EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling (2009)
C. Grimm and J. J. LaViola Jr. (Editors)

Sort, Merge, Repeat: An Algorithm for Effectively Finding Corners in Hand-sketched Strokes

A. Wolin, B. Paulson, and T. Hammond[1]

[1]Sketch Recognition Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
{awolin, bpaulson, hammond}@cse.tamu.edu

Abstract

Free-sketch recognition systems attempt to recognize freely-drawn sketches without placing stylistic constraints on the users. Such systems often recognize shapes by using geometric primitives that describe the shape's appearance rather than how it was drawn. A free-sketch recognition system necessarily allows users to draw several primitives using a single stroke. Corner finding, or vertex detection, is used to segment these strokes into their underlying primitives (lines and arcs), which in turn can be passed to the geometric recognizers. In this paper, we present a new multi-pass corner finding algorithm called MergeCF that is based on continually merging smaller stroke segments with similar, larger stroke segments in order to eliminate false positive corners. We compare MergeCF to two benchmark corner finders with substantial improvements in corner detection.

Categories and Subject Descriptors (according to ACM CCS): I.4.6 [Computing Methodologies]: Image Processing and Computer Vision—Segmentation - Edge and feature detection

1. Introduction

Sketch recognition is an emerging field that utilizes pen-based interfaces in an attempt to make human-computer interaction as natural as human-human interaction. In these interfaces, electronic styli replace traditional mice and keyboards and allow users to draw onto a digital screen as if it were intelligent paper.

Free-sketch recognition (or natural sketch recognition) allows users to draw without developer-placed constraints on the drawing style. Such constraints could force a user to draw primitives in separate strokes, draw strokes in a certain order, or learn a set of prespecified gestures (e.g., [Rub91, ACLLRM00]). Researchers have built free-sketch recognition systems in domains such as circuit diagrams [AD04], figure recognition [SvdP06], and UML diagrams [HD02] (Figure 1).

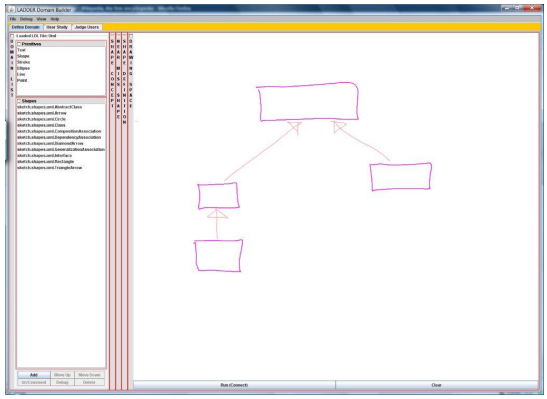


Figure 1: *A free-sketch recognition system for UML diagrams, implemented in LADDER.*

Some important free-sketch recognition systems that avoid constraining the user are geometric recognizers, which define shapes by sets of primitives and geometric rules [HD07] or graphical models [CD04]. A geometric recognition system works by first breaking down drawn strokes into primitive shapes, and then classifies the group of

Sketch-Based Interfaces and Modeling 2009, New Orleans, LA, August 1–2, 2009.

