WARNING: This document is now close to a draft, but it is very poorly written for the moment. Some pieces are missing as well.

Early Processing in Sketch Understanding

Tevfik Metin Sezgin and Randall Davis

Abstract

Freehand sketching is both a natural and crucial part of design, yet is almost totally unsupported by current design automation software. We are combining the flexibility and ease of use of paper and pencil with the processing power of a computer to produce a design environment that feels as natural as paper, yet is far smarter. One of the most basic steps in accomplishing this goal is converting the original digitized pen strokes in the sketch into the intended geometric objects, the early processing part of the task. In this paper, we describe a system which combines several sources of information to provide a robust early processing module for pen based interaction in general and freehand sketching in particular.

1 Introduction

Sketching is a very powerful, efficient, and natural way of expressing ideas. For example, imagine how hard it would be to teach physics without using sketches. Designers utilize freehand sketches extensively especially in the early phases of the design process. Unfortunately they end up having to sacrifice the utility of freehand sketching in exchange for the services that a computers offer, in part because most CAD programs don't support freehand sketching and they are meant to be used more in the later phases of design rather than the early design phase. Although designers often start sketching out ideas with pencil and paper, they end up having to move to a computer for detailed design, at which point the sketch on the paper is no longer accessible digitally, and the effort put in defining the rough geometry on paper is wasted.

We provide a system where users can sketch naturally. We digitally capture sketches and utilize them in aiding the design process. For example, a designer developing an alternate auto-reverse mechanism for a walkman can start the task by sketching out parts of the mechanism, verbally give the computer a description of the device and test his idea by telling the computer to simulate the device. Depending on the result of simulation, the designer can modify the initial sketch. A framework like this has the advantage of importing device geometry information specified in the initial sketch to the computing environment as early as possible. It may also aid extraction of design rationale which is usually more trouble to capture and document than it is worth.

As humans, we are very good in perceiving freehand sketches even if they are very rough. From the computer's perspective, however, strokes forming a sketch are just collections of pixels. In order to make anything useful with a sketch, we first have to convert the input strokes into the corresponding *low level geometric primitives*, such as lines, ovals, rectangles, arbitrary poly-lines, curves and their combinations. We can define *early sketch processing* as a mapping from raw strokes to the low level geometric primitives described above. Here are some of the advantages of a system allowing this conversion:

- It allows users to sketch freely, and interact with the computer without having to deal with cumbersome pallets and tool selection.
- Exploits direct manipulation, which is desirable in sketching and design tools. For example, simple sketching curves as opposed specifying them via end points and control points is easier, more natural, and intuitive.
- Makes application of cleanup techniques possible.
- Since strokes containing thousands of points are approximated by primitives defined by their geometry, the storage requirements are reduced tremendously. For example, a line is represented only by its end points, and a curve is represented by Bézier curves with two end points and two control points.
- A geometric primitive contains less information about the original stroke, but tells more about what the stroke was meant to be. This is more meaningful and valuable from a programmer's point view. For example, a circle with radius 10 centered at (30,25) is more informative than hundreds of pixels digitized while user drew the circle.
- Geometric primitives are easier to manage and are the preferred format for other programs used

as back-end engines. For example a drawing program or a mechanical simulator requires the input to be geometric primitives rather than raw strokes.

Early processing can be thought of as combination of *approximation*, *recognition*, and *beautification* as described below:

- Approximation is the task of fitting low level geometric primitives to a given stroke. At this stage the emphasis is on approximating the stroke with as few data points as possible. On one hand a certain error criteria must be met, on the other hand over-fitting should be avoided.
- Recognition comes into play when fitting a more specific geometric primitive is meaningful. For example, a stroke approximated by a four sided polygon can be converted into a square or a rectangle if it is sufficiently close to one.
- Beautification refers to adjusting the slopes of line segments in the fits generated by the approximation layer in order to preserve relations such as parallelism, orthogonality and contact.

To this point, we have provided the motivation for doing early processing, and gave some definitions. Now we will describe an early processing system which also does some visual syntax parsing, and gesture recognition. Later we will discuss, some of the difficulties encountered.

