EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2008)
C. Alvarado and M.- P. Cani (Editors)

ShortStraw: A Simple and Effective Corner Finder for Polylines

A. Wolin[†], B. Eoff[‡], and T. Hammond[§]

Texas A&M University
Dept. of Computer Science
College Station, TX 77843-3112

Abstract

In this paper we introduce ShortStraw, a simple and highly accurate polyline corner finder. ShortStraw uses a bottom-up approach to find corners by: (1) resampling the points of the stroke, (2) calculating the "straw" distance between the endpoints of a window around each resampled point, and (3) taking the points with the minimum straw distance to be corners. Using an all-or-nothing accuracy measure, ShortStraw achieves an accuracy more than twice that of the current best benchmark.

Categories and Subject Descriptors (according to ACM CCS): I.4.6 [Segmentation]: Edge and feature detection

1. Introduction

Sketch recognition involves understanding user-drawn strokes to allow for new human-computer interface techniques besides the standard mouse and keyboard. In an attempt to make human-computer interaction as natural as human-human interaction, we would like to build sketch systems that allow people to draw as they would naturally without placing drawing constraints on the user such as drawing strokes in a certain order, drawing each primitive in a separate stroke, or requiring the user to learn a set of prespecified gestures (e.g. [Rub91]). Researchers have begun to build free-sketch recognition systems in domains such as circuit diagrams [AD04] or UML diagrams [HD02].

A fundamental step in providing free-hand sketch recognition is allowing users to draw multiple primitives (such as a square drawn out of four lines) with a single stroke. Once a stroke is broken down into primitives, the primitives can be recognized with high accuracy [PH08], and then recombined using geometrical rules [AD04, HD05] to allow for recognition of naturally sketched shapes.

Corner finding is the technique that involves splitting up a stroke into primitives, such as lines and arcs. In polyline corner finders, such as the one presented in this paper, the corner finder finds the minimum set of points such that, if the polyline is split at those points, the resulting primitives would consist of only lines.

Other uses for polyline corner finding abound, including node traversal - identifying which nodes a user purposefully selected. For instance, ShapeWriter, previously known as SHARK [SZ05], allows a user to stroke out words on a virtual keyboard by drawing a stroke that connects each letter of the word in sequence; a corner finder could be used as a first pass to identify the letters (effectively, the corners) of the word.

Another node traversal application (and corner finder use) is an urban route planner and guidance system in conjunction with a mapping program. A user could use a stylus to trace a driving route onto a map. Since urban environments are typically set up in grids, the corners in the route would correspond to intersections and turns that the guidance system could watch out for and alert the driver to.

Polyline corner finding also has uses within 3D sketch systems. Many of these systems require users to draw straight lines in "slices" of Cartesian coordinates that allow users to overcome the issues involved with drawing within 3D environments. Other systems, such as [ML05],

[†] e-mail: awolin@cs.tamu.edu
[‡] email: bde@cs.tamu.edu
[§] e-mail: hammond@cs.tamu.edu

rely on primitive lines and planar curves to be drawn individually, and objects are then constructed by determining where lines meet in a three-dimensional space. Corner finding can assist these applications by allowing users to draw three-dimensional objects with single strokes.

As we can see, polyline corner finding is quite valuable for a variety of applications. Our goal is to provide an easy-to-program algorithm (such as [WWL07]) that outperforms all current polyline corner finding algorithms. A simple algorithm is valuable for educational purposes to help teach beginning students in a field and lead into more complicated techniques.

We present ShortStraw, an easily implemented algorithm for polyline corner finding that dramatically outperforms the state-of-the-art corner finding techniques for polylines. The remainder of this paper will discuss how our algorithm relates to previous work, our algorithm's implementation, and the results from comparing ShortStraw to current baseline corner finders.

2. Related 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.

Kim and Kim propose new curvature metrics in their corner finding system [KK06]. The 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 than in [SSD01]. In [KK06], the 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 arc length changes, so the curvature at each point is equal to the direction change at that point.

Both algorithms in [SSD01] and [KK06] rely on some techniques that novices to computer science might not have. Curvature for a point on a stroke can be found with derivatives or least-squares regression, both of which are found in more advanced math courses like calculus and statistics. Beginning programmers might not have taken these courses, which would hinder them from understanding how curvature is computed. Our algorithm is founded on a simple concept using only the length between two points, which requires a small amount of mathematical background in geometry. Also, like in [KK06], our algorithm resamples the points of a stroke.