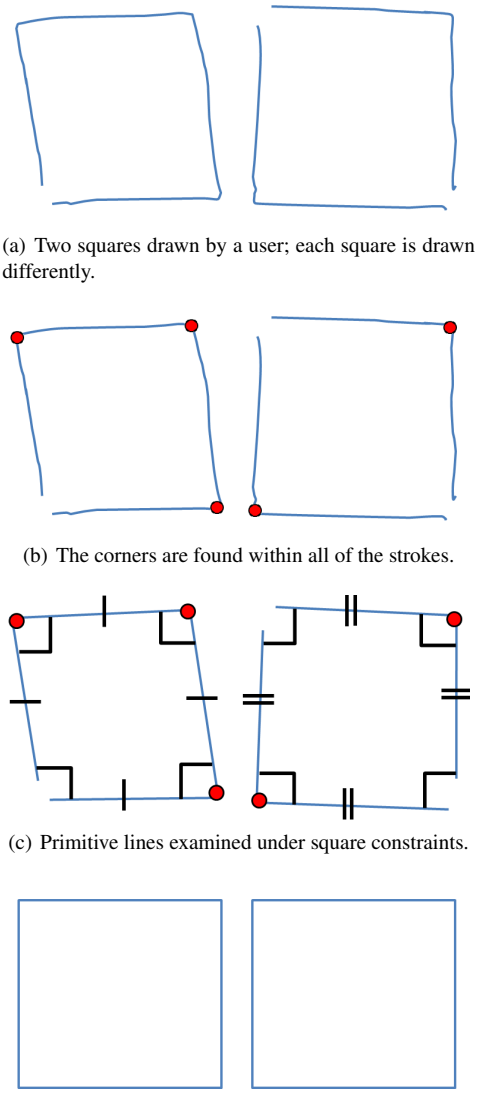
primitives using geometric constraints. This process avoids many drawing style issues since all symbols are broken down into a single, basic form.

Strokes can be classified as primitive shapes using recognizers such as Sezgin's [SSD01] and Paulson's [PH08]. These classifiers require strokes composed of multiple primitives to be split using corner finders. In a corner detection system, algorithms automatically break a user's drawn stroke into primitive lines and arcs. This task can be completed during stroke preprocessing, and the resulting primitives can then be sent to a geometric recognizer for stroke classification.

As a geometric recognizer example, suppose a user draws the two symbols in Figure 2(a). Both symbols are squares, yet they are drawn using a different number of strokes. Finding the corners of each stroke (Figure 2(b)) allows the recognizer to describe each symbol as consisting of four lines; the more complex descriptions are avoided. Geometric rules can then analyze the four lines and find that certain lines are perpendicular and of equal length (Figure 2(c)), and the geometric recognizer then classifies each symbol as a square based on the primitives within the symbol and the geometric constraints satisfied.

This process demonstrates the necessity for reliable corner finding techniques in free-sketch, geometric recognition systems. Corner finding is the first step in geometric recognition, and, if corner finding is inaccurate, then the entire recognition process is error prone and the resulting classification is unreliable.

In this paper we present MergeCF, a multipass corner detection algorithm that utilizes the stroke's curvature and the user's drawing pen speed in order to find the corners of a stroke. MergeCF then eliminates false positives by merging stroke segments together based on inherent false positive properties that help distinguish between correct and unnecessary corners. This powerful corner finder improves upon current state-of-the-art techniques using two different accuracy measures.



(a) Two squares drawn by a user; each square is drawn differently.

(b) The corners are found within all of the strokes.

(c) Primitive lines examined under square constraints.

(d) Two recognized squares.

Figure 2: *The geometric recognizer process from drawn strokes 2(a) to the recognized squares 2(d)*

2. Previous Work

Sezgin *et al.* use a stroke's curvature and pen speed to determine stroke corners [SSD01]. In their system, points of high curvature are considered corner candidates, as well as points of low speed. After the authors obtain an initial collection of curvature and speed corners, their system picks either the "best" curvature or speed corner, one at a time, and creates a new corner fit for the stroke using the picked corner and the previous corner fit. The best corner is determined by defined metrics. This process of adding the best curvature or speed corner candidate to create a new fit is continued, and then a final corner fit is chosen as the fit with the least amount of corners and an error below some threshold.

Sezgin *et al.*'s algorithm relies on the assumption that the correct corners to add to the system will always be ranked highest according to their metrics. Yet, if the first few corners added to the system are false positives, then any fit after these corners are added will also contain these false positives. Our corner finder does not make this assumption, and instead of building a corner set one corner at a time, we start with a large set of possible corners and eliminate false positives.

