6.835 Project 1: Early Sketch Understanding – Stroke Segmentation

Due: 6:00PM Wednesday, February 20th, 2013

This exercise is intended to give you some hands-on experience in dealing with pen-based computing. We will supply real data (pen strokes) and some basic Matlab code as a foundation. The exercise then involves implementing the corner finding approach in the attached paper by Stahovich [1].

As the paper is at times unclear on the approach it actually uses, we have outlined it for you. Nevertheless, you should read the paper carefully, and may discover that it is useful to refer to it as you work on this project, in order to understand some of the details.

You will no doubt discover that it is not always easy to figure out what the system described in the paper actually did; sadly this is typical of papers in the research literature. As a result it's useful to learn how to dig out the details: If you want to do research, you need to be able to at least reproduce other people's results; even if you simply want to use an idea contained in a paper, you have to be able to decipher it in detail. Sometimes this is more work than it should be, but knowing how to do this is an important skill.

Things to note about this project:

- It will likely take you longer than you expect, so start soon. Don't put it off, or you will not get it done in time. Please email Yale (yalesong@mit.edu) with any questions.
- We expect you to implement what's outlined here, not to reproduce everything described in the paper. If you get inspired you are of course welcome to go beyond what we outline, but make sure you do at least what is listed below.
- This exercise is about getting down to the details of dealing with real pen data. You will confront a collection of issues and details that must be dealt with if you want to make sense of pen input, and get to see how this actually works.

1 Before You Start

This project requires coding with Matlab. If you are not familiar with Matlab, please first refer to numerous tutorials available online and ask for help. The course web page has a listing of tutorials in the Materials section.

2 Getting Started

Download the project file package (`project1.tgz`) from the course web page; a `.zip` file is also available. The file contains some Matlab support code and the dataset `Strokes.mat`. After extracting the file, start Matlab (you need version 7 or later). Here is how to do this on Athena:

```
athena% tar xvfz project1.tgz
athena% cd project1
athena% add matlab
athena% matlab
```

Now add the working directory to your path and load the dataset:

```
>> addpath project1
>> load Strokes
>> showSegmentation(strokes(1), []);
```

`strokes(i)` is a structure representing the i^{th} stroke. It contains arrays x,y,and t, representing the position and timestamp of each sampled datapoint in the stroke.

3 Segmenting Strokes

The main part of the assignment is to identify corners and classify the segments between corners as lines or arcs. There are a number of parameters that may have to be adjusted to improve performance, so be sure to develop code that makes this easy. You're free to structure your code however you want, but you must provide this function:

```
[segpoints, segtypes] = segmentStroke(stroke)
```

- `stroke` is a stroke data structure loaded from the file `Strokes.mat`
- `segpoints` is an (N) x 1 array of the indices of points at which the `stroke` is segmented
- `segtypes` is an (N+1) x 1 binary array indicating the type of each segment: 0 for a line, and 1 for an arc.

Your stroke segmentation should follow the steps outlined in the paper:

1. Construct an array of the cumulative arc lengths between each pair of consecutive sampled points.

2. Construct an array of smoothed pen speeds at each point.

3. Construct an array of tangents[1] at each point.

4. Define curvature, which is the change in orientation over change in position along the arc length. You can convert the slopes to angles using the arc tangent function (`atan`). To see a problem that crops up with using arc tangent (and with using angles in general), plot the arctangent of the slopes that you obtained for `strokes(3)` – the "Ω" symbol, and briefly explain what you think is going on. As this phenomeon this will corrupt your estimates of the curvature (change in angle over time), they must be corrected. Write and test a method called `correctAngleCurve`.

5. Identify corners. The paper is a little unclear on this, but the basic idea is to have two separate criteria, both of which contribute points:

   (a) Using speed alone, select local minima of speed that are lower than a threshold percentage of the average speed, and

   (b) identify local maxima of curvature, and accept them if the speed at that point is below a threshold (even if not a minimum) and the curvature is above a threshold.

6. Combine the two sets of points identified in step 5, with any nearly coincident points merged. You will need to decide what "nearly coincident" should mean, and decide which of the two points to keep. You may also want to prune away corners that are too near the beginning or end of the stroke.

7. You should now have identified a set of segmentation points. For each segment, fit a line using linear regression, and a circle, using the provided code, `circfit`. Based on the residual errors[2] and the angle subtended by the arc of the circle, classify each segment as a line or curve (you can forget about elliptical curves).

8. You can view the output of your segmentation by calling `showSegmentation(stroke, segpoints, segtypes)`. You can view the output on all given shapes in the dataset using `evalAll(strokes)`, which will call your `segmentStrokes` function. In this viewer, segments classified as lines are rendered in blue, and segments classified as arcs are rendered in pink. Corners are indicated with circles. An example of reasonable (though far from perfect) system output is shown in Figure 1.

[1]To do this, fit a linear regression line to a window of points surrounding each point, and note the slopes. The size of this window is one of the parameters you will need to adjust. Matlab provides `regress`, a built-in function for performing linear regression.

Note that the Stahovich paper [1] proposes fitting an orthogonal regression rather than linear regression. Whether or not you try this, please don't use the "Orthogonal linear regression" package from Matlab File Exchange – it's too slow.