2 System description

2.1 Context

We chose early mechanical design process as our target domain because designers make extensive use of sketches at this stage. Our system is part of the sketch understanding system ASSIST (cite ASSIST related work here), developed by other members of our research group. Using ASSIST, a designer can sketch a mechanical device and explain how it works using speech, allowing the computer to construct a better model of the device mechanics.

2.2 Vertex detection techniques

2.2.1 Direction change

Among major challenges in stroke approximation is detecting the vertices. In the first glance one would expect the change in direction to be a good indicator of corners. Occasionally direction change alone will generate reasonable fits[1]. The basic idea is computing the angle[2] between the horizontal axis and the tangent to the curve drawn by the user at each sample point. If we let $p_i = (x_i, y_i)$ be the position of the i^{th} input point for $0 \leq i < n$, the direction d can be computed as:

$$d_i = atan2(y_{i+1} - y_i, x_{i+1} - x_i) \text{ where } 0 \leq i < n-1$$

In order to compute the change in direction, we need the timing data for each p_i. Let t_i be the timestamp for p_i. Now we can define change in direction as:

$$\frac{dd}{dt}|_i = \frac{d_{i+1} - d_i}{t_{i+1} - t_i}$$

The d and dd/dt graphs for the rectangular stroke in figure 6 are in figure 2. In this case the peaks of the dd/dt function are good indicators of corners. One can compare the function value to the average of its absolute value. Simple binary search for maxima gives us a set of indices $i_0 ... i_{k-1}$ pointing to the p_i that are most likely to be vertices.

Although using dd/dt information gives us good fits for shapes drawn rather carefully, it fails in some cases where the input is drawn in a sloppy fashion. In this case, the fit may contain some points which aren't actual vertices or even worse some vertices that were

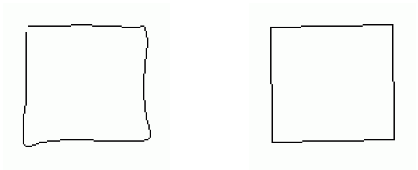

Figure 1: A stroke describing a square on the left, and the output produced on the right.

Figure 2: Direction graph for the square stroke, and its time derivative on the right.

[1] Informally, we will define a 'fit' as a poly-line for the time being. Later we will extend this definition.

[2] As we will discuss later on, we prefer using orthogonal distance regression rather than a point-wise interpolation for obtaining direction.

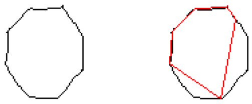

Figure 3: A stroke on the left, and the polygonal fit generated using the dd/dt data overlayed on the original stroke on the right.

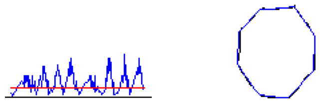

Figure 4: Speed graph for an octagon, and the generated fit on the right.

supposed to be included in the fit may be excluded. Our experience suggests that users find it very annoying if the system misses a vertex. Since our aim in the first place is providing a system where users can sketch freely, we have to handle sloppy drawings as well. Presence of noise in the direction data makes generating good fits quite a hard task, so we look at other sources of information. As we explain in the next section, speed data is a good alternate source of information for generating better fits.

2.2.2 Speed change

Direction change technique for generating fits is especially unsuitable for a particular subset of poly-lines where the direction change in certain vertices is subtle although the user has clearly intended to have a vertex. For example a convex polygons with n vertices will have an average direction change of π/n per vertex. As n goes up the direction change will become more subtle, making vertex detection a harder task, especially the stroke contains vertices where dd/dt is relatively high.

Figure 3 illustrates this phenomena. Here the direction fit missed three vertices where dd/dt was too small to signal a vertex. Of course one could loosen the vertex detection criteria catch these points, but this would cause many other non vertex points to be classified as vertices. We eliminate this problem by looking at pen's speed. We define speed as follows:

$$v_i = \frac{dx}{dt}|_i = \frac{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}{t_{i+1} - t_i}$$

Technique used in generating the speed fit is similar to that of direction fit, except now vertices are found by looking at the minima of the function. Figure 4 shows the speed data and the speed fit generated.