Stahovich also used pen speed and curvature to detect corners [Sta04]. Unlike [SSD01], Stahovich relies more heavily on chosen thresholds for average speed and curvature. Like our algorithm, Stahovich's algorithm does not build a

fit but seeks to remove false positives by merging segments together. Our algorithm uses a more lenient approach to thresholding and a more rigorous segment merging, whereas Stahovich's algorithm does the opposite.

Yu and Cai created a corner finder that uses only direction and curvature information to find the corners of a stroke [YC03]. Their system introduces the idea of feature area, or the area of a drawn stroke segment in relation to a beautified version of the same segment. Yu and Cai's corner finder performs stroke beautification; our algorithm does not. We partially use Yu and Cai's technique for finding the error of arcs through our use of the primitive recognizer, PaleoSketch [PH08].

Kim and Kim propose new curvature metrics in their corner finding system [KK06]. These metrics, local convexity and local monoticity, measure the curvature in the same direction at a point. Convexity adds together all of the curvatures of the same sign within a window, whereas local monotocity looks at decreasing curvatures of the same sign around a point. Kim and Kim also have a different measure for the curvature at a point. Their system first resamples the points of a stroke to be equidistant from one another. Since the distance between consecutive points is now constant, a point's curvature value does not have to take into account path length changes, so the curvature at each point is equal to the direction change at that point. Our algorithm does not use any techniques defined by Kim and Kim, but we use an implementation of their corner finder for reference during our analysis.

The corner finders previously mentioned all require developer set thresholds for different properties, such as curvature and speed thresholds or the interspacing distances for resampled points. Other corner finders avoid relying on hard-coded variables. Bandera *et al.* use a multi-pass algorithm to detect the curvature, or contour, scale for strokes of various sizes [ABS00]. Another Sezgin corner finder relies on finding the optimal scale for a stroke [SD04]. This technique increasingly applies Gaussian filters to curvature data, and, as the filters smooth the data, the number of detected corners drops. The optimal scale is determined to be where the number of corners reduced by increasing the smoothing factor tapers off.

ShortStraw is a recent corner finder developed for segmenting polyline strokes [WEH08]. Although ShortStraw is the state of the art for polylines, it cannot segment arcs and, thus, we avoid using it in our comparisons.

Our merging algorithm is similar to a merging technique that is alluded to in [PH08]. The corner finder technique mentioned in this paper is focused on finding corners within polylines, whereas our algorithm extends the idea to work with complex fits consisting of lines and arcs. Paulson and Hammond also defined different error fit measurements for primitives such as lines, arcs, curves, circles, helixes [PH08]. We use the primitive fit definitions they provide to calculate our line and arc fit errors.

3. Implementation

Our corner finder utilizes curvature and speed differences within a stroke to obtain an initial corner segmentation for our stroke. We then repeatedly merge smaller stroke segments with longer segments, and, if the fit for the merged segment is below a certain threshold, we eliminate the corner between the two segments.

3.1. Curvature and Speed Values

Our curvature and speed values are based on the equations given by [Sta04] and [YC03]. The distance between two points is the euclidean distance between the points, and the path length across a series of points $p_a, p_{a+1}, \ldots, p_b$ is taken to be the sum of the euclidean distances between each pair of points.

$$pathLength(a,b) = \sum_{i=a}^{b-1} \sqrt{(x_{i+1}-x_i)^2 + (y_{i+1}-y_i)^2} \quad (1)$$

Where x_i and y_i are the respective x and y values of the point, p_i. Curvature values for a point at stroke index i are taken to be the change in angle across a window of points, divided by the path length across the window.

$$c_i = \frac{\left|\sum_{i=a-k}^{a+k} atan2(\delta y_i/\delta x_i)\right|}{pathLength(a-k,a+k)} \quad (2)$$

Speed at a point, p_i, is calculated as the path length change across the point, divided by the time difference

$$s_i = \frac{pathLength(i-1,i+1)}{t_{i+1}-t_{i-1}} \quad (3)$$

3.2. Initial Fit

After we compute the curvature and speed values for each point, we find our initial set of corners by taking points that are local maxima (for curvature) and local minima (for speed), with respect to set curvature and speed thresholds. In this implementation, these thresholds were set to find points above the average curvature and below 75% of the average speed.

