PaleoSketch: Accurate Primitive Sketch Recognition and Beautification

Brandon Paulson
Sketch Recognition Lab, Texas A&M University
Department of Computer Science
College Station, TX 77843-3112 USA
bpaulson@cs.tamu.edu

Tracy Hammond
Sketch Recognition Lab, Texas A&M University
Department of Computer Science
College Station, TX 77843-3112 USA
hammond@cs.tamu.edu

ABSTRACT

Sketching is a natural form of human communication and has become an increasingly popular tool for interacting with user interfaces. In order to facilitate the integration of sketching into traditional user interfaces, we must first develop accurate ways of recognizing users' intentions while providing feedback to catch recognition problems early in the sketching process. One approach to sketch recognition has been to recognize low-level primitives and then hierarchically construct higher-level shapes based on geometric constraints defined by the user; however, current low-level recognizers only handle a few number of primitive shapes. We propose a new low-level recognition and beautification system that can recognize eight primitive shapes, as well as combinations of these primitives, with recognition rates at 98.56%. Our system also automatically generates beautified versions of these shapes to provide feedback early in the sketching process. In addition to looking at geometric perception, much of our recognition success can be attributed to two new features, along with a new ranking algorithm, which have proven to be significant in distinguishing polylines from curved segments.

Author Keywords
Sketch recognition, shape beautification, low-level processing, pen-based interfaces.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces – Input devices and strategies.

INTRODUCTION

Sketching has become an increasing popular form of human-computer interaction due to the increased use of Tablet PCs. Furthermore, tools have been developed that allow sketching to be easily incorporated into user interfaces [5][9]. It has also been argued that sketched gestures are typically easier to remember than textual commands [7]. Although there is much evidence to support the usefulness of incorporating sketching into user interfaces, sketch recognition as a whole has not yet entered into mainstream technology. We believe this is because current recognizers place many constraints on how users must draw certain shapes. Users are forced to learn the system rather than the system having to learn users' intentions. Our recognizer places virtually no constraints on how users must draw individual shapes; they are allowed to draw freely. Currently, our recognizer classifies single strokes into primitives. Primitives drawn with multiple strokes can be merged by an upper-level recognition system. A stroke is defined as the set of points (consisting of an x coordinate, y coordinate, and time value) sampled between pen down and pen up events.

In addition to placing few drawing constraints on users, our recognizer has several other benefits. First, it is able to return multiple interpretations. This capability allows recognition errors to be easily corrected as a simple pen-click could be used to switch to an alternative interpretation. Our system also recognizes more primitives than other popular low-level recognizers [10][12]. In particular, our recognizer distinguishes circles from ellipses and arcs from curves. We also recognize spirals and helixes; two shapes not supported by most low-level, geometric recognizers. By increasing the number of primitives capable of being recognized, we will allow sketch-based interfaces to be created for domains that include shapes which were previously indescribable in terms of the previous primitive lists. The shapes our system currently recognizes include:

- Line: a stroke with a relatively constant slope between all sample points
- Polyline: a stroke consisting of multiple, connected lines
- Circle: a stroke that has a total direction close to 2π, constant radius between the center point and each stroke point, and whose major and minor axes are close in size
- Ellipse: a stroke with similar properties of a circle, but whose major and minor axes are not similar.
- Arc: a segment of an incomplete circle
- Curve: a stroke whose points can be fit smoothly up to a fifth degree curve

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IUI'08, January 13-16, 2008, Maspalomas, Gran Canaria, Spain.
Copyright 2008 ACM 978-1-59593-987-6/ 08/ 0001 $5.00

- Spiral: a stroke that is composed of a series of circles with continuously descending (or ascending) radii but a constant center.
- Helix: a stroke that is composed a series of circles with similar radii but with moving centers. We also assume that helixes are drawn linearly.

Our recognizer has significantly better recognition rates than existing algorithms because of two new features we have developed. We will show that these features, along with our novel ranking algorithm, can be used to aid in shape recognition, particularly when determining polylines from curved shapes. We will also compare the overall accuracy of our system to the commonly used recognizer presented in [10].

PREVIOUS WORK

The idea of interacting with computers via pen-based input began in 1964 with the seminal work of Ivan Sutherland's Sketchpad system [11]. In 1991, Dean Rubine proposed a gesture recognition toolkit, GRANDMA, which allowed single-stroke gestures to be learned and later recognized through the use of a linear classifier [8]. Rubine proposed thirteen features that could be used to describe any single-stroke shape. Rubine also provides two techniques for rejecting bad gestures.

Rubine's work was later extended by Long et al. [6], who determined a new feature set that consisted of eleven of Rubine's features along with six of their own. This feature set was chosen after multi-dimensional scaling was used to determine the most relevant features. Like Rubine's recognizer, a linear classifier was used to classify single stroke shapes.

One disadvantage of these first two systems is that they use feature-based techniques which require extensive training. Furthermore, because of the features chosen, these systems required that strokes be drawn in the same manner each time they were drawn. For example, a circle drawn in a clockwise manner would not be the same as a circle drawn in a counter-clockwise manner. Such constraints produce recognizers that are heavily user-dependent. We want users to be able to draw shapes as they would naturally. When we tested the Rubine recognizer on our data, it only achieved an overall accuracy of 54.2%. This accuracy shows how feature-based classifiers which place many constraints on the user can perform poorly on natural sketch data.