Hershberger and Snoeyink, Yu and Cai, and Hse *et al.* all fit primitives to stroke segments in order to find the corners of a stroke for beautification purposes [HS92, YC03, HSN04]. Hershberger's algorithm in [HS92] is an extension of the Douglas-Peucker algorithm for line simplification presented in [DP73], which fits the "best" set of lines to a polyline stroke. This improved algorithm has been proven to run in $\Theta(n \log n)$ in the worst case, with the original algorithm's performance at $\Theta(n^2)$. Hse *et al.* fit both line segments and elliptical arcs to symbols using dynamic programming techniques [HSN04]. Although their algorithm is accurate and can handle more complex shapes than polylines, the algorithm's memory and run-time performance is rather poor compared to Douglas-Peucker's.

ShortStraw relies on a set threshold for the window (i.e. number of points) examined when determining if a certain point is a corner. In Teh and Chin's corner finder [TC89], they vary the window for each point examined during corner finding calculations. Although having a scaling window can increase the accuracy for finding points that are corners, ShortStraw was designed to be as simple as possible while still providing high polyline accuracy. Other research exclusively uses scaling techniques to locate corners. Rattarangsi and Chin smooth a stroke's x and y points with a varying Gaussian scale in order to eliminate noise and allow for easy corner detection [RC92], and Sezgin improved upon the performance of this algorithm in his implementation scaling curvature data [SD06].

3. Implementation

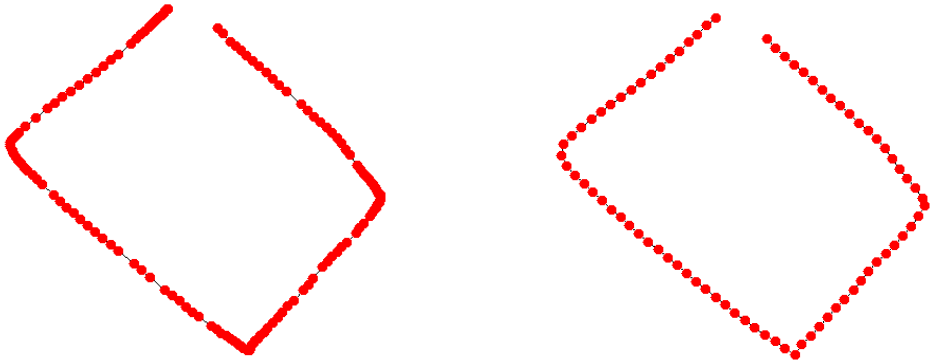
ShortStraw is designed to be simple to understand and easy to implement. As such, the entire algorithm can be discussed in detail in the paper, and pseudocode for the algorithm is also presented in the Appendix.

3.1. Resampling

The first step to ShortStraw involves resampling the points of a stroke to be evenly spaced apart. Resampling points is necessary in ShortStraw, for reasons that will be discussed in Section 3.2.1.

The algorithm for resampling is based on the algorithm presented in [WWL07]. Although the resampling remains the same, the interspacing distance of the points is determined differently.

In ShortStraw, points are resampled based on the diagonal length of the stroke's bounding box. The interspacing distance is equal to the diagonal divided by a constant factor. In



(a) Original points of the stroke (b) Resampled points of the stroke

Figure 1: *The original points (a) are varied in distance away from each other, whereas the resampled points (b) are interspaced evently..*

our implementation, this constant was set to 40; this constant factor was determined empirically. We found that increasing the value of this constant caused too much noise, whereas decreasing the constant created oversmoothed strokes. The interspacing distance is chosen this way in order to accommodate for strokes of varying size. Human perception of what constitutes a significant change in a symbol varies with stroke size [VD04].

The original points (called *points*) of the stroke can be resampled once we have calculated the interspacing distance, S. First, an empty set of points is created to store any new resampled points, and, for simplicity, this set is called *resampled*. The first point in the original point set, $points_0$, is then appended to *resampled*. A distance holder D is initialized to 0.

The main algorithm is as follows:

1. The Euclidean distance d between two consecutive points $points_{i-1}$ and $points_i$ is added to D.
2. If D is less than the interspacing distance S, then we increment i by 1 and repeat from step (1).
3. Otherwise:

   a. Create a new point q that is located approximately S euclidean distance away from the last resampled point. q_x and q_y are calculated to be $(S-D)/d$ distance between $points_{i-1}$ and $points_i$.
   b. Append q to *resampled*, and insert q before $points_i$.
   c. Repeat from step (1) without incrementing i.