These curvature and speed corners are found separately and then combined into one set of corners. We then iterate through our new corner set and remove any points that do not fit *both* the curvature and speed requirements. This idea stems from [Sta04], but we use more lenient thresholds in order to accept more points.

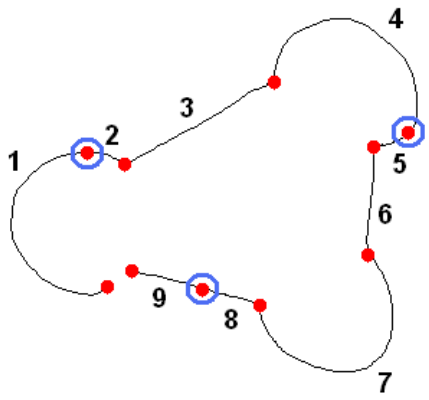


Figure 3: *Initial set of corners found for a stroke, which would split the stroke into 9 primitive lines and arcs. False positives are circled.*

We also check for points that are close together in proximity. If two corners are less than 15 pixels euclidean distance apart, then we remove the corner with the smallest curvature from the initial fit.

3.3. Merging Segments

Our initial corner fit tends to contain a few extraneous points that overfit the stroke. The main algorithm involved with our corner finding system is designed to eliminate these false positives, and MergeCF works on the assumption that corners surrounding the smallest segments (parts of the stroke between two corners) are those more likely to be false positives overfitting the data.

The algorithm first finds the smallest stroke segment, checks if the segment can be merged with any of its neighbors, and then merges the segment with the "best" neighboring segment. The best segment is determined to be the segment that has the least primitive fit error (either line or arc) when combining the two segments.

The fit error calculations we use come from [PH08]. The two primitives our system handles are lines and arcs, and Paulson's recognizer handles line and arc errors by calculating the least squares error and the feature area error [YC03] for each segment. Lines additionally have to pass a ratio test, where the test takes the euclidean distance length between the two points and divides the value by the path length between the points. If this ratio is greater than a set threshold, then the segment is a line. In our algorithm, the threshold is equal to 0.95. Arcs have to pass another test as well, which is a feature-based test using the NDDE and DCR metrics defined in [PH08]. These metrics are tuned to distinguish between polylines and arcs.

As an example, Figure 3 shows a symbol with an initial corner fit containing three false positives (the circled points) and numbered stroke segments. Merging segment 5 with segment 4 would still result in an arc fit error that is not too much higher than the either segment 4 or 5's original error. Yet, merging segment 5 with segment 6 would produce a very high primitive error for either lines or arcs. Therefore, the best segments to merge are 4 and 5, and the circled point in between the two segments is removed from the corner set.

3.4. Algorithm

A more formal definition of our algorithm is as follows:

1. Calculate the path length of each segment.
2. Calculate the average path length of the segments.
3. Sort the segments from step 1 in ascending order, based on the path lengths.
4. For each segment shorter than the average segment:
   a. Calculate the primitive fit error of the segment, $FitError_s$, and the fit error of the segment to the left and to the right of the short segment, if there are any. These are $FitError_{s-1}$ and $FitError_{s+1}$, respectively.
   b. Calculate the primitive fit error of the joined segments $s-1$ and s, which we will call $FitError_{left}$. Also, calculate $FitError_{right}$ from s and $s+1$.
   c. If $FitError_{left} < FitError_{right}$ and $FitError_{left} < 1.5 * FitError_{s-1} + FitError_s$, then remove the corner between $s-1$ and s. Otherwise, preform similar checks for the right side of the segment.
5. Repeat steps 1-4, but, after each run, multiply the average segment by the number of runs. This steadily increases our "segment shorter than" threshold. Stop repeating once every segment is shorter than this threshold.