Because feature-based techniques required much training on the users' behalf, a shift occurred towards more geometric-based recognizers. This includes two systems which greatly influenced the work presented in this paper. The first system is that of Sezgin et al. In [10], a system was described that was composed of an approximation stage, a beautification stage, and a recognition stage. The system used a novel approach to detect corners in sketched strokes. Corners were detected by finding the points of highest curvature, along with the points of lowest speed. These points were determined by finding points above or

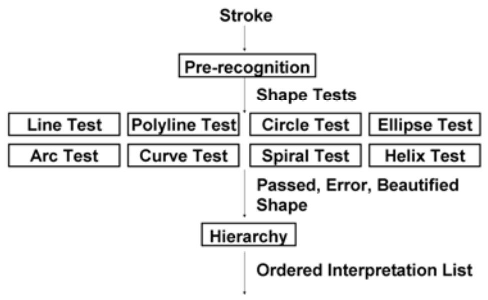

Figure 1. Recognizer Architecture

below a certain threshold. Hybrid fits between these two sets of points were calculated using an average based filtering technique. The error of each fit was determined by using an orthogonal distance squared error. The system was limited to only a few primitive shapes: lines, ellipses, and complex fits (fits composing of a combination of lines and curves). For complex fits and vertex (corner) approximation, the authors reported an accuracy of 96%.

In [12], Yu and Cai present an alternative low-level sketch recognizer. Corners were detected using only curvature data. The technique of using curvature as a means to segment strokes has been used in other works as well [4]. The most significant contribution of Yu and Cai was the introduction of a feature area error metric, which is also used in our recognizer. The shape set of the Yu recognizer was expanded to include lines, polylines, circles, ellipses, arcs, and helixes. For primitive shapes, they achieved near 98% accuracy; however, they admit to not having produced the recognition rates of Sezgin for complex shapes.

Low-level recognizers, such as those presented by Sezgin and Yu, are not adequate for most sketch recognition domains. Typically, diagrams and sketches consist of symbols that are more complex than the primitives supported by these recognizers. Therefore, tools such as LADDER [2] have been created which allow users to describe higher level symbols as a combination of lower-level primitives meeting certain geometric constraints. UML diagrams, mechanical engineering diagrams, circuit diagrams, military course of action diagrams, and flow charts are all examples of various sketch systems that have been produced using this shape definition language. Others have attempted to improve upper-level recognition by using context [3]; however, the use of context typically requires domain knowledge.

The work in this paper uses many of the concepts learned from the Sezgin and Yu systems. We improve on these systems with the goal of being able to recognize a larger number of primitive shapes while still maintaining a high recognition rate.

IMPLEMENTATION

The overall architecture for our recognizer can be seen in Figure 1. The recognizer takes a single stroke as input, and performs a series of pre-recognition calculations. Once these values have been computed, the stroke is then sent to various low-level shape recognizers. Each shape recognizer returns a Boolean flag stating whether the recognizer passed or failed, as well as a beautified shape object (in the form of Java2D shapes) that best fits the input stroke. Once the recognizers for all possible shapes are executed, the results are then sent to a hierarchy function which sorts the interpretations in order of best fit. We will first discuss the pre-recognition stage, followed by each individual shape test, and finally we will discuss the hierarchy we chose for ordering fits.

Pre-recognition

We begin pre-recognition by first removing consecutive, duplicate points from the stroke. These points can occur in systems with a high sampling rate. If two consecutive points either have the same x and y values or if they have the same time value then the second point is removed.

Next, a series of graphs and values are computed for the stroke, including direction graph, speed graph, curvature graph, and corners. The graphs are computed using methods from [10] and [12], whereas corners are calculated using a simple corner finding algorithm presented in the appendix. This algorithm is meant to produce a polyline interpretation that closely fits the original shape (even for non-polyline strokes). The polyline interpretation in Figure 3 was produced from this algorithm.

In addition to these graphs, we also compute two new features which we have found to be very helpful to the recognition process. The first is *normalized distance between direction extremes* (NDDE). To calculate this feature, we first take the point with the highest direction value (change of y over change of x) and the point with the lowest direction value and compute the stroke length between these two points. This length is then divided by the length of the entire stroke, essentially giving us the percentage of the stroke that occurs between the two direction extremes. For curved shapes, such as arcs, the highest and lowest directional values will typically be near the endpoints of the stroke, thus yielding very high NDDE values. Polylines, on the other hand, normally have one or more spikes in their direction graphs. These spikes often cause the point of highest or lowest directional value to no longer be near the endpoints of the stroke. Therefore, polylines typically will have NDDE values which are lower than curved strokes (see Figure 2).

Our second new feature also aids in determining polylines from curved strokes. We call this *direction change ratio* (DCR). Like NDDE, DCR is also meant to gauge whether or not spikes are present in the direction graph. This value is computed as the maximum change in direction divided by the average change in direction. Because there tends to be a great deal of noise at the beginning and ending of a stroke (what we typically refer to as "tails"), we ignore the first and last 5% of the stroke we calculating this feature. As seen in Figure 2, the polyline has a large downward spike in its direction graph, whereas the arc has relatively little change between consecutive direction values. Therefore, polylines will typically have higher DCR values than curved strokes.

As mentioned before, "tails" at the endpoints of strokes can be significant problems for recognition. We attempt to remove these tails before sending the stroke to each of the shape tests. To determine if a tail is present, we analyze the first and last 20% of the stroke and find the point within each section that has the highest curvature. If that curvature is higher than a threshold^A, then we break the stroke at that point and remove the tail. We do not perform tail removal on strokes with a low number of points^B or with too small of stroke length^C. (*Please note that we will use superscripts throughout this paper to denote thresholds whose values can be found in the appendix of this paper.)*

After removing tails, we conduct two other tests before performing any actual recognition. The first test determines whether or not the stroke is overtraced. Using the direction graph we can determine whether or not a stroke is overtraced based on the number of revolutions the stroke makes. This can be calculated by computing the total rotation of the stroke, as mentioned in [8], and dividing it by 2π. If the number of revolutions is greater than a certain threshold^D then we consider the stroke to be overtraced. We use the term "overtraced" to denote shapes that make multiple revolutions. This does not necessarily mean that each revolution is the same shape, as is the case of an overtraced circle. For example, a helix is also considered overtraced since it makes several revolutions which are self-intersecting.