[2]Although Matlab's `regress` function provides the residuals for the line fit, you will need to write your own function to compute the residuals of the circle fit.

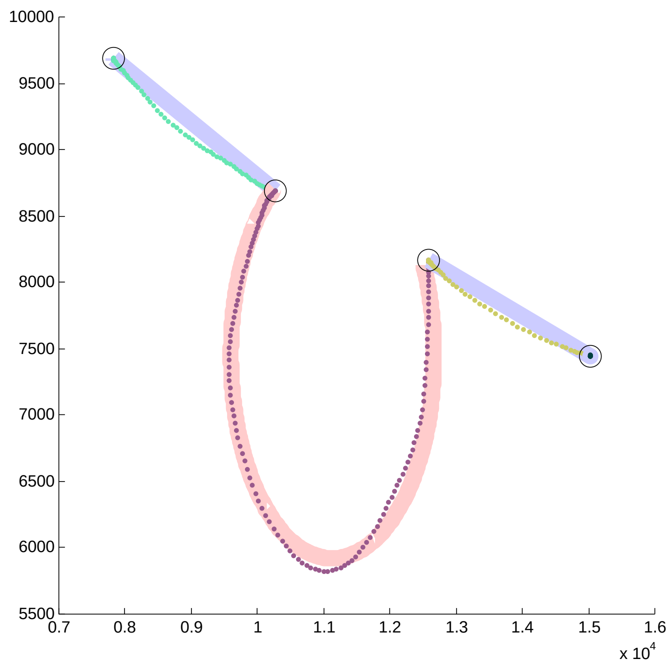

Figure 1: An example system output (not obtained using default parameters)

9. In your code, you should have at least seven parameters:

Parameter	**Value used in the paper**
Size of window for smoothing penspeed	5 total (2 points on each side)
Size of window for computing tangent	11
Speed threshold 1	25% of average
Curvature threshold	.75 degree / pixel
Speed threshold 2	80% of average
Minimum allowed distance between two corners	unspecified
Minimum arc angle	36 degrees

Using these values, evaluate your system and print out the results with `evalAll(strokes)`. Experiment with different values for these parameters, and print out the best results you can obtain, indicating the parameter values that you have chosen.

4 Submission

Please submit the following to the course website in the Homework section:

1. Your source code. Please submit all the `.m` files you created/used in this project.
2. The answer to the question in step 4, regarding the subtlety of arc tangent, and why this will corrupt the estimates of the curvature.
3. Your definition of "nearly coincident", and how you chose the value for the parameter (*i.e.* the minimum allowed distance between two corners).
4. A color print out (pdf) of the results of your system with default parameters.
5. Color print out (pdf) of the results of the best paremeter values you could identify, along with the values.
6. Compare the results of your chosen parameter values with those of the default. If yours turn out to be better, explain what you did, and why it works better.
7. The paper describes several additional steps to improve this system. Which of these (or what other steps) would be the most effective in improving your system as it now stands?

References

[1] Thomas F. Stahovich, Segmentation of pen strokes using pen speed. In *AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural.*

Segmentation of Pen Strokes Using Pen Speed

Thomas F. Stahovich
Mechanical Engineering Department
University of California
Riverside, California 92521
stahov@engr.ucr.edu

Abstract

We present a technique for segmenting pen strokes into lines and arcs. The technique uses pen speed information to help infer the segmentation intended by the drawer. To begin, a set of candidate segment points is identified. This set includes speed minima below a threshold computed from the average pen speed. It also includes curvature maxima at which the pen speed is again below a threshold. The ink between each pair of consecutive segment points is then classified as either a line or arc, depending on which fits best. Finally, a feedback process is employed, and segments are judiciously merged and split as necessary to improve the quality of the segmentation. Formal user studies were conducted, and our system was observed to perform accurately, even for new users.

Introduction

Despite the power and sophistication of modern engineering design tools, engineers often avoid using them until late in the design process. Instead, it is common for engineers to do much of their early work on paper, using sketches extensively. After the major design issues have been resolved, the sketched designs are then recreated on the computer in order to take advantage of the capabilities of design software. The problem here, we believe, is the cumbersomeness of the traditional user interface. When designs are in flux, the inconvenience of such user interfaces places too much overhead on the creative process.

In our research, we are working to change this by creating user interfaces that allow users to operate software by means of familiar sketching skills. The ultimate goal is to create software that is as easy to use as paper and pencil, yet is as powerful as traditional software. Rather than the user having to learn how to use software, software should be able to read, understand, and use the kinds of sketches people ordinarily draw. For example, an engineer should be able operate a mechanical simulation tool by drawing the kinds of simple sketches that he or she would draw when solving problems by hand.

In attempting to create software that embodies the ease and freedom of sketching, care must taken to avoid placing new constraints on the drawing process. For example, some existing sketch-based systems require that each pen stroke represent a single shape, such as a single line or arc segment (Igarashi *et al.* 1997; Shpitalni & Lipson 1996). Other systems allow pen strokes to have more complicated shapes, but, each stroke must constitute a single symbol or gesture (Rubine 1991; Landay & Myers 2001). While these kinds of constraints on drawing facilitate shape recognition, they can result in a less than natural drawing environment.

