transitions and observation emissions. An HMM is specified by three parameters A, B, π. A is the transition probability matrix $a_{ij} = P(q_{t+1} = j | q_t = i)$, B is the observation probability distribution $B_j(v) = P(O_t = v | q_t = j)$, and π is the initial state distribution. $Q = \{q_1, q_2, ...q_N\}$ is the set of HMM states and $V = \{v_1, v_2, ...v_M\}$ is the set of observations symbols. Readers are referred to [14] for a comprehensive tutorial on HMMs.

Given an HMM $\lambda(A, B, \pi)$, we can answer three questions:

- Given a sequence of observations $O = o_1, o_2, .., o_k$, we can efficiently (i.e., in polynomial time) compute $P(O|\lambda)$ using the Forward-Backward algorithm.
- For a given sequence O, we can efficiently compute the best sequence of state transitions for generating O. This is done using the Viterbi algorithm.
- Given a set of observations, we can estimate HMM parameters A, B and π to maximize $P(O|\lambda)$. This is done with the Baum-Welch parameter estimation procedure.

5.5 The recognition system

HMMs have been successfully used in speech recognition, and our approach was in part inspired by this, because both speech and sketching are linear in time and compositional. In speech, phonemes form words, words form sentences and in sketching strokes form objects, objects form scenes. Having pointed out this similarity, we now describe how we use the stoke approximation scheme we introduced in this paper within our framework for sketch recognition.

5.5.1 Encoding

Given a stroke, we generate a geometric approximation of the stroke using our system, so we get one of oval, line, polyline, curve, or a mixture of curves and polylines. We encode the approximations to convert sketches into observation sequences to be used in HMM training and classification. Our encoding has a total of 13 symbols. Four of them encode lines: positively and negatively sloped lines, horizontal and vertical lines. Three encode ovals: circles, horizontal and vertical ovals. Four symbols encode polylines with 2, 3, 4, and 5+ edges, and one symbol encodes complex approximations (i.e., mixture of curves and lines). Finally we have a symbol that we use to denote two consecutive intersecting strokes.

Instances of the same object sketched in different styles may have encodings of different lengths. For example, if the user draws a square in four separate strokes instead of three, the corresponding encoding will be longer. Thus, our training data will have varying length encodings, and both training and classification routines take this into account. Below we describe how we formulate training and recognition using a single model per input length of each class.

5.5.2 *Modeling with HMMs*

Assume we have n object classes. Encodings of training data for class i may have varying lengths, so let $L_i = \{l_{i1}, l_{i2}, ...l_{ik}\}$ be the distinct encoding lengths for class i. We partition the training data into $K = \sum_{i=1}^{n} |L_i|$ sets such that the data in each partition comes from the same object and have the same length. Now we train K HMMs, one for each set, using the Baum Welch method. Note that each training set has uniform length, thus we know the input length for each HMM. In this framework, each class i is represented by $|L_i|$ HMMs, and we have an inverse mapping that tells us which HMMs correspond to which classes.

With the above setup, performing isolated object recognition is quite easy. We run the Forward-Backward procedure with the encoding O generated from the isolated object as its input. The Forward-Backward procedure gives us the probability $P(O|\lambda_i)$. We simply do this for each HMM, find the model with the highest likelihood, and use the inverse mapping to get the object class. Unfortunately isolated object recognition requires the input sketch to be segmented, and segmentation is itself a hard problem.