MergeCF's algorithm underperforms when trying to merge two line segments. The primitive fit error for two individual line segments tends to be much lower than the fit error for the joined segments. After running the above algorithm, we iterate through our remaining segments and check specifically for two consecutive lines. If both lines have similar slopes, we merge the two segments together by eliminating the corner between them.

3.5. Intuition

Our algorithm to merge smaller segments works because of the inherent way that corners are initially detected. Complex and polyline stroke symbols tend not to have segments that have extreme variance in length. This is due to the inherent problems with "hook" detection where very small stroke segments attached to much longer segments are typically noisy data (hooks) that should be removed [Sta04]. To avoid having issues with overzealous hook removal, stroke segments within symbols tend to be on the same scale.

If an initial corner fit contains few false negatives (i.e.,

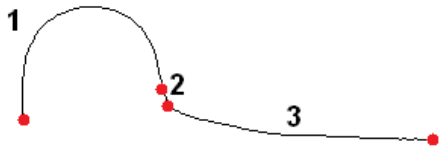


Figure 4: *Initial set of corners found for a stroke consisting of an arc and a line. Segment 2 is the smallest, unneeded segment and should be merged with Segment 1.*

missing corners), then the majority of the corners found can be assumed to be false positives. Now, if we assume that all of the stroke segments are drawn at the same scale, then any false positives would split a stroke segment of average length into smaller pieces. Therefore, the merging algorithm should start by examining the smallest stroke segments since they are most likely to contain false positives as end points.

Another reason why we want to merge smaller segments first is due to the way fit errors are calculated. Suppose a stroke consisting of an arc and a line has the initial fit shown in Figure 4. If the algorithm started by merging the largest segments first, then segment 2 would be merged with Segment 3 since the error calculation for the line consisting of 2 and 3 is not substantially different than the line error for Segment 3 alone. In fact, Segment 2 can be considered a hook of Segment 3 since it is substantially smaller. A much better option would be to merge Segment 1 and 2 together to form a slightly larger arc. To avoid the problem of merging Segment 2 and 3 we let the smallest segments decide their best merging options.

Continually increasing the threshold that determines which segments are small ensures that all stroke segments will eventually be evaluated.

4. Results

MergeCF was developed around a training data set consisting of 157 unistroke shapes drawn collected from five users. The algorithm was tested on a different set of data based on the symbols found in [KK06]. This test set consisted of 501 complex shapes and polylines, each drawn with a single stroke. During the collection of both data sets, users were asked to sketch a given shape with easily defined corners. Example shapes can be seen in Figure 7.

Results were gathered on two other corner finders as well, Sezgin *et al.*'s algorithm [SSD01] and Kim and Kim's algorithm [KK06]. We implemented both algorithms as presented in their respective papers, and we tested all of the corner finders on the same data sets.

We used two different measures to determine the accuracy of a corner finder. The first accuracy measure is a "correct corners found" accuracy as presented by [SSD01]. This accuracy is calculated by dividing the number of correct corners found divided by the total number of correct corners perceived. Essentially, this accuracy measure does not discount false positives, and returning every point in a stroke would constitute a 1.00 accuracy since all of the correct corners have been found.

To counteract this issue, we also calculate an all-or-nothing accuracy for each corner finder. All-or-nothing implies that only the minimum number of corners to segment a figure are found (i.e., there are no false positives or negatives) in order for a stroke to be considered correctly segmented. This accuracy is calculated by taking the number of correctly segmented strokes divided by the total number of strokes.

The results in Table 1 show how our algorithm outperforms both the corner finding algorithms from Sezgin and Kim.