The work presented here concerns the low level processing of pen strokes necessary to overcome some of these kinds of constraints. In particular, we present an approach for automatically segmenting pen strokes into the intended geometric primitives. Our approach enables one to draw a shape with as few or as many stokes as desired. For example, one can draw a triangle with one, two, or three pen strokes. Likewise, it enables one to include parts of different shapes or symbols in the same pen stroke.

The challenge in segmenting a pen stroke is deciding which bumps and bends are intended, and which are accidents. We have found it difficult to determine this by considering shape alone. The size of the deviation from an ideal line or arc is not a reliable indicator of what was intended: sometimes small deviations are intended, while other times large ones are accidents. Our approach to segmentation relies on examining the motion of the pen tip as the pen strokes are created. We have observed that it is natural to slow the pen when making many kinds of intentional discontinuities in a shape. For example, even if not drawn with four precise lines, the intended corners of a square can be easily identified as points at which the pen speed is a local minimum.

Our segmenter begins by examining a pen stroke to identify the *segment points*, the points that divide the stroke into different primitives. The initial set of candidate segment points includes speed minima below a threshold, which is computed from the average pen speed. Points at which curvature is a maximum are also included, but only if there is corroborating pen speed information. Next, the ink between each pair of consecutive segment points is classified as either a line or an arc, depending upon which best fits the ink. Finally, a feedback process is employed, and segments are judiciously merged and split as necessary to improve the quality of the segmentation.

Our segmenter serves as a foundation for our broader research efforts to build sketch understanding systems. For example, we have constructed parsers that examine a stream of segmented pen strokes and extract individual symbols (Gennari, Kara, & Stahovich 2004; Kara, Gennari, & Stahovich 2004). We have also constructed recognizers that classify symbols by examining their segmented pen strokes (Calhoun *et al.* 2002; Gennari, Kara, & Stahovich 2004). We have combined all of these tools and techniques to build various sketch-based engineering applications, such as a circuit analysis tool (Gennari, Kara, & Stahovich 2004) and a tool for analyzing vibratory systems (Kara, Gennari, & Stahovich 2004).

The next section provides an overview of related work. This is followed by a detailed, technical description of our approach. A user study evaluating the performance of our segmenter is then presented. Finally, proposed future work is discussed, and conclusions are presented.

Background

Segmentation of pen strokes is similar to corner detection in digital curves. Corner detection algorithms typically locate corners by searching for points of maximum curvature. To suppress noise and false corners, the data must be smoothed. The main difficulty is selecting a reliable amount of smoothing. Early approaches (e.g., (Teh & Chin 1989)) relied on a single scale, which created difficulties for curves containing both large and small features. Later work has addressed the problem of curves containing features at various scales. For example, Rattarangsi and Chin (1992) developed a scale-space approach, in which curvature maxima that persist across multiple scales indicate corner points. Likewise, Lee et al. (1995) developed a multi-scale algorithm based on the wavelet transform. Sezgin has applied a multi-scale approach to sketches. His work suggests that curvature data alone is inadequate for segmenting hand-drawn pen strokes (see below).

Yu (2003) has developed a pen stroke segmentation approach in which the curvature and tangent angle are iteratively smoothed. The resulting segmentation is compared to the original ink, and if the fit is not precise, the stroke is recursively subdivided until it is. In our experiments, we have found that a precise fit to the raw ink is often not what the drawer intended.

The earliest report of using pen speed for segmenting we have been able to find is the work of Herot (1976). His system found corners by identifying points at which pen speed was a minimum. The author reported that the system did not work well for all users and he concluded that the program contained a "model of human sketching behavior that fit some users more closely than others."

Sezgin, Stahovich, and Davis (2001) presented a technique that uses speed and curvature to segment hand-drawn pen strokes. Segment points are located at points of minimal speed and maximal curvature. The technique is suitable for segmenting pen strokes into line segments, but it cannot handle arcs. Also, this technique iteratively adds segment points until the error of fit between the line segments and raw ink is less than a threshold. Our approach is far less concerned with the error of fit, as a tight fit to the ink is often not what was intended. As a variant of this technique, Sezgin (2001) explored the use of multi-scale methods for selecting speed minima and curvature maxima. However, he found that unless the pen strokes were exceptionally noisy, there was little benefit in doing so.

Agar and Novins (2003) have developed a segmenter for polygons. If the mouse is stationary for more than 30ms, the location is taken to be a segment point. This is analogous to our pen speed approach, but it requires that the mouse be paused at each corner. Additionally, the approach can handle only line segments and not arcs.

Dudek and Tsotsos (1997) have developed a novel approach that directly searches for segments rather than segment (corner) points. Energy minimization is used to compute an approximation curve that best matches the input curve while at the same time attempting to maintain a desired curvature. If a low energy approximation cannot be found, the approximation curve is subdivided and the process is iterated. This process is repeated with different values of the desired curvature, and can result in overlapping segments with different curvature values. The approach may not be suited to sketches, as most shape recognition techniques assume that segments do not overlap.

Segmenting Technique

To begin the segmentation process, an initial set of candidate segment points is identified. This set includes speed minima below a threshold computed from the average pen speed. It also includes curvature maxima at which the pen speed is below a threshold. The ink between each pair of consecutive segment points is then classified as either a line or an arc, depending on which fits best.