Finally, a test is used to determine if the stroke represents a closed figure. We begin by computing the distance between the endpoints and dividing it by the stroke length. In order for a stroke to be closed this ratio must be less than some threshold^E and the number of revolutions must be greater than another threshold^F (different from the overtraced threshold).

Line Test

To determine if a stroke is a line, the following conditions must be met. First, we fit a least squares line to the stroke

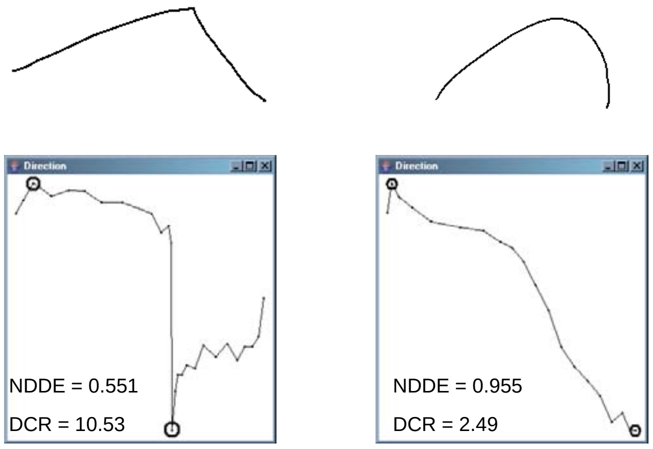

Figure 2. Direction graphs for a polyline (left) and arc (right). The points of highest and lowest direction value are circled. The polyline has a lower NDDE value because its minimum direction value is in the middle of the stroke, whereas the arc has its direction extremes closer to the endpoints.

points. The orthogonal distance squared between the best fit line and the actual stroke points is calculated similar to that in [10]. This distance is divided by the stroke length to determine the least squares error, which must be below a certain threshold[G]. Next, the feature area of the line is determined using techniques from [12]. Again this value is divided by the stroke length to get an error which must be within a threshold[H]. Finally, we verify that the stroke is not overtraced and only contains two (or three) corners (one for each of the endpoints). We allow three corners in order to account for occasional examples in which the line was drawn with a lot of noise. If all of these conditions are met, then we accept the stroke as a single line.

We return a beautified shape object that consists of the line created by connecting the endpoints of the stroke. We could have alternatively returned the computed best fit line, but we believe that endpoints are significant in sketching, particularly in diagramming domains where lines are meant to act as connectors. This endpoint-significance theme continues in the beautification of the other low-level shapes as well.

Polyline Test

The polyline test begins by breaking the stroke into sub-strokes at the calculated corners. We send each sub-stroke to the line test and keep track of the sum of least squares errors as well as the sum of the feature area errors. These errors are normalized by dividing by the length of the stroke. In order for a stroke to pass as a polyline, one of three conditions must exist.

1. Each sub-stroke must pass the line test.
2. The average least squares error of each sub-stroke must be less than some threshold[I].
3. The stroke has a high DCR value[J].

The shape object returned consists of the lines formed by connecting the corners calculated from the corner finder.

Ellipse Test

One advantage of our ellipse test compared to others' is that we allow for rotated, as well as overtraced, ellipses. To allow for this, we first calculate the ideal major axis, center, and minor axis. The major axis is determined by finding the two stroke points that are the furthest away from each other. The center is taken as the average x and y values of the stroke points. The minor axis is constructed as the perpendicular bisector of the major axis at the center point. This line is extended and clipped where it meets the stroke points, which may be at interpolated values. Once these values are computed, we then check to make sure the following conditions apply:

1. The stroke must have passed the closed shape test from pre-recognition.
2. The stroke's NDDE value must be high[K]. This value tends to be less relevant for small ellipses, so this condition is ignored if the major axis does not meet certain length requirements[L].
3. The feature area error (feature area divided by area of the ideal ellipse, as described in [12]) must be less than some threshold[M]. If the stroke is overtraced, then it is broken into sub-strokes at each 2π interval in the direction graph. All of the sub-strokes (minus the last sub-stroke, as it may be an incomplete ellipse) are then fit to ellipses and the error becomes the average feature area error across each sub-ellipse.

The shape object returned consists of the beautified ellipse formed from the ideal center, major axis length, and minor axis length. This shape is then rotated around the center point based on the angle of the major axis.

Circle Test

For circles, we re-use many of the tests conducted for ellipses. We start by first calculating an ideal radius, which is computed as the average distance between each stroke point and the ideal center from the ellipse test. Once we have this value, we then verify that the stroke passed conditions 1 and 2 (again, small circles[N] according to radius are ignored for NDDE) of the ellipse test. To verify that a stroke is better fit with a circle rather than an ellipse, we find the major axis to minor axis length ratio, which after subtracted from 1.0, must me less than some value[O]. We also perform feature area error verification; however, the feature area is divided by the area of an ideal circle rather than an ideal ellipse. This error must be less than some threshold[P]. We handle overtraced circles similarly to the way we handled overtraced ellipses. The shape object returned is the beautified circle formed from the calculated center (from ellipse test) and radius.

Arc Test

For our recognizer, we consider arcs to be segments of circles; therefore, in order to determine the best fit arc, we need to determine the best fit circle that the arc is a part of. To do this, we first calculate the ideal center point of the arc using a series of perpendicular bisectors. First, we connect the endpoints of the stroke and find the perpendicular bisector at the midpoint. We determine where that bisector intersects the stroke (through interpolation between stroke points) and then connect two more lines from that point to the endpoints. We take two more bisectors at the midpoints of those two lines and find where they intersect each other. That intersection point is the center of our arc.

We then calculate the ideal radius of the arc by taking the average distance between the stroke points and the center point. In order to pass the arc test a stroke must not be closed or overtraced and must have a NDDE value that is high[K]. Once again, the NDDE value is ignored in the cases of small arcs[N] (based on radius). We also verify that the stroke's DCR value is low[J]. Finally, we calculate the feature area of the arc and make sure its error is below a certain threshold[Q]. A beautified arc is constructed from the ideal center, radius, and angles between the center point and endpoints. As with lines, we want to make sure that endpoint consistency is maintained.