The main algorithm loop terminates when $i > |points|$. The algorithms for both the interspaced distance calculation and the point resampling can be found in the Appendix. An example of a resampled stroke can be seen in Figure 1(b).

3.2. Corner Finding

ShortStraw finds corners using both a bottom-up and top-down approach. The bottom-up approach attempts to build corners from primitive information, whereas the top-down approach looks at higher-level patterns to determine possible insertion or deletion of corners.

3.2.1. Bottom-Up

ShortStraw finds corners in a stroke based on the length of the "straws". A straw for a point at p_i is computed as:

$$straw_i = |p_{i-W}, p_{i+W}| \quad (1)$$

where W is a constant window and $|p_{i-W}, p_{i+W}|$ is the Euclidean distance between the points p_{i-W} and p_{i+W}. As a stroke starts to bend at a corner, the straws of points will begin to shorten, and the local minimum straw at point index k is a likely corner.

To find the initial corner set, all the straws are first computed for points p_W to $p_{|points|-W}$. The median straw is then found and a threshold t is set to be equal to the $median \times 0.95$. For each $straw_k \in straws$, if $straw_k$ is a local minimum below the threshold t, then k is a corner. We set the window $W = 3$; this value was determined to be the most effective at helping locate correct corners. An example of finding corners from straws is seen in Figure 2.

From these equations, it follows that the straw length must remain relatively constant throughout the stroke in order for the correct corners to be found. Resampling the points of a stroke assures that our algorithm will have a static straw length for the majority of the stroke, whereas the straws of non-resampled points (such as in Figure 1(a)) would be highly variant.

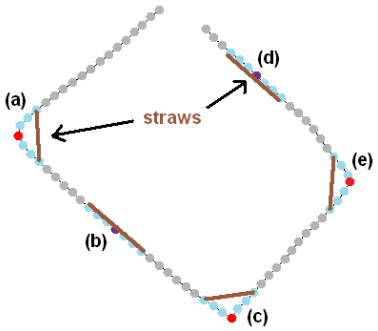


Figure 2: *An example of "straws" in a stroke. The points (a-e) all have a window of ±3 points. the distance at endpoints at these windows forms a straw, with the shortest straws being at points (a), (c), and (e). These points are considered corners. Points (b) and (d) have straws that are close to the median straw length, so these points are not initial corner candidates.*

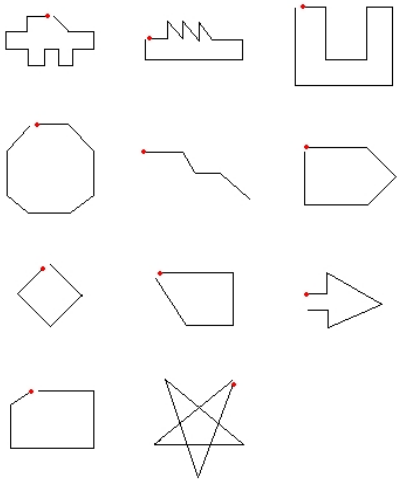


Figure 3: *The 11 symbols used during corner finder testing.*

3.2.2. Top-Down

After the initial set of corners is found by taking the shortest straws, some higher-level processing is run on the stroke to find missed corners and remove false positives.

ShortStraw first checks to see if each consecutive pair of corners passes a line test. Two points at indices a and b pass a line test if the chord distance and the path distance between the two points are relatively equal. We represent this equality through the ratio:

$$r = \frac{\text{DISTANCE}(points, a, b)}{\text{PATH-DISTANCE}(points, a, b)} \quad (2)$$

where $0.0 \leq r \leq 1.0$, since the squared distance between the two points will never be greater than the squared path distance. If the ratio in Equation 2 is above a developer-set threshold, then the segment between the points at a and b is considered to be a line. In our system, this threshold is set to 0.95 (See the Appendix the functions to compute DISTANCE, PATH-DISTANCE, and the IS-LINE test).

If the stroke segment between any two consecutive corners c_m and c_n does not form a line, then there must be additional corners in-between c_m and c_n. Missing corners are assumed to be approximately halfway between the c_m and c_n. Since these potential corners are below the original threshold t, the threshold is relaxed and the new corner to add is taken to be the point with the minimum straw that is in the middle half of the stroke segment. This process of adding corners is repeated until all of the stroke segments between pairs of consecutive corners are lines.