Although the initial segmentation is usually accurate, it can often be improved through feedback. For example, if two adjacent segments form pieces of the same arc, it is likely that they were intended to be so. In this case, the two are merged into a single segment. Conversely, if a segment is a particularly poor fit for the ink, this suggests that a segment point may have been missed. This often occurs when there is a smooth change in the sign of curvature, for example, when moving from one lobe of an "S" shape to the other as shown in Figure 4. This kind of transition can be made without slowing the pen, and so cannot be detected as a speed minimum. Consequently, if a segment is a poor fit for the ink, points at which the curvature changes sign are considered as possible additional segment points.

The sections that follow describe the various steps of the segmentation process including: initial processing of the ink, identification of segment points, fitting of segments, and merging and splitting.

Initial Processing of the Ink

Our software is designed work with a digitizing tablet and stylus, or other similar device, that provides time-stamped coordinates. For example, we have used Wacom Cintiq and Intuos2 tablets, and a Tablet PC. During the initial processing phase, we use the time-stamped coordinates to compute pen speed and curvature. The first step is to construct the arc length coordinate of each point. Arc length is measured along the path of the pen stroke, and is computed in the obvious way by summing straight line distances:

$$d_i = \sum_{j=1}^{i} \left\| \vec{P}_j - \vec{P}_{j-1} \right\| \tag{1}$$

where $\vec{P}_j$ is the coordinates of the j^{th} data point. The first data point has index j = 0 and $d_0 = 0$.

We then use a centered finite difference approach to compute pen speed:

$$s_i = \frac{d_{i+1} - d_{i-1}}{t_{i+1} - t_{i-1}} \tag{2}$$

where t_i is the time-stamp of the i^{th} point. The speed at the first and last point of a pen stroke is taken to be equal to that at the second and penultimate points, respectively. Often, there is noise in the pen speed signal. To correct this, we apply a simple smoothing filter: The speed at each point is averaged with that of the two points on either side.

There are various ways of computing curvature. For example, one could use the standard analytic geometry technique for computing the curvature of parametric curves (Mortenson 1985). Our approach, however, is to compute curvature as the derivative of the tangent angle, θ, with respect to arc length:

$$C = \frac{\partial \theta}{\partial s} \tag{3}$$

We use this approach for several reasons. First, our system already computes an accurate tangent, which is used for other purposes. Second, this method naturally smoothes the data so that no additional smoothing is needed.

To construct the tangent at a given point, we first construct a least squares line fit to a window of data points centered around that point. Using a window of points has the effect of smoothing noise. The larger the window, the larger the smoothing effect. We have found that a window of eleven points provides adequate smoothing without loss of essential information about the shape. If the least squares line is an accurate fit for the window of points, it is used as an approximation of the tangent. Otherwise, a least squares circle fit is constructed, and the tangent is taken from the circle.

We could compute the derivative of the tangent angle by means of numerical derivatives, but this would require smoothing. To avoid this, we again use a least squares line fit, this time applied to the graph of the tangent angle versus arc length. The slope of the least squares line gives the curvature in units of radians per pixel. Here again, when computing the least squares line, we use a window of eleven points.

We have found that our approach to calculating curvature works well in practice. In fact, this approach is similar in spirit to the way draftspersons used to compute graphical derivatives in the era before computers. In some sense, we are smoothing the data the way a draftsperson would by eye. Although not reported here, we have shown that our approach is comparable to the traditional curvature calculation based on the methods of analytic geometry, combined with Gaussian smoothing. Thus, if desired, one could directly implement our segmentation approach using the more traditional technique.

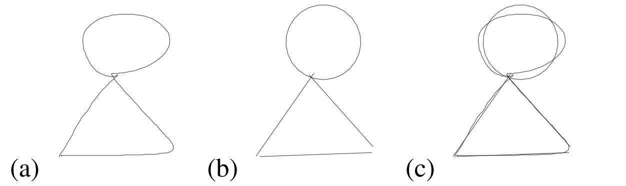
Figure 1: A hand-drawn pivot symbol. (a) Raw ink. (b) Segmented ink. (c) Raw and segmented ink overlayed.

Candidate Segment Points

Once the initial processing of the ink is completed, the next step is to compute the set of initial candidate segment points. The first and last points of a pen stroke are always included in this set. The remaining segment points are identified by examining speed and curvature.

Our most reliable criterion for selecting segment points is based on pen speed. Segment points typically occur at locations where speed is a local minimum. Consider, for example, the sketch of a pivot in Figure 1(a). This sketch, which was drawn with a single pen stroke, was intended to be three lines and an arc (Figure 1(b)). Figure 2 shows the speed profile for the pen stroke. The intended segment points correspond to local speed minima as indicated by circles. There are other speed minima that do not correspond to intended segment points, but these are distinguishable by their higher speed.

Our approach, therefore, is to locate segment points at speed minima that are slower than some threshold. We select the threshold as a fraction of the average speed along the pen stroke. (The ordinate in Figure 2 directly corresponds to possible values of the threshold.) In practice, a threshold of between 25% and 100% of the average speed works well. A larger threshold will decrease the number of intended segment points that are missed, while a smaller value will decrease the number of unintended segment points that are selected.