Curve Test

In our implementation, we could allow for any degree curve; however, we need to limit this degree not only to keep recognition to a practical running time, but also because complex figures could easily be represented by some *n*-degree curve. For our system, we limit curves to fourth and fifth degree curves, choosing the degree that best fits the stroke points. Unlike other tests where we generate ideal shapes after checking for passing conditions, for curves we generate the ideal shape first, and then check the least squares error of the generated shape against the actual stroke points.

To generate an ideal curve, we use the Bézier curve formula. In order to use this formula we must first calculate *d+1* control points, where *d* is the degree of the Bézier curve. Currently, we use a naïve approach to approximate these points. To find these points, we first find the parametric value of each stroke point. We determine a point's parametric value by dividing the length of the stroke up to that point by the length of the entire stroke. Once these values are computed, we take the endpoints (whose parametric values are 0 and 1), as well as, *d-1* other points which are spread evenly across the stroke. For example, for a fourth degree curve, we would take the endpoints, along with the points whose parametric values were close to 0.25, 0.5, and 0.75. We then estimate the control points by solving a system of equations using these selected points and their corresponding parametric values. Once we have the estimated control points, we can then generate the ideal curve according to the Bézier curve formula where *n* is the degree of the curve and P_i is the set of computed control points:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} P_i (1-t)^{n-i} t^i$$

To pass the curve test, a stroke must have a low[J] DCR value as well as a low least squares error[R]. The least squares error can be easily computed by plugging the corresponding parametric values of each of the stroke points into the Bézier curve equation and comparing it to its actual location. The beautified shape object is simply the curve generated by the Bézier curve equation.

Spiral Test

For spirals, we begin by breaking the stroke up at every 2π interval in the direction graph like we did for overtraced circles and ellipses. The center of the spiral is taken to be the center of the bounding box for the entire stroke. The average radius is calculated as in the circle test. Strokes passing the spiral test must meet the following conditions:

1. The stroke must be overtraced.
2. The stroke's NDDE value must be high[K].
3. Each sub-stroke (minus the last sub-stroke) is fit to a circle. Although we do not expect these strokes to necessarily pass the circle fit test, we are able to calculate the ideal radius of each sub-stroke. For spirals, these radii must either be completely ascending or completely descending through the progression of each sub-stroke.
4. The average radius of the stroke divided by the bounding box radius must be less than a threshold[S]. The radius of the bounding box is the average between half of the width and half of the height. This test is used to aid in determining the difference between spirals and other overtraced shapes (mainly circles). For overtraced circles, the bounding box radius will be very close to the average radius, whereas with spirals the average radius will typically be smaller because each consecutive sub-stroke should get closer and closer to the center point of the spiral.
5. The centers of each consecutive sub-stroke must be close to each other. To test this we find the sum of the distances between the centers of each consecutive sub-stroke and divide it by the average radius times the number of revolutions the spiral makes. This value must be less than some threshold[T].
6. To further check the closeness of centers, we find the centers of the sub-strokes that are farthest apart. The distance between these centers should not exceed the diameter of the spiral.
7. Finally, we calculate the distance between endpoints and divide it by stroke length. This value is helpful for distinguishing spirals from helixes. In helixes, this value will be high whereas spirals should be lower. We verify that this value is below another threshold[U].

To return the shape object that represents the beautified spiral, we generate an Archimedes spiral that starts at the center point and has the polar equation, $r = a\theta$. In this equation the radius, *r*, changes as θ changes. The *a* value represents the "tightness" of the spiral. We set this value to be the bounding box radius divided by 2π times the number of revolutions. To generate the spiral we continuously increment (or decrement) the theta value until the radius value reaches the bounding box radius. We decide whether to increment or decrement theta based on whether the spiral was drawn clockwise or counter-clockwise (which can be determined by looking at the slope of the direction graph). In order to preserve the outer-most endpoint of the spiral, we can shift the theta value by the angle created from the endpoint and the starting center point. Once we find the *r* value that corresponds to the current theta, we simply plug it into the polar equation for a circle to generate x and y coordinates.

Helix Test

Helixes are essentially checked for during the spiral test so little work is required for this test. For a stroke to be passed as a helix it must pass conditions 1 and 2 of the spiral test, but must fail condition 7.

Creating a beautified helix object is more complex than creating a beautified spiral, since we want to maintain both of the endpoints of the stroke. Essentially, we want to use the polar equation of a circle like we did for spirals, but instead of iteratively changing the radius, we want to instead change the position of the center point and maintain a constant radius. We chose the constant radius as the average distance between the stroke points and the major axis of the stroke (as calculated from the ellipse test). We also find our starting and ending center points by finding the points on the major axis that are a length equal to the radius away from the endpoints of the stroke. Once these two points are specified, we parametrically find the center point for the current iteration. The parametric value used is the absolute theta value divided by 2π times the number of revolutions. We continue to increase or decrease theta until its absolute value becomes greater than 2π times the number of revolutions (the total direction).

Complex Test

A complex fit is a default interpretation in instances where none of the above tests (with the exception of curves and polylines) pass. Since it is a default interpretation, this test will always pass but may not always be used as an interpretation returned by the hierarchy. Our method for handling complex shapes is much like that of [12]. We, too, bias towards a complex fit with the fewest number of primitive shapes. We start by breaking a stroke up into two sub-strokes at the point of highest curvature. Each of these strokes is then recursively sent back into the recognizer until we get all non-polyline primitives.

We then add an additional step, which is different from the implementation of Yu and Cai. Since breaking a stroke at the point of highest curvature may not always guarantee that we break at a logical location, we send our sub-strokes to a secondary function which attempts to recombine consecutive sub-strokes and check to see if they can be recognized as a single primitive. If they can, then the two sub-strokes are replaced with a single sub-stroke that is a combination of the two. Because our recognizer also returns beautified shapes, we can also generate a beautified complex shape by simply combining all of the shapes returned from the sub-strokes.