Although exploiting the dynamics of pen's motion provides a more robust fitting method, it is not entirely sufficient by itself. In particular, poly-lines formed by a combination of very short and long line segments are sometimes poorly approximated because the maximum speed reached along the short line segments isn't sufficiently high to trigger detection. This problem arises when drawing thin rectangles which appear frequently in mechanical devices. The solution to this problem is combining dd/dt and speed information as we will explain in the next section.

2.2.3 Generating hybrid fits

In order to come up with better fits, we utilize both direction change and speed data. The hybrid fit generation procedure assumes two fits F_d and F_s (the dd/dt and speed fits). Hybrid fit generation has three stages:

Computing vertex certainties

Prior to hybrid fit generation, the vertices in F_d and F_s are assigned certainties C_d and C'_s indicating how likely it is for a particular vertex $v \in F$ to be a real vertex. Certainties for vertices in F_d do not necessarily have to be comparable to those in F_s, but within each fit, vertices with higher certainties should be more likely to be real vertices. For $v = F_{d_i}$ the corresponding certainty C_{d_i} is taken to be $|d_{j-k} - d_{j+k}|$ where j is the index of v in the input stroke S, and k is some small integral value defining a neighborhood. In other words C_{d_i} is a measure of absolute change of direction in a local neighborhood defined by k. For $v = F_{s_i}$ the corresponding certainty C_{s_i} is defined by $\max_{0 \le j < n} v_j / v_i$ where $n = |S|$. Intuitively this captures how much the pen slowed down at v. Finally, once the certainties are computed, we sort the vertices in each fit by their certainties moving the vertices with highest certainties to the front.

Computing a set of hybrid fits

The initial hybrid fit F_{h_0} is taken to be the intersection of F_d and F_s. A number of other fits are generated by appending F_{h_i} by vertices selected from two sets $F_d{}'$, $F_s{}'$ where

$$F_d{}' = F_d \setminus (F_d \cap F_s)$$

$$F_s{}' = F_s \setminus (F_d \cap F_s)$$

The general rule for creating a new fit $F_{h_{i+1}}$ is:

$$F_{h_{i+1}} = F_{h_i} \cup \{v\}$$

where $v = \begin{cases} x|x \in {F_d}' \wedge x \notin F_{h_i} & \text{if} \quad i \equiv 0 \pmod 2 \\ x|x \in {F_s}' \wedge x \notin F_{h_i} & \text{if} \quad i \equiv 1 \pmod 2. \end{cases}$

Selecting the optimum fit from F_{h_i}

So far, we discussed how to generate a class of fits to a given stroke S, but haven't defined a measure for describing the goodness of a fit. Least squares error works extremely well in this case. For each fit, an error ε_i is computed as follows:

$$\varepsilon_i = \frac{1}{n} \sum_{s \in S} ODSQ(s, F_{h_i})$$

$$OD(s, F) = LPDSQ(s, u, v) \text{ where}$$

$$u, v \in F, \; s = S_k, \; u = F_l = S_p,$$

$$v = F_{l+1} = S_q, \text{ and } p < k < q$$

Here $ODSQ$ stands for *orthogonal distance squared*, and $LPDSQ$ stands for *line to point distance squared*.

Usually fits with more vertices have smaller ε_i. This is actually natural because the more points are used, the better a stroke can be approximated. In fact S itself is a perfect fit, but there is a tradeoff between choosing a fit with many vertices that approximates the input stroke very well, and a poor fit with only a few vertices. This tradeoff is illustrated in figure 5. Using least squares as a fitness criteria lets us choose fits with ε_i no greater than a desired error. In our case, we set an error limit and designate F_{h_i} with the least number of vertices satisfying the criteria as our final fit F_f.