Interestingly, we have found that our approach is not very sensitive to the particular value of the threshold used. For example, our user studies (below) show little variation in the overall accuracy of the segmentation over the range in threshold from 25% to 100%. The reasons for this are discussed in "Discussion and Future Work."

We typically, use a small threshold (25%) because very low pen speed is a clear indication of an intended segment point. If a speed minimum is above the threshold, the point may still be a segment point, but additional information is required to be certain. In this case, we examine the curvature of the ink. In Figure 2, for example, segment points (i) and (ii) are detected with a threshold of 25%. Segment point (iii) is above this threshold, but, as shown in Figure 3 this point corresponds to a maximum of curvature, which provides additional evidence about the existence of a segment point.

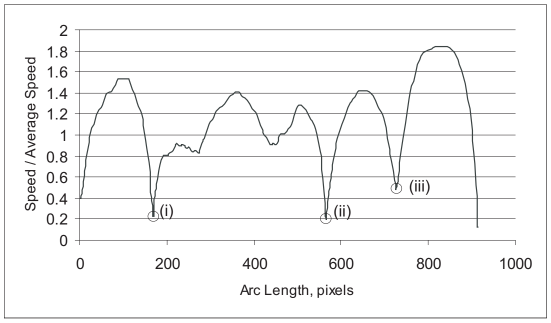

Figure 2: Pen speed normalized by the average speed for the pivot in Figure 1. Intended segment points are indicated by circles.

One approach to identifying segment points would be to identify points that are both a minimum of speed and maximum of curvature. In practice, we have found it adequate to simply identify points that are a maximum of curvature and which have low speed. This avoids problems when speed minima and curvature maxima are nearby, but not precisely coincident.

Based on empirical studies, we have identified a reliable criterion based on both curvature and pen speed: If a point is an extremum of curvature, the absolute value of the magnitude of the curvature exceeds 0.75 degree / pixel, and the pen speed is less than 80% of the average pen speed, the point is included in the initial set of candidate segment points. The second requirement helps with nearly straight lines. Often the sign of the curvature fluctuates for such lines, resulting in multiple extrema. However, because the ink is nearly straight, the magnitude of the curvature at the extrema is quite small. The thresholds use here work well for the hardware we use, and have proven to work well for a wide range of users, but will likely need tuning for other hardware.

The speed-based and curvature-based segment points are always included in the initial set of segment points. There is a third class of segment points that are not considered initially. These are the points at which the curvature changes sign. We define three qualitative "signs" for curvature: +1 if the magnitude is greater than 0.1 degree / pixel, -1 if the magnitude is less than -0.1 degree / pixel, and 0 otherwise. The thresholds were determined empirically to eliminate irrelevant fluctuations in the curvature that occur for nearly straight lines. Again, these values work well for our hardware, but will require tuning for other hardware.

A change in curvature sign is not a reliable indication of an intended segment point, thus such points are considered only when the other segment points result in a poor fit for the ink. For example, it is common for there to be a change in curvature sign on each side of a 90 degree corner. Clearly, such changes in curvature sign do not correspond to intended segment points. For this reason, segment points based on curvature sign are not part of the initial set of candidate segment points. Instead, they are considered only during the splitting process described below.

It is possible for there be to be small clusters of closely located segment points. For example, there may be two speed minima separated by only a few data points. Consequently,

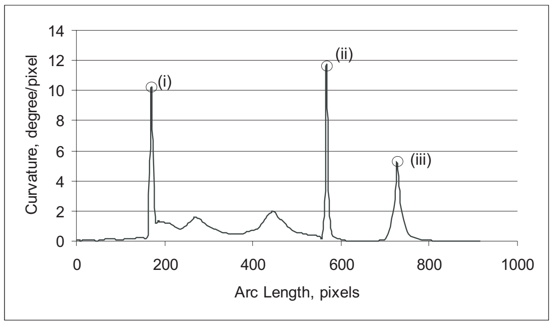

Figure 3: Ink curvature for the pivot in Figure 1.

once the speed and curvature segment points have been calculated, nearly coincident segment points are eliminated.

Fitting Segments

Once the initial set of candidate segment points have been identified, the next step is to fit primitives to the segments. Least squares line and circle fits are constructed for the segment between each pair of consecutive segment points, and the errors of fit are noted. We define *error of fit* as the average distance from the least squares line or arc to the data points in question. The segment is typically classified by whichever shape fits it with the smallest error of fit. However, it is common for nearly straight lines to be accurately fit by an arc with a large radius. Thus, even if a segment is best fit by an arc, it is classified as such only if it would represent at least one tenth of a circle (36^o).

If a segment is classified as a line segment, its end points are determined by constructing perpendiculars from the first and last data points to the least squares line. Similarly, for arcs, the end points are determined by a constructing radial lines through the first and last data points.

Merging and Splitting

After the initial segments have been computed, a quality control process is begun. The segments are compared to the original ink, and segments are merged, split, and deleted as necessary. In this fashion, feedback is used to improve the accuracy of the segmentation.

If there is a very short segment adjacent to a long one, we have found that, frequently, the short one was unintended. Thus, if a segment is shorter than 20% of the length of an adjacent segment, the program attempts to merge them. (This constant, as well as all of the others used for merging and splitting, were obtained empirically.) The program computes a new segment containing the data points of the two original segments. The type (line or arc) of this new segment is forced to be the same as that of the longer of the original two. If the error of fit of the new segment is no more than 10% greater than the sum of the errors of fit of the original two segments, they are discarded and replaced with the new one. Otherwise, the new segment is discarded.