Hierarchy

Since we do not use a consistent error metric across all shape tests, we had to create a hierarchy to sort our interpretations. When ordering interpretations the hardest problem is determining when a complex, curve, or polyline fit is the best fit for a stroke. To help solve this problem we came up with a new ranking algorithm that takes advantage of an inherent property of the corner finding algorithm we used. The corner finding algorithm presented in the appendix is meant primarily to converge to the corners of polylines only. When applied to curved strokes the algorithm tries to produce the best possible polyline interpretation with as few lines as it can. This typically will yield a large number of corners across the curved segment. We use this property to our advantage, as we can predict approximately the minimum number of corners that would be found in most of the curved segments such as arcs, curves, circles, and ellipses. We assigned a shape score based on our initial observations. Essentially, these scores tell us the minimum number of lines we will accept in place of these shapes. The scores were as followed:

Line – 1	Arc – 3
Curve – 5	Circle – 5
Ellipse - 5	Helix – 5
Spiral - 5	

Helixes and spirals are hard to assign scores because they are arbitrarily large and the number of rotations differs across each occurrence. Therefore, we gave them a default score of 5. In order to determine whether or not we should choose a polyline fit over a complex fit, we simply sum the ranks of each interpretation and choose the fit with lowest rank (complex wins tie). Figure 3 gives an example of the algorithm in work.

Interpretations are ordered according to our hierarchy, which can be referenced in the appendix of this paper. Once we have ordered the interpretation, we perform one last calculation in our hierarchy function. If a complex fit was added to the interpretation list then we check it for sub-stroke "tails" (typically in the form of very small lines or curves) that occur at the endpoints of the stroke. If the length of the first or last sub-stroke in a complex interpretation divided by the entire stroke length is less than some threshold^{V} then we remove it from the complex fit. If this reduces the size of the complex fit to a single shape then we replace the complex fit with the appropriate single shape interpretation.

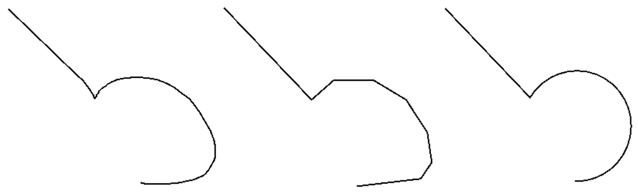

Figure 3. An input sketch (left) along with two possible interpretations – a polyline interpretation (middle) and a complex interpretation made up of one line and one arc (right). To determine which of these interpretations is best we sum their ranks. The polyline interpretation is made up of 8 lines and therefore has a rank of 8. The complex interpretation is made up of 1 lines and 1 arc, thus yielding a score of 1+3=4. In this case the complex interpretation is chosen over the polyline interpretation because of its lower rank.

RESULTS

In our study, we collected two sets of data. The first set of data consisted of 900 shapes collected from 10 users. Each user drew 10 examples of each primitive shape, along with 10 examples of a complex shape. We asked users to draw a complex shape consisting of one line and one arc, an example which some recognizers have difficulty in interpreting [12]. We used this first data set for training to help establish thresholds and develop our hierarchy.

We then collected a second data set of the same size with the same number of users and used this set to test our recognizer. In addition to running the test set through our recognizer, we also tested the data set against a modified version of the recognizer created by Sezgin et al. [10]. This recognizer was modified in work done by [1] to include the return of multiple interpretations. For our experiment, we wanted to see how often the correct interpretation was

	Correct Interpretation Produced				Correct Interpretation is Top Interpretation			
	Paleo	Paleo-F	Paleo-R	SSD	**Paleo**	Paleo-F	Paleo-R	SSD
Arc	1.00	0.98	1.00	N/A	1.00	0.35	1.00	N/A
Circle	1.00	1.00	1.00	1.00	0.97	0.68	0.68	0.83
Complex	1.00	0.43	0.99	1.00	0.97*	0.33	0.99	0.99†
Curve	0.99	1.00	0.99	N/A	0.99	0.09	0.74	N/A
Ellipse	1.00	0.99	1.00	1.00	1.00	0.74	0.75	0.56
Helix	1.00	1.00	1.00	N/A	1.00	0.99	1.00	N/A
Line	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.99
Polyline	1.00	1.00	1.00	1.00	0.94	0.91	0.58	0.92
Spiral	1.00	1.00	1.00	N/A	1.00	0.99	1.00	N/A
Total	**99.89%**	93.33%	99.78%	99.80%	**98.56%**	67.56%	86.00%	85.80%

Table 1. Results for each recognizer. Paleo refers to the proposed recognizer. Paleo-F is the proposed recognizer without the NDDE and DCR features included. Paleo-R is the proposed recognizer without the ranking algorithm and SSD is the recognizer from [10]. *Of these interpretations, all but one returned either a line/arc or line/curve interpretation. †Of these, 27% returned a one line, multiple curve[1] interpretation. All others consisted of a multi-line, multi-curve interpretation.

among the listed interpretations as well as how often the correct interpretation was the top or best interpretation. Since the Sezgin recognizer does not distinguish between ellipses and circles we will count circles as being correct if the Sezgin recognizer returns an ellipse. Also, because the Sezgin recognizer does not handle spirals, helixes, or individual arcs and curves we have omitted those examples from his recognizer. Approximation of curved regions is mentioned in [10], but only for complex shapes consisting of lines and curves.

We also wanted to see the impact that our new features, as well as our new ranking algorithm, had on the accuracy of the system. To see this effect we also tested our recognizer: a) without including the NDDE and DCR features and b) without including the ranking algorithm. In total we have tested four recognizers: our recognizer (denoted by "Paleo"), the Sezgin et al. recognizer ("SSD"), our recognizer without the two new features ("Paleo-F"), and our recognizer without the ranking system ("Paleo-R"). Table 1 displays the full results from our experiment.