We perform segmentation and classification at the same time by transforming the problem into a shortest path problem such that the shortest path in a graph that we generate gives us the segmentation. We then perform classification as described above. We begin by building a graph $G(V, E)$: V consists of $|O|$ vertices, one per observation, and a special vertex v_f denoting the end of observations. Let k be the input length for model λ_i. Starting at the beginning of the observation O, for each observation symbol O_s, we take a substring $O_{s,s+k}$. Next we compute the loglikelihood of this substring given the current model, $log(P(O_{s,s+k}|\lambda_i))$, and add a directed edge from vertex v_s to vertex v_{s+k} in the graph with an associated cost of $|log(P(O_{s,s+k}|\lambda_i))|$. No edges are added if the destination index $s + k$ exceeds the index of v_f. We complete the construction of G by repeating this operation for all models. In the constructed graph, having a directed edge from vertex v_i to v_j with cost c means that it is possible to account for the observation sequence $O_{i,j}$ with some model with a loglikelihood of $-c$. It is important to note that the constructed graph may have multiple edges connecting two vertices, each with different costs. By computing the shortest path from v_1 to v_f in G, we minimize sum of negative loglikelihoods which is equivalent to maximizing the likelihood of the observation O. The indices of the shortest path gives us the segmentation. Classification is achieved by finding the models that account for each computed segment.

A nice feature of the graph based approach is that the shortest path in G gives us the most likely segmentation of the input, but it is also possible to get next k-best segmentations using a k-shortest path algorithm. We use the algorithm described in [6] to get alternate segmentations. This information can directly be used by another algorithm for dealing with ambiguities or by the user as it is done in speech recognition systems with n-best lists.

Also, this framework allows fast, scalable segmentation and classification for sketches. Both the Baum-Welch learning algorithm and the Forward-Backward algorithm have polynomial time complexities. In sketch recognition, classification is the time critical operation and unlike other systems with exponential complexities, in our model, computing $P(O|\lambda_i)$ takes only $O(N^2T)$ operations, where N is the number of states in λ_i, and T is the length of the encoded observation. This makes real time recognition feasible.

5.5.3 *System performance*

We conducted an experiment to evaluate the method described above, learning 10 object classes from the domains of geometric objects, military course of action diagrams, stick figure diagrams, and mechanical engineering drawings. Training data was sketched using up to 6 styles with 10 examples per style to capture the variations in encoding for each style. The examples were manually segmented to obtain training data. The test data included 88 objects sketched using the same sketching styles.

Because sketching is incremental, we preferred Bakis (left-to-right) HMM topology where the state index increases or remains the same. This is done by initializing $a_{ij} = 0$ for $i > j$. B, π and the other entries in A are set to random values preserving stochastic properties. We used the maximum number of nodes in the sketching style diagrams obtained from our user study to set the number of states per HMM to 10. The system's correct identification rate was above 95%. Obviously a more through evaluation of system performance with larger training and test sets is desirable, but we regard the above results as encouraging early results supporting the appropriateness of our approach. Fig. 28 shows our system's output in one of the example sketches using the second method. The shortest path graph generated during recognition is shown in Fig. 29.

This recognition framework features fast recognition and segmentation algorithms, and user user adaptability. On the other hand, it is limited in some aspects because it assumes that users sketch with styles that the system knows about. We are also working on generic sketch recognition algorithms that don't assume a drawing order to objects.

The recognition framework we described in this section is one of the many ways in which the output of our stroke approximation system can be used for sketch recognition. The generic sketch recognition algorithms we are developing also use our stroke approximation system to preprocess the incoming strokes and generate concise descriptions in terms of geometric primitives to be used for recognition of domain specific objects. We now move on to describe work related to our stroke approximation system.

6. RELATED WORK

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

- It should be possible to draw arbitrary shapes with a single stroke, (i.e., without requiring the user to draw objects in pieces).

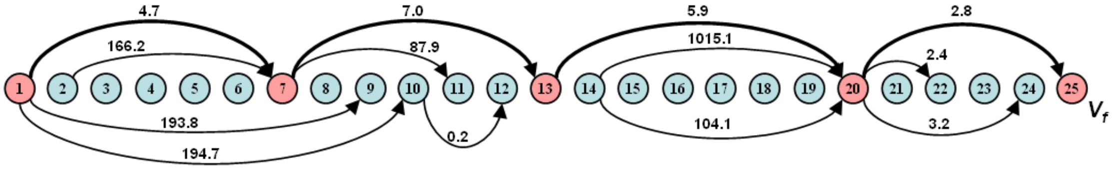


Figure 29: Shortest path graph for the scene depicted in Fig. 28. Edges corresponding to the shortest path are indicated with bold arrows. Nodes 1, 7, 13, 20 and 25 corresponding to beginning and end of objects. Only some of the edges are drawn to avoid cluttering the graph. In Fig. 28, objects were drawn in the following order: CS2, CS1, stick figure, and the rectangle.