We have found that at the start and end of a pen stroke, the stylus often leaves small, unintended bits of ink that form sharp discontinuities. Therefore, we eliminate segments at the start or end of a pen stroke containing fewer than 15 points. Similarly, if the first or last segment is shorter than 10% of the average length of the three immediate neighbors, it is discarded.

If adjacent segments are of the same type, the program checks to see if they might reasonably be interpreted as the same segment. For example, if two arcs are adjacent, the program computes a new arc containing the data points from the two original arcs. If the error of fit is no more than 10% greater than the sum of the original errors of fit, the two arcs are replaced by the new one. Note that the program considers merging two segments only if their drawing directions are consistent.

If a particular least squares line or arc does not fit the ink well, the program attempts to improve the fit by including a segment point based on a change in the sign of the curvature. The program splits a segment in this fashion if the error of fit is greater than seven pixels. In other words, if on average, the data points are at least seven pixels from the least squares line or arc, the program attempts to split the segment. This value was determined empirically to work with our hardware, but will likely require tuning for use with other hardware.

Typically there are only a few curvature-sign segment points in any given segment. Consequently, it is feasible to exhaustively consider each of them. The program considers splitting the segment with each of the curvature-sign segments points, one at time. The best choice is the one in which the sum of the errors of fit for the two new segments is minimum. If this minimum is less than 65% of the original error of fit, the new segmentation is retained, otherwise it is rejected. This threshold is designed to require significant improvement in the fit before a new segment point is added.

Figure 4 shows an example of how curvature-sign points are used. In the initial segmentation, curvature sign points are excluded, and the stroke is incorrectly segmented into a single arc segment. Because the fit is poor, the program tests both curvature-sign points in the middle of the curve and finds that an improved segmentation can be achieved. The result is shown in Figure 4b.

We have found it useful to apply our merging and splitting routines repeatedly. The special merging routine that handles noise at the start and end of each stroke is applied first. Next, the general routines for merging segments are applied, followed by two applications of the splitting routine. It is possible that splitting may produce segments that should be merged with their neighbors. Thus, the final step consists of an additional application of the general merging routines.

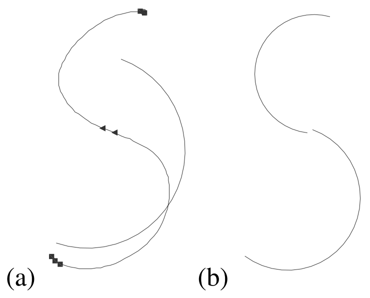

Figure 4: (a) Candidate segment points for an "s-curve." Segmentation is a single arc if curvature sign points (shown as small triangles) are ignored. (b) Final segmentation when curvature sign points are considered.

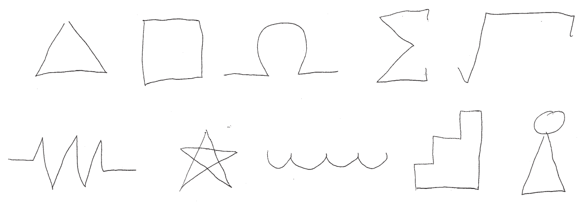

Figure 5: The ten shapes used in the user studies.

User Studies

To test our segmenter, we conducted two user studies in which multiple subjects were asked to draw the set of shapes shown in Figure 5. The subjects were instructed to draw accurately but naturally, and were informed that the experiment was intended to evaluate the accuracy of our segmenter. We specifically selected subjects who had no previous experience with our system, but who did have at least some experience using a PDA or digitizing tablet. Subjects were given only a minute or two to become familiar with the system before providing samples for the study. Thus, our results reveal how well our software performs for a new user. We have found that after one has gained moderate experience with our system, one is able to achieve even higher accuracy than demonstrated in these studies.

For both user studies, we used an Intuos2 digitizing tablet with an inking stylus. This stylus leaves physical ink on a piece of paper placed over the tablet. The computer display showed the raw ink rather than the segmented ink, as we did not want the subject to alter his or her drawing based on the program's performance. In fact, the subjects were given no feedback at all about how well the program performed.

The first user study evaluated the suitability of our speed threshold for the typical user. For this study, the digitizing tablet was set to a resolution of 1024x768 (low resolution mode). Five subjects were asked to draw the ten symbols in Figure 5 four times each. All were asked to draw the shapes at a size of approximately 3cm, which is a comfortable size when viewing the ink on the computer display. (The second user study, described below, explored accuracy as a function of symbol size.)

Table 1 shows the results of the first study. The performance of the system was evaluated in terms the number of missing and extra segment points. Missing points can occur for one of three reasons: (1) no candidate segment point was found, (2) a candidate was found but was later eliminated by merging of the two adjacent segments, or (3) a candidate was found but was later eliminated during the clean up of the start or end of the pen stroke. Extra points are those that were not intended as segment points, but were labeled as such by the program.