In addition to testing the accuracy of our system, we also analyzed execution time. Our recognizer had a total recognition time of 26,539 milliseconds for all 900 examples, an average of 29.5 milliseconds per example. The Sezgin recognizer had a total recognition time of 23,212 milliseconds, an average of 25.8 milliseconds per example. Both recognizers obviously perform in real-time.

We also wanted to test the accuracy of our complex fits. We chose to have users draw complex examples consisting of one line and one arc, a notoriously hard example. The recognizer from [12] claimed to recognize these shapes with an accuracy of 70%. In our tests, the SSD recognizer correctly returned a one line, multiple curve[1] interpretation 27% of the time. For our recognizer, a "one line, one arc" interpretation was correctly returned 92% of the time (93.8% in cases where the complex interpretation was the top interpretation). All but one of the remaining 8% of complex examples consisted of a line/curve combination. Upon observation of these examples, it could be argued that the examples were interpreted correctly, since most of these examples were drawn with less circular and more elliptical arcs. Some examples can be seen in Figure 4. The remaining incorrect complex fit was recognized as a two curve interpretation because the line portion of the stroke was drawn in a curvy manner.

DISCUSSION

In our experiment, the correct shape interpretation was returned 99.89% of the time; however, only 98.56% of the time was the correct interpretation the top interpretation. Of this 1.44% error (12 examples), half came from polyline interpretations that were incorrectly classified as complex shapes. Most of these examples were polylines that were drawn in a wavy manner, as seen in Figure 5.

Of the remaining six misclassified examples, three came in the form of circles that were drawn more like ellipses (Figure 6). The other three examples were complex shapes that were misclassified as either a polyline or a curve. In one case, a polyline interpretation was chosen over a complex interpretation of one line, one curve (Figure 7). This occurred because the user drew an elliptical arc and

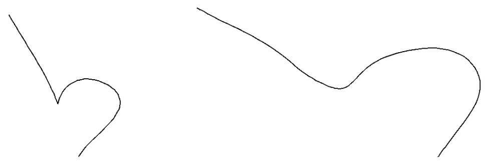

Figure 4. Examples of complex shapes drawn with elliptical arcs, thus causing the recognizer to return a one line, one curve interpretation rather than one line, one arc (the users' intentions). Future work would include finding multiple complex interpretations.

[1] We considered a multiple curve interpretation to be correct for single arcs since the SSD recognizer breaks arcs down into multiple curved segments.

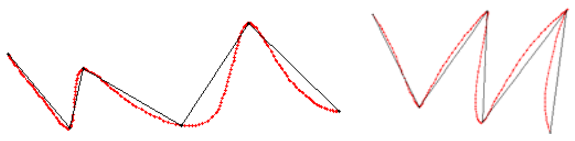

Figure 5. Polylines that were classified as complex shapes because their polyline fit (shown in black) contained too much error. For both of these examples a one line, two curve interpretation was returned in front of the polyline one

the corner finding algorithm only found four corners within the curved region. Because the complex rank of this interpretation (5) is greater than the polyline rank (4) the recognizer chose a polyline interpretation. If the recognizer would have chosen a one line, one arc interpretation (rank 4), then the ranks of the two interpretations would have been equal and a complex interpretation would have been correctly chosen as the top interpretation. The remaining complex error came in the form of curve fits being chosen over complex fits. These cases occurred when the user had a very curvy transition between shapes, and the corner we would typically segment at was not very defined.

Although it could be argued that most of our misclassified examples would also be misclassified by a human recognizer, we still consider these to be recognition errors since our goal is to capture user intention. In some cases, context could be used in a higher-level recognition system to help choose the lower ranking interpretation. For example, if we have a domain that would never contain shapes consisting of a mixture of lines and arcs, then we would know that in the ambiguous cases of polyline versus complex (line/arc combination), we should always choose a polyline interpretation. This is one of the advantages of having a low-level recognizer capable of returning multiple interpretations.

While it is important to return many interpretations, we also want to make sure that we prune away examples that we know cannot be correct. The average size of our ordered interpretation list was 2.68, meaning on average 2.68 out of the 9 shape tests passed per input stroke. One of the shapes in the list will be polyline, as it is always added as a default interpretation. We had a very high accuracy in producing the correct interpretation; however, our recognizer failed for a single example. This example was a curve which was classified as a complex fit of one line and one arc. The curve interpretation was not added to the interpretation list because it was determined that the transition was not smooth enough to be a single curve. The stroke had a high

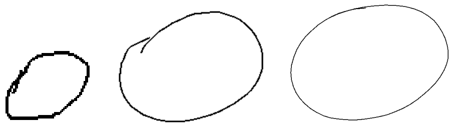

Figure 6. Circles (as intended by the user) misclassified as ellipses. A circle interpretation was, however, returned as an alternative interpretation. It could be argued that a human recognizer would also misclassify these shapes.

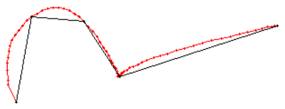

Figure 7. Complex shape in which a polyline interpretation (in black) was incorrectly chosen before the complex interpretation. This occurred because the recognizer returned a complex interpretation of one line, one curve which had a higher rank than the polyline interpretation.

DCR value (9.5) and low NDDE value (.72), which are not characteristic of typical curves. Figure 8 shows the misclassified curve.

When analyzing the results, our recognizer outperformed the SSD recognizer in all shapes except complex fits. In this case, we simply passed the SSD if it returned a complex fit; we were unable to test the fit to verify that a one line, one arc interpretation was indeed returned. Furthermore, we are able to recognize other shapes which are currently not supported by the SSD recognizer. We also can see that our two new features, NDDE and DCR, are significant in aiding the recognition process, particularly with arcs, curves, and complex shapes containing arcs and curves. The ranking algorithm had significance as well, particularly with distinguishing polylines, curves, circles and ellipses.