5. Discussion

Our corner finder significantly improves corner detection over the two benchmark systems in both accuracy measures. MergeCF finds less false positives and negatives than our Sezgin and Kim implementations, and the all-or-nothing accuracy is over twice that of the previous best corner finder's.

MergeCF performs better than the other corner finders for a few reasons. Sezgin *et al.*'s algorithm assumes that the best corners, or the correct ones, will always be ranked higher than any false positives. This assumption is often invalid on complex shapes where minor speed differences and line noise can greatly affect the the author's corner metrics. Noisy arcs are the main culprit in this issue and produce many false positives along subtle bumps or peaks in the arc. Also, since Sezgin *et al.*'s algorithm chooses the fit with the least number of corners below a certain threshold, it is often the case where correct corners are missing from the final segmentation if the threshold is too high for a shape. If the majority of the corner fits are below the threshold, then the corner fit with the least number of corners can be a poor choice (See Figure 5).

Kim and Kim's corner finding algorithm produces many false positives, mainly due to sensitive thresholds present in

	MergeCF	Sezgin	Kim
False Positives	159	173	188
False Negatives	98	608	741
Correct Corners Found	3314	2804	2671
Total Correct Corners	3412	3412	3412
Correct Corners Accuracy	0.971	0.822	0.783
All-or-Nothing Accuracy	0.667	0.298	0.194