A collinear check is then run on subsets of triplet, consecutive corners. If the three corners are collinear, then the middle corner is removed from the corner set. This process checks and removes false positives. Three consecutive corners c_l, c_m, and c_n are deemed collinear if the stroke segment between c_l and c_n passes an IS-LINE test.

It is important to note that the final corners returned are from the resampled points. If a domain requires the original points of a stroke to be used, a developer implementing ShortStraw can map resampled corners to original points simply by taking each corner found and searching for the closest original point to that corner.

4. Results

We built ShortStraw around training data gathered from students. Five students sketched a total of 60 polylines, ranging from simple lines to more complex six-line symbols. This data set served as a training set as we constructed and configured ShortStraw.

To test ShortStraw, we collected another set of polyline data consisting of 11 shapes drawn by six different users. The shapes are the 11 found in Figure 3. A single set of these 11 symbols contains 37 right, 16 obtuse, and 12 acute angles. The users were presented with each shape, and over the course of the study each shape was drawn four times. 264 strokes were collected, but there was an error in collecting and saving some of the user data and 20 user strokes were not properly saved. As such, our final test set consisted of 244 polyline strokes.

For comparison, we also tested an implementation of Sezgin's corner finder as well as Kim and Kim's [SSD01, KK06]. We used two different measures to determine the accuracy of each corner finder. The first accuracy measure is a "correct corners found" accuracy that is described in

[SSD01]. This accuracy is calculated by dividing the number of correct corners found divided by the total number of correct corners a human would perceive. This accuracy measure does not discount false positives and only penalizes for false negatives. Therefore, a system that returns every point possible as a corner would achieve a perfect 1.00 accuracy since all of the correctly perceived corners would be found.

We use a different accuracy measure to counteract this issue: all-or-nothing accuracy. All-or-nothing implies that only the minimum number of corners to segment a figure are found in order for a stroke to be considered correctly segmented. In other words, for a stroke to be counted a correct stroke it must have no false positives or negatives. This accuracy is calculated by taking the number of correctly segmented strokes divided by the total number of strokes. We feel that all-or-nothing accuracy is a more important accuracy measurement since ShortStraw is designed to be used quickly in user interfaces and we do not want users to become frustrated if their polylines they do not segment correctly. From a user's point of view, the computer is either correct or it is wrong, and we wanted to model this behavior in our results.

The results from our tests can be found in Table 1, and examples can be seen in Figure 4.

	ShortStraw	Sezgin	Kim
False Positives	32	42	76
False Negatives	38	324	387
Correct Corners Found	1804	1518	1455
Total Correct Corners	1842	1842	1842
Correct Corners Accuracy	0.979	0.824	0.790
All-or-Nothing Accuracy	0.741	0.278	0.297

Table 1: *Accuracy results for ShortStraw and two baseline corner finders. The results are for a set of 244 polyline shapes drawn by six different users.*

5. Discussion

ShortStraw has an outstanding improvement over both current baseline corner finders. The all-or-nothing accuracy for ShortStraw is over twice times that of the second-best corner finder, our Kim and Kim implementation. Furthermore, ShortStraw greatly improves upon the [SSD01] version of accuracy measuring the correct number of corners found, as the algorithm has one-tenth the number of false negatives than either the [SSD01] algorithm or the [KK06] algorithm.

The implementation of ShortStraw is also very simple, and we provide the entire algorithm in the Appendix section of this paper. We had a sophomore undergraduate student unfamiliar with sketch recognition read our paper and code our algorithm. After completion, the student mentioned that the algorithm was "fairly easy to implement", and the entire time to read the paper, understand the algorithm, finish the implementation, and debug and test the code took the undergraduate took only 5-6 hours.

ShortStraw has some other benefits that have not been previously mentioned. ShortStraw is a very quick algorithm and not computationally intensive, so it can be easily used on mobile devices such as PDAs or touch-screen cell phones. A quick analysis of ShortStraw shows that resampling the points takes only $O(n)$ time and $O(n)$ memory. Calculating the straws for each point also runs in $O(n)$ time, as well as finding the initial corner fit. The only two sections of the algorithm that do not run in linear time include calculating the median straw length (which can run as quickly as $O(n \log n)$ with an efficient sorting algorithm), and the POST-PROCESS-CORNERS function, which runs in time $O(cn)$ where c is the number of corners found in the stroke. In the very unlikely case that every stroke point is a corner ($c = n$) AND all of the corners were missed during initial processing (requiring each stroke point to be added as a corner via the HALFWAY-CORNER function that searches for a corner under relaxed constraints), this function, and, thus, the entire algorithm, has a worst case scenario of $O(n^2)$ running time.