HIGH-LEVEL INTEGRATION

We have successfully integrated our low-level recognizer into a higher-level sketch recognition system, LADDER [2]. Through this integration, we have been able to begin testing the accuracy of our complex fits beyond one line, one arc combinations. Although we have not formally evaluated higher degree complex fits, we have seen promising results as seen in Figure 9.

FUTURE WORK

We are pleased with the results we achieved thus far, but we still wish to continue testing our recognizer. In particular, we would like to test it against the recognizer from Yu [12]. We are mainly interested in determining how our complex interpretation results would compare with theirs. In our experiment we achieved 94% accuracy with the same types of shapes that their recognizer only recognized 70% of the time (arc/line combinations). Obviously these were not tested with the same data set. It would be interesting to see if their recognizer still has 70% accuracy with our data set.

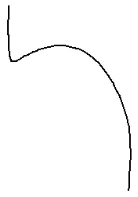

Figure 8. The only example to not have its correct interpretation returned at all (curve). Our classifier returned a one line, one arc interpretation, which could arguably be the interpretation of a human recognizer.

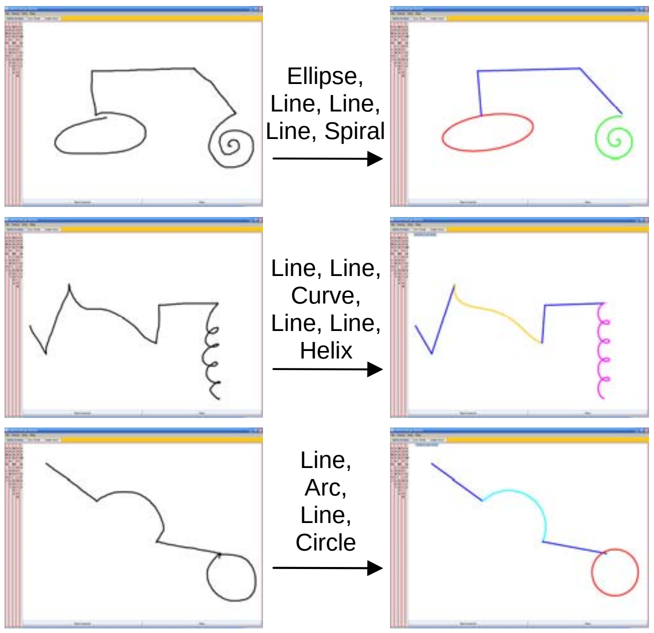
Ellipse, Line, Line, Line, Spiral
Line, Line, Curve, Line, Line, Helix
Line, Arc, Line, Circle

Figure 9. Examples of higher degree complex fits achieved through the integration of our recognizer into LADDER.

In addition to this, we would like to further test complex interpretations that include more than two sub-strokes. The integration of our recognizer into LADDER has allowed us to do preliminary testing, which seems to indicate promising results for higher degree complex fits; however, it has yet to be formally tested and evaluated.

We also want to explore the idea of "continuation strokes." As mentioned before, our goal is to produce sketch recognizers that place little constraint on a user's drawing style. Currently, we make the assumption that all primitive shapes are drawn with a single stroke. We would ultimately like to erase this assumption. We are currently looking into adding the concept of stroke continuation into our current recognizer. Stroke continuation refers to the act of a user drawing a low-level shape, stopping (by picking up the pen), and then later continuing the previous drawn stroke. Instead of recognizing a continuation stroke as two separate shapes, we ideally would want to merge the two strokes into a single shape.

Another area we would like to explore is creating a universal way to compare error in various shape interpretations. We are currently exploring the possibility of using a feature-based classifier that uses the values of the different geometric tests presented throughout this paper as a feature set. Current results look promising for the eight basic primitives, but we are unsure how a feature-based classifier will perform when given different variations of complex fits.

Our group is also interested coming up with better corner finding algorithms. The one used by this recognizer is simple, but does not always give a perfect polyline interpretation. One idea we would like to explore is the idea of "invisible corners" – corners that better approximate the interpretation of the stroke but that don't actually lie on the stroke itself. As a motivating example, we imagine a rounded rectangle (which most users will sketch when asked to draw a rectangle). In this case, choosing corners that are real stroke points will not be as good of a fit as those that could be estimated, for example, by finding the intersection of the best least squares line for each segment of the rectangle.

CONCLUSION
Because sketching is a very natural means of interaction between humans, many experts are looking into ways of integrating sketch recognition into traditional user interfaces. With this integration comes the need to develop robust and accurate recognizers. However, many low-level sketch recognizers struggle with the trade-off between the number of primitive shapes it recognizes and accuracy. In this paper we described a low-level sketch recognition and beautification system that uses two new features, along with a novel ranking algorithm. The system is capable of recognizing eight primitive shapes, along with complex shapes, with accuracy rates over 98.5%. These rates proved to be comparable to the current state-of-art low-level recognition systems that do not recognize as many primitives. Furthermore, through the integration of our recognizer into the high-level sketch recognition system, LADDER, we have seen promising results for fitting higher degree complex interpretations.

ACKNOWLEDGMENTS
Thanks to those in the 2006 sketch recognition class for their ideas as well as the other members of the SRL – Paul Logasa Bogen II, Amanda Coots, Henry Choi, Katie Dahmen, Mark Eaton, Brian Eoff, Pankaj Rajan, Aditya Ramgopal, Vijay Singh, Jennifer Weingarten, and Aaron Wolin. This work is supported by NSF:IIS Grant #0744150.

APPENDIX
Thresholds: Below are the thresholds used in the implementation of the system described in this paper. These thresholds were determined empirically through our initial training data set and are given to allow readers to reproduce our results.