2.3 Handling curves

Previous section explains how to get robust poly-line fits to data. As long as the input consists of poly-lines the above scheme works very well. On the other hand, if the input is a curve or even worse if it is some combination of curves and poly-lines some more work has to be done. Handling curves has two sub-tasks to be accomplished. First regions of curvature should be identified, and then faithful Bézier approximations should be generated. Having generated a poly-line approximation to S, the task of detecting areas of curvature is simplified dramatically. One particularly good method is that of looking at accumulated direction change along S. One can actually integrate the absolute direction change along S, but a much better technique is comparing the Euclidean distance l_1 between each consecutive vertex $u, v \in F_f$ against the integrated distance l_2 between the corresponding vertices in S. In the linear regions of S, the ratio l_2/l_1 is very close to 1, and in curved regions this ratio becomes significantly higher than 1. Once we detect the regions of curvature, we approximate those regions with Bézier curves. A Bézier curve is defined by two end points and two control points. If we let $u = S_i$ and $v = S_j$ and $i < j$ be the end points of the segment in S to be approximated, the control points are computed as follows:

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

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

$$k = \frac{1}{3} \sum_{i \le k < j} |S_k - S_{k+1}|$$

where $\hat{t}_1$ and $\hat{t}_2$ are the unit length tangent vectors pointing inwards the curve segment to be approximated. The $1/3$ factor in k controls how much we scale $\hat{T}_1$ and $\hat{T}_1$ in order to reach the control points. The summation in computation of k is simply the length of the chord between S_i and S_j. This constant was computed empirically, but works very well for hand sketched data. Surprisingly the same constant was also used in [6].

Since we are already using the optimal poly-line fit as the basis to our final fit[1], we only need to measure and minimize the errors of the Bézier curve segments. As it was the case in fitting poly-lines, we use least squares technique for evaluating the goodness of a fit, so the orthogonal distances should be computed from the fit to each S_i on the input stroke being approximated. However, since Bézier curves are special third degree polynomials computing the minimum distance from an arbitrary point to the curve involves minimizing a sixth degree polynomial which is equivalent to solving a fifth degree polynomial. There is no practical closed form solution for fifth degree polynomials in general. On the other hand a numerical solution approach is be costly and heavily dependent on the goodness of the initial guesses for roots. The alternate method of subdividing the approximating Bézier curve and then computing the error makes it possible to compute a fairly good measure of the error in $O(n)$ where n is $|S|$. An in depth treatment of finding the closest point on a curve to an arbitrary point can be found in [1]. If the computed error is higher than the maximum error tolerance, Bézier approximations with high errors are recursively divided into two smaller pieces introducing two control points and an extra vertex, thus generating a better fit. Another possible approach would be doing gradient descent to minimize the error before attempting subdivision, but this involves multiple calls to the error computation routine. Also in most cases a subdivision is necessary for satisfactory results.

[1] Here, we extend the notion of fit to include Bézier curves and their combinations with poly-lines.

2.4 Object Recognition

So far, we described a process for generating faithful representations of hand-sketched data using lines and curves. Although this process by itself serves as a nice abstraction there are cases where it becomes more desirable to designate the input as belonging to a more specific class. For example the stroke in figure ?? can be approximated by a series of curve segments, but in this case a circle is more meaningful. Another example can be seen in figure 6. Although the ends points of S don't touch, the output is a perfect rectangle rather than a polyline with four sides. We call this task of coming up with a more specific mapping for a stroke as *object recognition*.

In our system, a template matching approach was used for recognizing circles, squares and rectangles – again making use of least squares error technique. In addition we recognize some domain specific objects shown in figure 7.

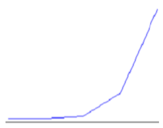

Figure 5: A graph showing the least square errors of the hybrid fits sorted in descending order by the number of vertices they have.

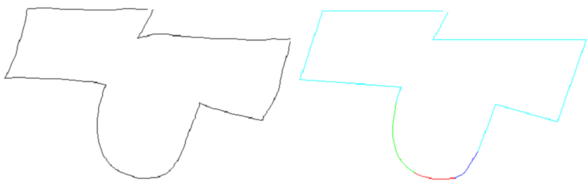

Figure 6: A stroke and the corresponding fit generated. The fit contains a number of lines and three Bézier curves approximating the curved region.

Figure 7: Strokes for a spring and a ground symbol and the outputs after recognition.