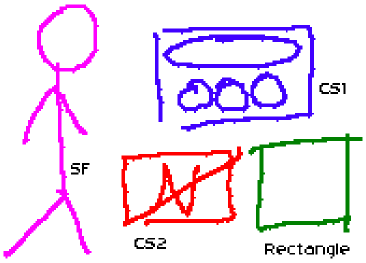


Figure 28: The output of our system for the test case shown in Fig. 26 with drawing order CS2, CS1, stick fig., and rectangle.

- 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 [19] 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 [4] 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 [23] 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 adapt to user styles and 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 [19] and [4] 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 [19] and [4] 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.

7. FUTURE WORK

We see three main directions for future work: making improvements to the current system, carrying out user studies, and integrating this system with other systems that require stroke approximation functionality.

7.1 Potential improvements

In the scale space literature, some authors proposed scale selection methods for computer vision tasks. In particular, in [11] and [10] Lindeberg describes how what he calls *the normalized* γ *derivatives* can be used to guide the scale selection in edge and ridge detection. We plan to explore whether this technique can be adapted for the problem of feature point detection and curve approximation.

7.2 User studies

User studies require choosing a number of domains where users sketch extensively and asking users to sketch naturally as they would with pencil and paper. The studies would measure the degree to which the system is natural i.e., supplies the feeling of freehand sketching while still successfully interpreting the strokes.

Another interesting task would be to observe how designers' sketching styles vary during a sketching session and how this may be used to improve stroke approximation. For example, humans naturally seem to slow down when they draw things carefully as opposed to casually. It would be interesting to conduct user studies to verify this observation and 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.

7.3 Integration with other systems

We are also trying to integrate this system to other work in our group that has focused on higher level recognition of mechanical objects via Bayesian networks [1]. This will provide the opportunity to add model-based processing of strokes in the Bayesian network case so that early operations like vertex localization may be usefully guided by knowledge of the current best recognition hypothesis.

8. 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.

9. REFERENCES

[1] C. Alvarado, M. Oltmans, and R. Davis. A framework for multi-domain sketch recognition. *AAAI Spring Symposium: Sketch Understanding*, 2002.

[2] J. Babaud, A. P. Witkin, M. Baudin, and R. O. Duda. Uniqueness of the gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:26–33, 1986.

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

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

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

[6] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing Vol.28, No.2, pp. 652-673*, February 1988.

[7] 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.

[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.*, 2000.

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

[11] T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision, 30(2):79– 116*, 1998.

[12] J. V. Mahoney and M. P. J. Fromherz. Three main concerns in sketch recognition and an approach to addressing them. *AAAI Spring Symposium: Sketch Understanding*, 2002.

[13] B. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *, IEEE Trans. PAMI, Vol 20, 493 - 505*, 1998.

[14] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE, Vol. 77, no 2.*, February 1989.

[15] 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.

[16] 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.

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

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

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

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

[21] A. Witkin. Scale space filtering. *Proc. Int. Joint Conf. Artificial Intell., held at Karsruhe, West Germany, 1983, published by Morgan-Kaufmann, Palo Alto, California*, 1983.

[22] A. L. Yuille and T. A. Poggio. Scaling theorems for zero crossings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:15–25, 1986.

[23] R. Zeleznik, K. P. Herndon, and J. F. Hughes. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH'96*, pages 163–170, 1996.