Table 1: *Results from the individual corner finding algorithms.*

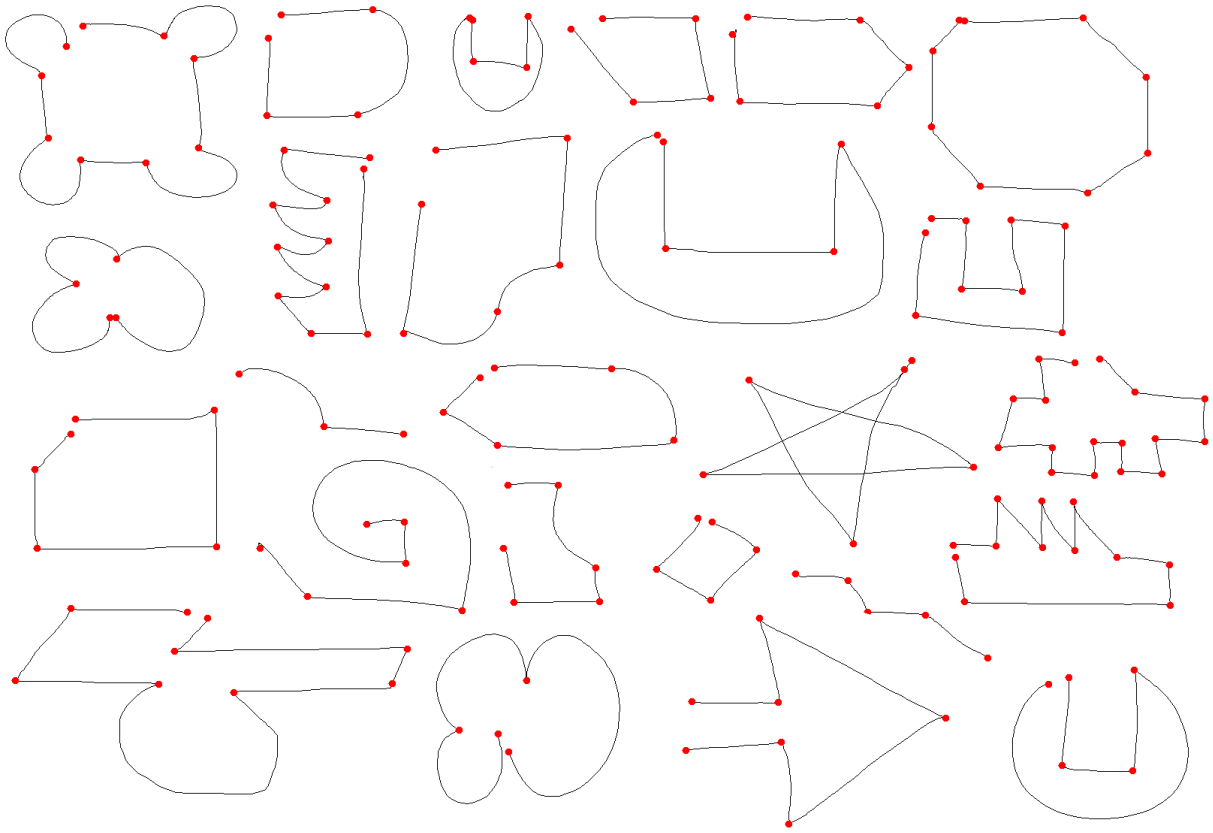


Figure 7: *Examples of correctly classified symbols by MergeCF. These symbols come from the set of 501 complex and polyline shapes drawn by six users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper.*

their system. Their algorithm oversmooths the data by using resampled points as well as smoothing curvature metrics, and when the data is too smooth, points with a high curvature have only slightly higher curvature values than points with average curvature.

Figure 6 illustrates the issues just mentioned. The first corner fit (A) is the result we obtain using our algorithm, and it is the correct fit for the shape. The second corner fit (B) is from Sezgin *et al.*'s algorithm. In this fit, the two false positives are ranked highly in their curvature metrics, and they are added to the final corner fit early. Sezgin *et al.*'s algorithm does not allow for removal of corners after they have been added, so the final fit contains these two. In the third corner fit shown (C), Kim and Kim's algorithm finds all of the correct corners, but it also finds 3 false positives. The curvatures at these points are high enough for those points to be considered corners under our Kim and Kim implementation.

MergeCF avoids the issues from Sezgin and Kim's algorithms by

- Having an initial fit with few false negatives
- Evaluating individual corners and segments at a local level
- Using inherent properties of false positives to examine short segments first
- Performing multiple passes through the segments to ensure that each segment is eventually evaluated and merged if necessary.

6. Future Work

Improving our complex fit detection, especially with arcs, is our main goal for the future. Merging two smaller arcs together can be difficult since arcs are classified as sections of circles and the error associated with arcs tends to be high. Appending two slightly offset arcs often produces a shape that has a considerably higher error than either of the individual arcs. One possible solution would be to allow MergeCF to use a new primitive: curve. A Bezier curve could be created to approximate two adjacent arc segments, and if the curve fits the segments well, then the corner between the

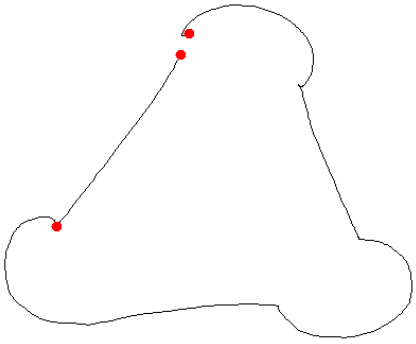


Figure 5: *An incorrect corner fit found by Sezgin* et al.*'s system. The error threshold for a "good" fit for this shape is set to be substantially high by the algorithm, so any slight improvements to the initial fit drop the total fit's error to be below the threshold. Since adding one corner greatly improves the fit, and since three corners is the smallest number of corners in all of the fits below the threshold, the algorithm incorrectly chooses the three-corner fit to be the best.*