When evaluating the accuracy of the computed segmentation, we accounted for variations in the way each subject drew the shapes. For example, the number of intended "wiggles" in the spring-like symbol varied from one subject to

	(a) Threshold = 25%						(b) Threshold = 100%					
User	1	2	3	4	5	**Ave**	1	2	3	4	5	**Ave**
Number of Seg Points	230	229	244	229	233	**233**	230	229	244	229	233	**233**
Missing Seg Points	11	5	17	0	2	**7.0**	0	0	0	0	0	**0.0**
Mistakenly Merged	2	1	2	0	0	**1.0**	2	0	1	0	0	**0.6**
Missing Start/End	0	1	3	0	0	**0.8**	0	1	4	0	0	**1.0**
Extra Seg Points	0	3	0	0	2	**1.0**	12	13	3	1	5	**6.8**
Correct Seg Points, %	94.3	95.6	91.0	100.0	98.3	**95.8**	93.9	93.9	96.7	99.6	97.9	**96.4**
Correct Symbols, %	82.5	82.5	67.5	100.0	90.0	**84.5**	72.5	75.0	87.5	97.5	90.0	**84.5**

Table 1: User study with speed thresholds of (a) 25% and (b) 100%.

the next. Table 1 tabulates the number of intended segment points for each subject, which was typically about 230. The segmentation error for each subject is defined as the sum of the missing and extra segment points divided by the total number of intended segment points. The segmentation accuracy is defined as one minus this value.

Table 1(a) shows the results obtained with a speed threshold of 25%. The average accuracy across all five subjects was 95.8%. Additionally, on average, 84.5% of the symbols had no segmentation errors of any kind. The symbols in this study contained an average of 6 segment points, thus there are multiple ways for there to be an error in a given symbol. This is why this second measure of accuracy is lower than the first.

Most of the segmentation errors occurred because no candidate segment point was identified. On average, there were 7 such errors for each set of 40 symbols. (Note, again, that each set of 40 symbols contained about 230 segment points.) Significantly fewer points were missed because of segment merging or start/end cleaning – there was approximately one of each of these errors for each set of 40 symbols. We did notice, however, that some subjects drew the square root and summation symbols with very small serifs, which were incorrectly eliminated as start/end noise. (Some subject drew large serifs, while other did not draw them at all.)

To evaluate how sensitive our approach is to the speed threshold, we resegmented the ink using a larger threshold of 100% of the average speed (Table 1(b)). With a threshold of 25%, there was on average 8.8 missing segment points and 1 extra segment point for each set of 40 examples. With the higher threshold, there was on average 1.6 missing segment points and 6.8 extra ones. As one would expect, as the threshold increases, the number of missing points decreases and the number of extras increases.

For four of the subjects, accuracy decreased only a little with the increased threshold. This suggests that the accuracy is not overly sensitive to the threshold. For the third subject, however, there was a significant increase in accuracy with the larger threshold. Later discussions with that subject revealed that he was a trained calligrapher and was skilled at maintaining a constant pen speed so as to avoid ink blotches.

The second user study (Table 2) was intended to evaluate the accuracy of the system for various sizes of the ten shapes in Figure 5. This study also employed five participants, only one of which (subject 1) had participated in the first study. Each subject was asked to draw each of the ten symbols at sizes of 1cm, 2cm, and 4cm. Because small symbols were used in this test, it was necessary to operate the tablet at a higher resolution setting of 2048x1536. To obtain a basis for comparison with the first user study, each subject was also asked to draw the symbols at 4cm using the low resolution setting (1024x768). Overall, we found that the there was only a small decrease in accuracy for the smaller sized shapes. Similarly, on average, the accuracy for large symbols with the high resolution mode was the same as with the low resolution mode.

Symbol Size	1cm	2cm	4cm	4cm
Tablet Resolution	High	High	High	Low
Num Seg Pts, Ave	58.0	57.8	58.4	57.0
Missing Seg Pts, Ave	2.0	0.2	0.0	0.4
Mistaken Merge, Ave	0.2	0.2	0.4	0.0
Missing Start/End, Ave	1.2	0.4	0.2	1.0
Extra Seg Pts, Ave	1.0	2.0	2.0	1.0
Correct Seg Pts, Ave, %	92.4	95.1	95.6	95.8
Correct Sym's, Ave, %.	68.0	76.0	80.0	80.0

Table 2: Size study. Speed threshold 85%.

Discussion and Future Work

Our goal is not to match the ink precisely, it is to match the drawer's intent. We believe that this necessitates the use of predefined parameter values, such as speed and curvature thresholds, in our program. These parameters are, in essence, a model of what a person would perceive as important in a hand-drawn sketch. We do not believe that the intrinsic properties of a curve alone, are adequate to indicate the drawer's intent. Empirically determined parameter values are essential. We take an engineering perspective on this issue: if our program can produce the outcome the user expects, then we have achieved our goals.

The parameter values used in our program are matched to the drawing hardware we use. They are designed to work with drawing hardware that is about the size of a standard sheet of letter paper. Applying our techniques to a large device, such as an electronic whiteboard, or a smaller device, such as a PDA, would require tuning of the parameters.

We have found that our system is not very sensitive to the speed threshold used. This comes from several factors. When the threshold is too small, segment points are missed. However, such points are typically identified as curvature-based segment points. These points effectively have a higher speed threshold, but use curvature information to ensure accuracy. Conversely, when the threshold is too large, the extra segment points are eliminated by merging.