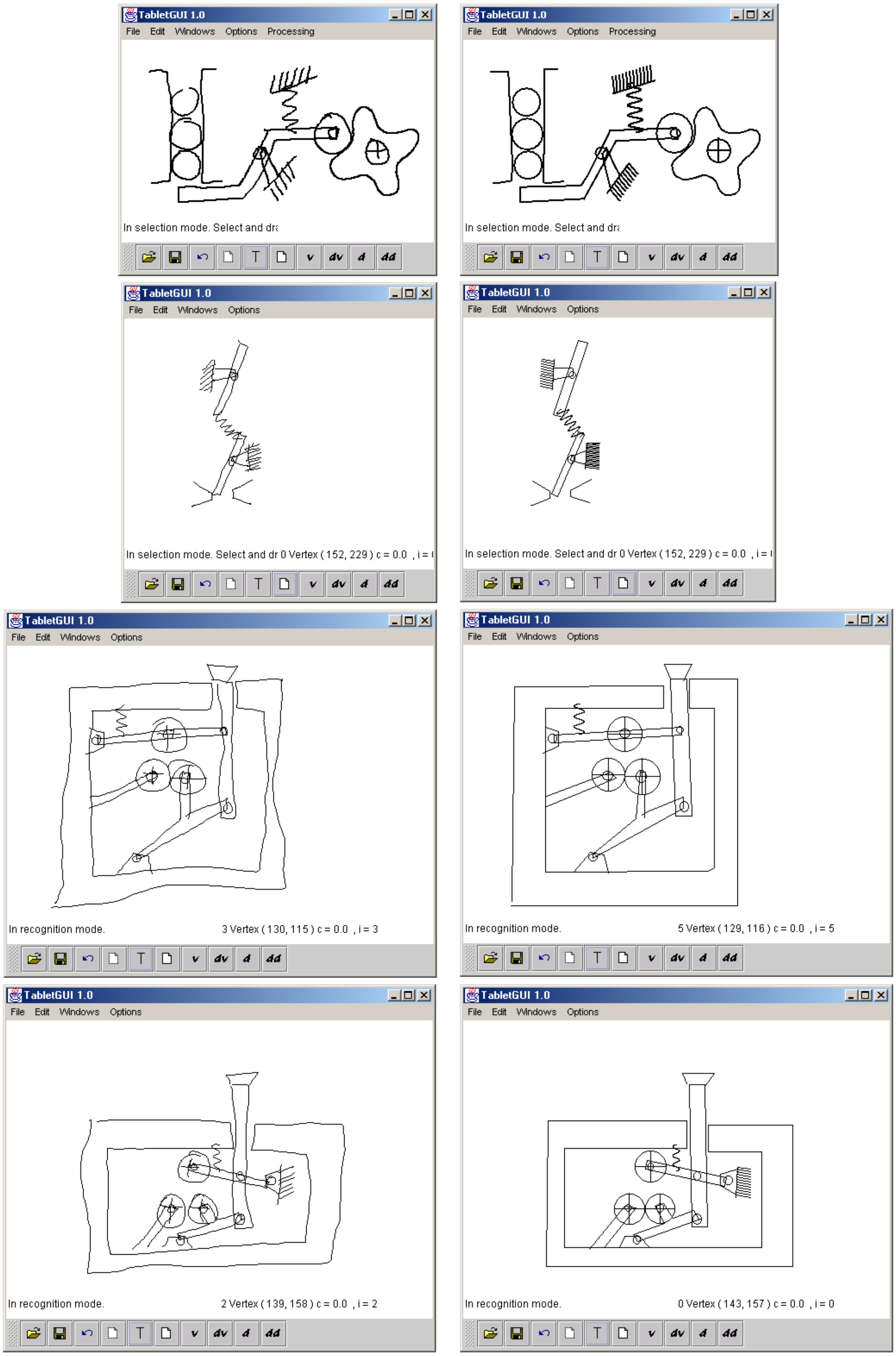


Figure 30: Performance examples: The first two pair are sketches of a marble dispenser mechanism and a toggle switch. The last two are sketches of the direction reversing mechanism in a tape player.