To further reduce the algorithm's computation time, the Euclidean distance measurement for calculating the straw length can be replaced with a squared distance measurement. This eliminates the need to perform over n square root calculations since the actual length of the straw is not important (only the straw's relation to the median straw length). We refrain from performing that step in the description of the algorithm to make the explanation easier to conceptualize for quick understanding and implementation. All additional distance calculations after the straws are computed, such as the path distance calculations in the IS-LINE function, must then use the squared distance measurement as well to remain in the same scale as the straws.

Another important aspect of ShortStraw is that the corner finding algorithm does not use any temporal information. Our corner finder could therefore be used in conjunction with systems that reconstruct strokes from static, offline images [QY04], whereas the algorithm in [SSD01] relies on speed information to locate corners.

Both baseline corner finders are designed to work with complex fits as well as polylines, whereas ShortStraw is designed only for polylines. Our algorithm is not designed to work well with arc and curvature segments since the median straw length of strokes with high curvature vary widely.

Finally, although ShortStraw does not explicitly use the word "curvature", each straw or chord length is in essence a simplified form of curvature. Instead of calculating curvature as the change in tangent across a series of points, a straw is a more naive representation for how bent a series of points are. If we were to redescribe our algorithm in terms of curvature, on a global scale we resample using a large number of points, and then we progressively "compute curvature"

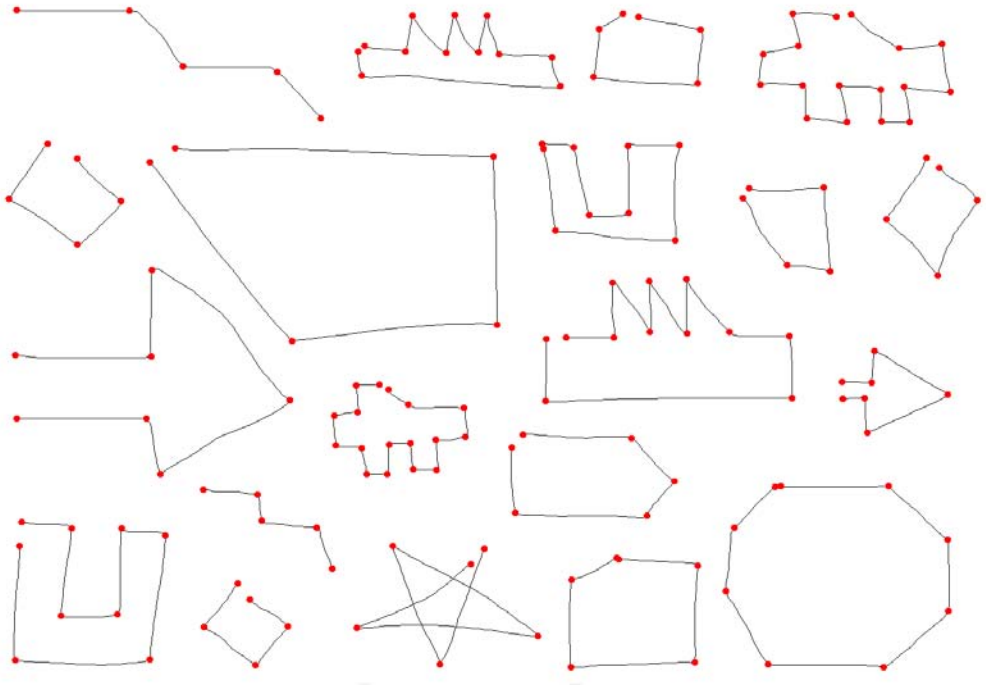


Figure 4: *Examples of correctly classified symbols by ShortStraw. These symbols come from the set of 244 polyline shapes drawn by six test users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper.*

over an expanse of 7 points (our straws). The intuition behind the improvement gained from this algorithm compared to other algorithms is that we are able to effectively smooth the stroke to remove noise without the common problem of removing corner precision:

- **Smooths out noise:** Both resampling and computing straw lengths across 7 points cause the algorithm to be less susceptible to the pixelized noise commonly prevalent in stroke points.
- **Keeps corner precision:** Because the resampled stroke still contains a large number of points, and, because the system progressively computes the straw lengths by moving only one resampled stroke point at a time, the algorithm is able to keep the corner precision which is usually lost during stroke smoothing.