2.5 Beautification techniques

Beautification refers to the adjustments made to the approximation layer's output to make it look as it was intended. Sketches are ambiguous and despite their high expressive power even we sometimes fail to recognize what was intended in a particular sketch. It is therefore convenient to adjust the slopes and lengths of the line segments so that lines that were meant to be parallel or the same size actually appear so. This task is accomplished by quantizing the slopes and the lengths of lines in the final fit for S. The intuition behind this idea is that in a given stroke (or even in a sketch) there are certain clusters that the slopes and lengths fall into. We detect these clusters and place the objects into these clusters by their features. This quantization idea is very similar to the color-map creation process in computer graphics except now we are building length-maps or slope-maps.

2.6 Evaluation

3 Difficulties

One of the major difficulties that arise when dealing with sketches is noise. Freehand sketches are not clean to start with. Jittering of the hand combined with the information loss during digitization makes matters worse. It is surprising how radically the slope of a seemingly smooth line varies throughout the stroke. Although there are well known techniques for

Figure 8: An overtraced oval on the left, and the output produced on the right.

filtering noise, they usually don't perform as expected because the number of data points is limited. Also in some cases filtering smoothes sharp edges as an undesired side effect. Throwing away some of the data points (for example every other point) was preferred by some for reducing the noise [6]. The down side of discarding points is that it causes loss of information. Instead we preferred using orthogonal distance regression to fit lines to the points within a sliding window. This technique proved to be effective although it has a relatively high computational cost. An detailed discussion of difficulties and techniques in orthogonal distance regression can be found in ODRPACK user guide [7].

Overtracing is another complication. Our system handles limited amount of overtracing. Figure 8 illustrates an instance of overtracing, and the output.

4 Related work

The Phoenix architecture [6] built by Schneider has some of the same motivations behind. Phoenix was designed to aid interactive curve specification via sketching rather than providing approximations for arbitrary strokes which may contain poly-lines as well. Although some amount of support for automatic vertex detection was provided sometimes the system failed to do so and the user was forced to specify the vertices manually. We believe this kind of need for user intervention makes sketching tools hard to use. Schneider also points out the problems introduced by noisy data and uses Gaussian filters to smooth the data. This technique didn't prove to be as useful in our system mainly because we support arbitrary stroke approximation and filtering makes vertex detection even harder.

In [8] Lynn Eggli describes a system for sketching with constraints. This system supports geometric recognition for simple strokes. In addition, implements a constraint maintenance system, and allows operations such as sweeping and extrusion for generating 3D solid geometries. The set of primitives supported is limited. Each stroke is interpreted as a line, arc or as a Bézier curve. More complex shapes are formed by combinations of these primitives forcing the user to lift the pen at the end of each primitive stroke, inhibiting the natural feeling of sketching. High emphasis is given to getting a very precise fit to the input stroke, and a sequential fitting scheme is used where the system tries to fit a line, an arc and a Bézier respectively. For each primitive, the fitting method is iterative in nature. Fitting process begins with an approximation and then an iterative search starts looking for better fits by playing with the parameters of the initial fit. The error measure used for fitting lines is computed along the vertical axis for lines with $-1 < m < 1$ and the horizontal axis otherwise where m is the slope of the line.

Michael Banks and Elaine Cohen describe a system for generating realtime spline curves from interactively sketched data [9]. They approach the problem with computer graphics techniques.

Zeleznik et al. has designed a mode based stroke approximation system. His system uses simple rules for detecting the drawing mode and interpretation is done according to the current mode. The basic limitation of this system is that the user has to draw objects in pieces making it an unnatural interface. Switching modes is also unnatural and awkward, for example, in order to draw a curve the user clicks the mouse button pauses for a while and then starts drawing the curve. They actually touch this problem in the future work section, and suggest better mode switching method using modifier keys or by exploiting pressure sensors in pens, but we believe this is still unnatural. The system is purely gestural with different primitives such as cones, spheres and quad curves adding to the complexity compared to the simplicity of sketching.

Several people have worked on beautification focusing on exploiting features such as parallelism, perpendicularity, congruence and symmetry. Igarashi et al. [10] describe a system that combines beautification with constraint satisfaction. Bolz's [11] system on the other hand supports semi-automatic beautification as an extension to a conventional graphics editor.