One purpose of our segmenter is to support symbol and shape recognizers. For this purpose, it is essential that we consistently identify just the intended segment points. Our user studies have indicated that our program can achieve an accuracy of around 96%. We have found that this level of performance is adequate for reliable recognition. Interestingly, recognition may be able to provide feedback to help improve segmentation. Once a symbol has been recognized, knowledge of its typical geometry and topology can be used to correct segmentation errors.

Our user studies show that our program works well for typical users, even those who have had little experience with our system. In future work, however, we would like to develop training techniques to customize our system for each user. One approach would be to have the user provide a set of standard training examples, and the program could optimize its parameters to maximize segmentation accuracy. Alternatively, the system could be adaptive and incrementally improve its performance as the user corrects segmentation errors during ordinary use.

Conclusion

The challenge in segmenting a pen stroke is to identify the geometric primitives intended by the drawer. Often, the intent is not a literal interpretation of the stroke. Consequently, a segmentation technique driven by the objective of matching the ink is likely to produce poor results. Rather, our approach uses pen speed information to help infer intent. We have observed that it is common for the drawer to slow the pen tip at points of intended discontinuities in a pen stroke.

Based on this insight, we have developed a technique for segmenting hand-drawn pen strokes into lines and arcs. To begin the segmentation process, an initial set of candidate segment points is identified. This set includes speed minima below a threshold, where the threshold is computed from the average pen speed along the pen stroke. It also includes curvature maxima at which the pen speed is again below a threshold. Once the initial set of candidates has been generated, the ink between each pair of consecutive segment points is classified as either a line or an arc. A feedback process is then employed, and segments are judiciously merged and split as necessary to improve the quality of the segmentation.

Our system does employ empirically determined constants. They are tuned to the particular hardware we use, and will likely need adjustment for optimal performance on other hardware. We have found that these constants are suitable for most users who have tested our system.

User studies of our segmenter indicate that it performs well, even for the new user. In these studies, our segmenter had an accuracy of between 92% and 96%, depending on the task. One interesting observation from these studies is that the accuracy is surprisingly insensitive to the speed threshold used for identifying segment points.

In summary, this work has demonstrated the utility of pen speed for inferring the intended segmentation of pen strokes. It also demonstrates that feedback can be used to significantly improve the initial segmentation. There are clearly still opportunities to improve the performance of our segmentation approach. However, our user studies suggest that our approach is sufficiently reliably for implementing usable systems.

References

Agar, P., and Novins, K. 2003. Polygon recognition in sketch-based interfaces with immediate and continuous feedback. In *Proc. 1st international conf. on comp. graphics and interactive techniques in Austalasia and South East Asia*, 147–150.

Calhoun, C.; Stahovich, T. F.; Kurtoglu, T.; and Kara, L. B. 2002. Recognizing multi-stroke symbols. In *AAAI '02 Spring Symposium, Sketch Understanding*.

Dudek, G., and Tsotsos, J. 1997. Shape representation and recognition from multiscale curvature. *CVIU* 68(2):170–189.

Gennari, L. M.; Kara, L. B.; and Stahovich, T. F. 2004. Combining geometry and domain knowledge to interpret hand-drawn diagrams. In *AAAI '04 Fall Symp., Making Pen-Based Interaction Intelligent & Natural*.

Herot, C. F. 1976. Graphical input through machine recognition of sketches. In *Proc. 3rd annual conf. on comp. graphics and interactive techniques*, 97–102.

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

Kara, L. B.; Gennari, L. M.; and Stahovich, T. F. 2004. A sketch-based interface for the design and analysis of simple vibratory mechanical systems. In *2004 ASME Design Engineering Technical Conf., DETC'04*.

Landay, J. A., and Myers, B. A. 2001. Sketching interfaces: Toward more human interface design. *IEEE Computer* 34(3):56–64.

Lee, J.-S.; Sun, Y.-N.; and Chen, C.-H. 1995. Multiscale corner detection by using wavelet transform. *IEEE Trans. on Image Processing* 4(1):100–104.

Mortenson, M. E. 1985. *Geometric modeling*. John Wiley & Sons, Inc.

Rattarangsi, A., and Chin, R. T. 1992. Scale-based detection of corners of planar curves. *IEEE PAMI* 14(4):430–339.

Rubine, D. 1991. Specifying gestures by example. *Computer Graphics* 25:329–337.

Sezgin, T.; Stahovich, T.; and Davis, R. 2001. Sketch based interfaces: Early processing for sketch understanding. In *Perceptive UI Workshop, PUI'01*.

Sezgin, T. M. 2001. Feature point detection and curve approximation for early processing of free-hand sketches. Master's thesis, MIT.

Shpitalni, M., and Lipson, H. 1996. Classification of sketch strokes and corner detection using conic sections and adaptive clustering. *ASME J. of Mechanical Design* 119(2):131–135.

Teh, C. H., and Chin, R. T. 1989. On the detection of dominant points on digital curves. *IEEE PAMI* 11(8):859–872.

Yu, B. 2003. Recognition of freehand sketches using mean shift. In *Proc. International Conf. on Intelligent User Interfaces, IUI'03*.