6. Future Work

One of the areas where ShortStraw can improve is in recognizing corners at heavily obtuse angles, such as in Figure 5. Obtuse angles are sometimes too close to shallow arcs or slightly curved lines for ShortStraw to recognize that there should be a corner. A possible fix for this issue involves utilizing a varied window or straw length for each point, such as the method for dynamic chord lengths provided in [TC89]. Although this technique would sacrifice simplicity, the benefits might offset the obtuse angle problem.

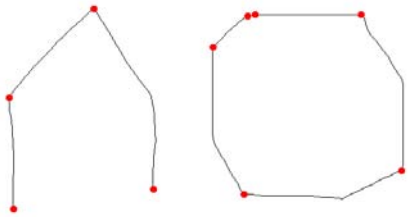


Figure 5: *ShortStraw misses corners most often at obtuse angles, such as in the two figures above.*

7. Conclusion

We presented ShortStraw, an accurate polyline corner finder that is easy to understand and implement. ShortStraw allows users to draw polylines free-form while achieving a very high all-or-nothing accuracy measure that is far beyond the current baseline corner finders. Our corner finder can be quickly integrated into sketch-based interfaces such as for route planning or node finding.

8. Acknowledgments

This work is supported in part by the National Science Foundation (Grant numbers 0744150 and 0757557).

References

[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.

[DP73] DOUGLAS D., PEUCKER T.: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 2 (1973), 112–122.

[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.

[HD05] HAMMOND T., DAVIS R.: Ladder, a sketching language for user interface developers. *Elsevier, Computers and Graphics 28* (2005), 518–532.

[HS92] HERSHBERGER J., SNOEYINK J.: *Speeding Up the Douglas-Peucker Line-Simplification Algorithm*. Tech. rep., Vancouver, BC, Canada, Canada, 1992.

[HSN04] HSE H., SHILMAN M., NEWTON A.: Robust sketched symbol fragmentation using templates. *Proceedings of the 9th international conference on Intelligent user interface* (2004), 156–160.

[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.

[ML05] MASRY M., LIPSON H.: A sketch-based interface for iterative design and analysis of 3d objects. In *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling* (2005), pp. 109–118.

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

[QY04] QIAO Y., YASUHARA M.: Recovering dynamic information from static handwritten images. 118–123.

[RC92] RATTARANGSI A., CHIN R.: Scale-based detection of corners of planar curves. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 14*, 4 (Apr 1992), 430–449.

[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.

[SD06] SEZGIN T. M., DAVIS R.: Scale-space based feature point detection for digital ink. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, p. 29.

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

[SZ05] SHUMIN ZHAI PER-OLA KRISTENSSON B. A. S.: In search of effective text input interfaces for off the desktop computing. In *Interacting with Computers 17* (2005), pp. 229Ű–250.

[TC89] TEH C. H., CHIN R. T.: On the detection of dominant points on digital curves. *IEEE Trans. Pattern Anal. Mach. Intell. 11*, 8 (1989), 859–872.

[VD04] VESELOVA O., DAVIS R.: Perceptually based learning of shape descriptions. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)* (San Jose, California, 2004), pp. 482–487.

[WWL07] WOBBROCK J. O., WILSON A. D., LI Y.: Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology* (New York, NY, USA, 2007), ACM, pp. 159–168.

[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.

9. Appendix

This section contains the full algorithm for ShortStraw in object-oriented pseudo-code. The variable *points* contains a sequential series of (x,y) points, whereas *corners* contains a set of indices that reference points. For example, $corner_i = j$ indicates that $point_j$ is the i^{th} corner found.