Luke Weisman [5] and Manoj Muzumdar [4] have also worked on some low level recognition but their systems doesn't support arbitrar strokes.

itemize

Cocktail Napkin

Discussion about:

- Contrast with gesture recognition.

- Contrast with OCR, HW recognition, spatio-temporal analysis, learning for gestures via HMMs, TDNNs and ART (adaptive resonance theory) It can be mentioned that our approach has the advantage of being able to give intuitive reasons about why something failed or why a rectangle got recognized as a circle.

5 Future Work

On the larger scale, our system is part of a sketch understanding system ASSIST (cite ASSIST related work here), developed by other members of our research group. Using ASSIST, a designer can sketch a mechanical device and explain how it works using speech, allowing the computer to construct a better model of the device mechanics. Using all the information gathered ASSIST generates a model that can be simulated by Working Model 2D, an off the shelf simulator, allowing the designer to see the device behavior and make the necessary modifications. For example, a designer developing an alternate auto-reverse mechanism for a walkman can start the task by sketching out parts of the mechanism, verbally give the computer a description of the device, and test his idea by telling ASSIST to simulate the device.

Potential future directions may include building sketch understanding systems for domains other than the mechanical engineering domain, such as software design or chart design. Also combining handwriting recognition and natural language understanding with sketching and other kinds of interaction to support tasks like design rationale capture remains as a challenging and interesting task.

6 Conclusion

Noting that we can't have intelligent spaces without intelligent paper, we have built a system mapping raw strokes sketched by users into approximating geometries using multiple sources of information. With this system, users can sketch on an LCD tablet as if they were drawing on paper and have the computer understand the low level geometry enabling a more natural interaction with the computer. Our system also makes it possible to utilize computers in the design process as early as in the brainstorming stages.

Acks:

Research Support: This work is supported in part by the Ford/MIT Collaboration under the Virtual Engineering project, in part by the Lockheed Martin Space Operations Company under CSOC Contract NAS9-98100, and in party by the MIT Oxygen Collaboration.

References:

[1] Nicholas J. Redding Implicit Polynomials, Orthogonal Distance Regression, and Closest Point on a Curve *IEEE Transactions on Pattern Analysis and Machine Intelligence* , pp. 191-199, 2000

[2] M.D. Gross. Recognizing and Interpreting Diagrams in Design. *2nd Annual International Conference on Image Processing*, 1995, pp. 308-311

[3] M.D. Gross and E. Do. Ambiguous Intentions: a Paper-like Interface for Creative Design. *Ninth Annual Symposium for User Interface Software and Technology*, pp. 183-192. Marc Brown and Ramana Rao (eds.), Seattle, 1996.

[4] M. D. Muzumdar ICEMENDR: Intelligent Capture Environment for Mechanical Engineering Design. Master's Thesis, MIT, 1999

[5] Luke Weisman A Foundation for Intelligent Multimodal Drawing and Sketching Programs. Master's Thesis, MIT, 1999

[6] Philip K. Schneider Phoenix: An Interactive Curve Design System Based on the Automatic Fitting of Hand-Sketched Curves. Master's Thesis, University of Washington, 1988

[7] Paul T. Boggs, Richard H. Byrd, Janet E. Rogers and Robert B. Schnabel ODRPACK User's Reference Guide for ODRPACK Version 2.01 Software for weighted Orthogonal Distance Regression June 1992

[8] D. Lynn Eggli Sketching with Constraints Master's Thesis, The University of Utah, 1994

[9] Michael J. Banks and Elaine Cohen Realtime Spline Curves from Interactively Sketched Data SIGGRAPH, Symposium on 3D Graphics, 1990, pp 99-107

[10] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, Hidehiko Tanaka Interactive Beautification: A Technique for Rapid Geometric Design UIST 97 pp 105-114, Banff, Alberta, Canada

[11] Dieter Bolz Some Aspects od the User Interface of a Knowledge Based Beautifier for Drawings Intelligent User Interfaces, 1993, pp 45-52

[12] Robert Zeleznik, Kenneth P. Herndon and John F. Hughes SKETCH: An interface for Sketching 3D scenes Proceedings od SIGGRAPH'96 Aug 1996, pp 163-170