A. 0.5	H. 10.25	O. 0.425	V. 0.1
B. 5.0	I. 0.0036	P. 0.35	W. 9.0
C. 70.0	J. 6.0	Q. 0.4	X. 10.0
D. 1.31	K. 0.8	R. 0.37	Y. 0.99
E. 0.16	L. 30.0	S. 0.9	Z. 0.06
F. 0.75	M. 0.33	T. 0.25	
G. 2.0	N. 16.0	U. 0.2	

Hierarchy: Our hierarchy is given below with shape interpretations at the top being added before shape interpretations at the bottom. Shape interpretations may appear multiple times in the hierarchy, but are only added once to our list.

1. All lines.
2. Arcs whose feature area error is less than the feature area of its polyline interpretation.
3. Polylines with very high DCR values[W] and low number of sub-strokes[X]. We use a less strict DCR threshold[J] if all sub-strokes passed the line test.

4. Non-overtraced circles whose feature area error is less than the feature area of its polyline interpretation. We do make an exception however. If the polyline test passed and the polyline rank is less than that of the circle (as determined by the ranking algorithm) then polyline is added in front of the circle interpretation. This exception does not apply to small circles[N].
5. Non-overtraced ellipses whose feature area error is less than the feature area of its polyline interpretation. As with circles, we add polylines that meet the conditions mentioned in part 4. Again, this would not apply to small ellipses[L]. A circle fit will also be added with the ellipse as an alternative interpretation.
6. Arcs not already added from step 2
7. Spirals that may have also passed an overtraced circle or overtraced ellipse test.
8. Circles (including overtraced) not added in step 3 (polyline condition still applies).
9. Ellipses (including overtraced) not added in step 4 (polyline condition still applies).
10. All helixes with scores less than the complex interpretation score. If the complex score is lower then it is added, followed by the helix.
11. All curves.
12. All spirals not added in step 7.
13. All other polylines.
14. If the interpretation list is empty at this point, or the top interpretation is a curve or polyline, then we execute a complex test. If the complex test returns an interpretation that contains all lines or polylines then we add a polyline interpretation. If not, then we compare the ranking of the complex fit with the ranking of the top interpretation (whether it is a curve or polyline). If the complex rank is less than the current interpretation rank then the complex interpretation is added at the front of the list. Otherwise, we add the complex fit to the end of the interpretation list.
15. Polyline is always added as a default interpretation (regardless of whether or not its test passed).

Corners: The goal of our corner finding algorithm is to determine a good polyline interpretation for the stroke. We begin by first trying to determine the neighborhood where a corner may lie. To do this we begin at the first point of the stroke (first corner) and iteratively choose the next consecutive point until we determine that the sub-stroke between the two points is no longer a line. To determine this, we do a quick line test of dividing the distance between the two points by the length of the sub-stroke. As long as that ratio is greater than a threshold[Y], then the sub-stroke is considered a line. Once we reach a point that violates the line condition we mark the previous point as a corner, update the current point to be the new first point and continue the test to find the next corner. After the initial test is run, we have a list of preliminary corners.

Next, we perform a round of merging to make sure that we don't have corners that are right next to each. To do this, we look at the points that are within the "neighborhood" (we used a percent threshold[Z] of all points) of each one of the corners. If a neighbor point is also a corner, then the two corners are merged into a single corner at the averaged index. If one of the corners to be merged is an endpoint then we simply remove its counterpart rather than performing a merge.

After finding the merged corners, we then analyze the neighborhood of each non-endpoint corner one last time and find the point in the neighborhood with the highest curvature (which should look more perceptually like the true corner). We replace each corner with the neighborhood point of highest curvature, and perform merging once more. We continue to merge until no change is made in consecutive merge attempts.

REFERENCES

1. Alvarado, C., Oltmans, M. and Davis, R. A Framework for Multi-Domain Sketch Recognition. In *Proc. of the AAAI Spring Symposium on Sketch Understanding*, AAAI Press (2002), 1-8.

2. Hammond, T. and Davis, R. LADDER, A Sketching Language for User Interface Developers. *Computers & Graphics* 29, 4 (2005), 518-532.

3. Kara, L.B. and Stahovich, T.F. Hierarchical Parsing and Recognition of Hand-sketched Diagrams. In *Proc. of the 2004 ACM Symposium on User Interface Software and Technology*, ACM Press (2004), 13-22.

4. Kim, D.H. and Kim, M.J. A Curvature Estimation for Pen Input Segmentation in Sketch-based Modeling. *Computer-Aided Design* 38, 3 (2006), 238-248.

5. Long, Jr., A.C., Landay, J.A. and Rowe, L.A. "Those Look Similar!" Issues in Automating Gesture Design Advice. In *Proc. of the 2001 Workshop on Perceptive User Interfaces*, ACM Press (2001), 1-5.

6. Long, Jr., A.C., Landay, J.A., Rowe, L.A. and Michiels, J. Visual Similarity of Pen Gestures. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2000), 360-367.

7. Morrel-Samuels, P. Clarifying the Distinction Between Lexical and Gestural Commands. *The International Journal of Man-Machine Studies* 32, 5 (1990), 581-590.

8. Rubine, D. Specifying Gestures by Example. In *Proc. of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press (1991), 329-337.

9. Saund, E., Fleet, D., Larner, D. and Mahoney, J. Perceptually-Supported Image Editing of Text and Graphics. In *Proc. of the 2003 ACM Symposium on User Interface Software and Technology*, ACM Press (2003), 183-192.

10. Sezgin, T.M., Stahovich, T. and Davis, R. Sketch Based Interfaces: Early Processing for Sketch Understanding. In *Proc. of the 2001 Workshop on Perceptive User Interfaces*, ACM Press (2001), 1-8.

11. Sutherland, I.E. Sketch Pad A Man-Machine Graphical Communication System. In *Proc. of the SHARE Design Automation Workshop*, ACM Press (1964), 6.329-6.346.

12. Yu, B. and Cai, S. A Domain-Independent System for Sketch Recognition. In *Proc. of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, ACM Press (2003),141-146.