Algorithm 1: MAIN(*points*)

Input: A series of original, non-resampled points
Output: The corners for the resampled points

```
1 S ←DETERMINE-RESAMPLE-SPACING(points)
2 resampled ←RESAMPLE-POINTS(points, S)
3 corners ←GET-CORNERS(resampled)
4 return corners
```

Algorithm 2: RESAMPLE-POINTS(*points*, *S*)

Input: A series of points and an interspacing distance
Output: The resampled points

```
1  D ← 0
2  resampled ← points_0
3  for i ← 1 to |points| do
4      d ← DISTANCE(points_{i-1}, points_i)
5      if D + d ≥ S then
6          q.x ←
           points_{i-1}.x + ((S − D)/d) × (points_i.x − points_{i-1}.x)
7          q.y ←
           points_{i-1}.y + ((S − D)/d) × (points_i.y − points_{i-1}.y)
8          APPEND(resampled, q)
9          INSERT(points, i, q)
10         D ← 0
11     else
12         D = D + d
13 return resampled
```

Algorithm 3: DETERMINE-RESAMPLE-SPACING(*points*)

Input: A series of points
Output: The interspacing distance for the resampled points

```
1  topLeft.x ← MIN_x(points)
2  topLeft.y ← MIN_y(points)
3  bottomRight.x ← MAX_x(points)
4  bottomRight.y ← MAX_y(points)
5  diagonal ← DISTANCE(bottomRight, topLeft)
6  S ← diagonal/40.0
7  return S
```

Algorithm 4: GET-CORNERS(*points*)

Input: A series of resampled points
Output: The resampled points that correspond to corners

```
1  corners ← ∅
2  APPEND(corners, 0)
3  W ← 3
4  for i ← W to |points| − W do
5      straws_i ← DISTANCE(points_{i−W}, points_{i+W})
6  t ← MEDIAN(straws) × 0.95
7  for i ← W to |points| − W do
8      if straws_i < t then
9          localMin ← +∞
10         localMinIndex ← i
11         while i < |straws| & straws_i < t do
12             if straws_i < localMin then
13                 localMin ← straws_i
14                 localMinIndex ← i
15             i ← i + 1
16         APPEND(corners, i)
17 APPEND(corners, |points|)
18 corners ← POST-PROCESS-CORNERS(corners, straws)
19 return corners
```

Algorithm 5: PATH-DISTANCE(*points*, *a*, *b*)

Input: A series of points and two indices, *a* and *b*
Output: The path (stroke segment) distance between the points at *a* and *b*

```
1  d ← 0
2  for i ← a to b − 1 do
3      d ← d + DISTANCE(points_i, points_{i+1})
4  return d
```

Algorithm 6: DISTANCE(*points*, *a*, *b*)

Input: A series of points and two indices, *a* and *b*
Output: The Euclidean (chord) distance between the points at *a* and *b*

```
1  Δx ← points_b.x − points_a.x
2  Δy ← points_b.y − points_a.y
3  return √(Δx² + Δy²)
```

Algorithm 7: POST-PROCESS-CORNERS(*points*, *corners*, *straws*)

Input: A series of resampled points, an initial set of corners, and the straw distances for each point
Output: A set of corners post-processed with higher-level polyline rules

```
1  continue ← FALSE
2  while ¬continue do
3      continue ← TRUE
4      for i ← 1 to |corners| do
5          c_1 ← corners_{i−1}
6          c_2 ← corners_i
7          if ¬IS-LINE(points, c_1, c_2) then
8              newCorner ←
               HALFWAY-CORNER(straws, c_1, c_2)
9              INSERT(corners, i, newCorner)
10             continue ← FALSE
11 for i ← 1 to |corners| − 1 do
12     c_1 ← corners_{i−1}
13     c_2 ← corners_{i+1}
14     if IS-LINE(points, c_1, c_2) then
15         REMOVE(corners, corners_i)
16         i ← i − 1
17 return corners
```

Algorithm 8: HALFWAY-CORNER(*straws*, *a*, *b*)

Input: The straw distances for each point, two point indices *a* and *b*
Output: A possible corner between the points at *a* and *b*

```
1  quarter ← (b − a)/2
2  minValue ← +∞
3  for i ← a + quarter to b − quarter do
4      if straws_i < minValue then
5          minValue ← straws_i
6          minIndex ← i
7  return minIndex
```

Algorithm 9: IS-LINE(*points*, *a*, *b*)

Input: A series of points and two indices, *a* and *b*
Output: A boolean for whether or not the stroke segment between points at *a* and *b* is a line

```
1  threshold ← 0.95
2  distance ← DISTANCE(points_a, points_b)
3  pathDistance ← PATH-DISTANCE(points, a, b)
4  if distance/pathDistance > threshold then
5      return TRUE
6  else
7      return FALSE
```