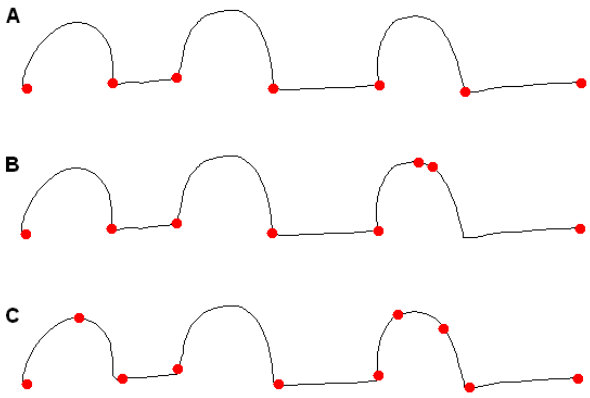


Figure 6: *The same shape with corner fits from our algorithm (A), Sezgin's algorithm (B), and Kim's algorithm (C). Our algorithm correctly finds the corners, whereas the other two corner finders have false positives and negatives.*

segments would be eliminated. The primitive recognizer we use from [PH08] already has a definition for curves, and our main merging algorithm would have to be tweaked to handle the error associated with these primitives.

7. Conclusion

We have presented a new corner finding technique that utilizes curvature and pen-speed values of a stroke to obtain an initial corner fit. After a fit is found, we eliminate false positives by examining small stroke segments and merging the segments with similar neighbors. Overall, our system greatly improves upon the existing benchmarks with room for further improvement for complex fits.

References

[ABS00] A. BANDERA C. URDIALES F. A., SANDOVAL F.: Corner detection by means of an adaptively estimated curvature function. In *Electronics Letters, Vol. 36, No. 2* (2000), pp. 124–126.

[ACLLRM00] A. CHRIS LONG J., LANDAY J. A., ROWE L. A., MICHIELS J.: Visual similarity of pen gestures. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2000), ACM Press, pp. 360–367.

[AD04] ALVARADO C., DAVIS R.: Sketchread: a multi-domain sketch recognition engine. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology* (New York, NY, USA, 2004), ACM Press, pp. 23–32.

[CD04] CATES S., DAVIS R.: New approach to early sketch processing. In *Making Pen-Based Interaction Intelligent and Natural* (Menlo Park, California, October 21-24 2004), AAAI Press, pp. 29–34.

[HD02] HAMMOND T., DAVIS R.: Tahuti: A geometrical sketch recognition system for uml class diagrams. *Papers from the 2002 AAAI Spring Symposium on Sketch Understanding* (March 25-27 2002), 59–68.

[HD07] HAMMOND T., DAVIS R.: Ladder, a sketching language for user interface developers. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, p. 35.

[KK06] KIM D., KIM M.-J.: A curvature estimation for pen input segmentation in sketch-based modeling. In *Computer-Aided Design* (2006), pp. 238–248.

[PH08] PAULSON B., HAMMOND T.: Paleosketch: Accurate primitive sketch recognition and beautification. In *IUI (Intelligent User Interfaces)* (2008).

[Rub91] RUBINE D.: Specifying gestures by example. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1991), ACM Press, pp. 329–337.

[SD04] SEZGIN T. M., DAVIS R.: Scale-space based feature point detection for digital ink. In *Making Pen-Based Interaction Intelligent and Natural* (Menlo Park, California, October 21-24 2004), AAAI Fall Symposium, pp. 145–151.

[SSD01] SEZGIN T. M., STAHOVICH T., DAVIS R.: Sketch based interfaces: Early processing for sketch understanding. *Workshop on Perceptive User Interfaces, Orlando FL* (2001).

[Sta04] STAHOVICH T. F.: Segmentation of pen strokes using pen speed. In *AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural* (2004).

[SvdP06] SHARON D., VAN DE PANNE. M.: Constellation models for sketch recognition. In *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling* (2006).

[WEH08] WOLIN A., EOFF B., HAMMOND T.: Shortstraw: A simple and effective corner finder for polylines. In *Eurographics 2008 - Sketch-Based Interfaces and Modeling (SBIM)* (2008).

[YC03] YU B., CAI S.: A domain-independent system for sketch recognition. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2003), ACM Press, pp. 141–146.