Usability of a Pen Interface for Sketching Graphs

by Robin Stewart
May 18, 2007

1. Introduction/Motivation

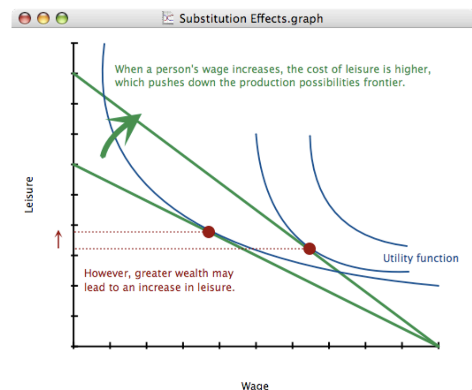
Pen-based interfaces have been created for a wide range of application domains, with varying levels of success. Although these interfaces have been widely touted as an intuitive way to interact with computers, it is difficult to predict which domains will really benefit from pen interaction. The task of sketching line graphs seems a likely candidate, since most students perform this activity regularly using pen and paper. Inspired by recent advances in pen stroke segmentation, I implemented a stroke recognizer that is built on related work but is specifically designed to recognize the types of lines used in line graphs.

However, it is unclear whether pen stroke recognition is really the best interface for creating line graphs on a computer. Although it is a familiar interface for people used to paper, it is not necessarily the most efficient technique: there are a number of alternate interface possibilities which further constrain the line being drawn and allow the user to bypass the ambiguity inherent in free hand-drawn strokes. To find out which type of interface is most usable in this domain, I implemented three versions of a software application called Graph Sketcher and performed a comparative usability study. The results of the study indicate that the more constrained, lower-tech interfaces are often preferable to full stroke recognition. I posit that this result extends beyond my particular implementation because it reflects the number of constraints inherently necessary to specify a curved line.

2. The Interfaces

In this section, I describe the base application and the modifications that I developed for performing stroke recognition.

2.1. Graph Sketcher



Graph Sketcher is a Mac OS X software application that I started working on when I realized that sketching economics graphs (such as the figure at right) using pencil and paper was much easier, faster, and more intuitive than using any existing software programs. Illustration programs are very flexible but require the user to tedi-

ously draw each axis tick mark and label; it is also hard to position elements precisely in a numeric grid. Mathematics and statistics software packages are very precise and tend to draw axes and labels automatically, but they *require* the user to enter most items numerically and the output graphs are difficult to annotate. By contrast, simply sketching a graph with pencil and paper is fast, very flexible, and as precise as you need it to be -- but the results are less professional, harder to read, and cannot take advantage of digital editing, sharing, search, and retrieval technologies.

What sort of interface should the ideal graph sketching program have? Ideally, we'd like to take the best of both worlds: the simplicity and flexibility of sketching, plus the precision and scalability of computer-recognized content. Two plausible approaches are:

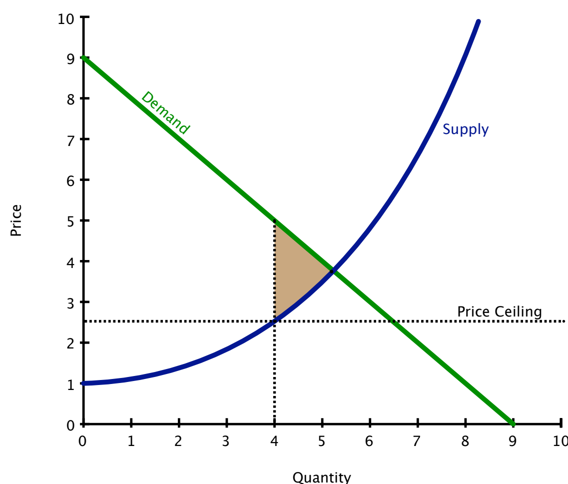
1. Based on the success of pen and paper, the program could imitate a sketchpad and interpret the marks as graph elements such as lines, points, axes, and labels.

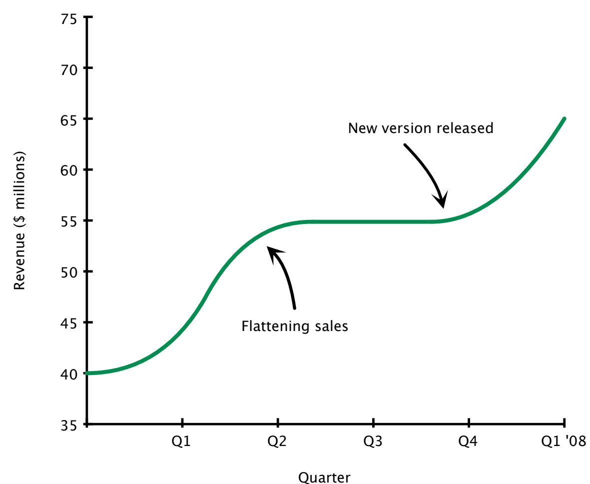
2. Based on the success of "direct manipulation" interfaces, the program could provide all the necessary graph elements and allow the user to add, select and arrange them as desired.

The existing Graph Sketcher program is based mostly on the direct manipulation approach. Still, the manipulation *techniques* resemble sketching on paper. To draw a line, the user chooses the "draw" tool and simply drags the mouse (or stylus) from the line's start to its end. The line can later be resized or curved by dragging appropriate handles.

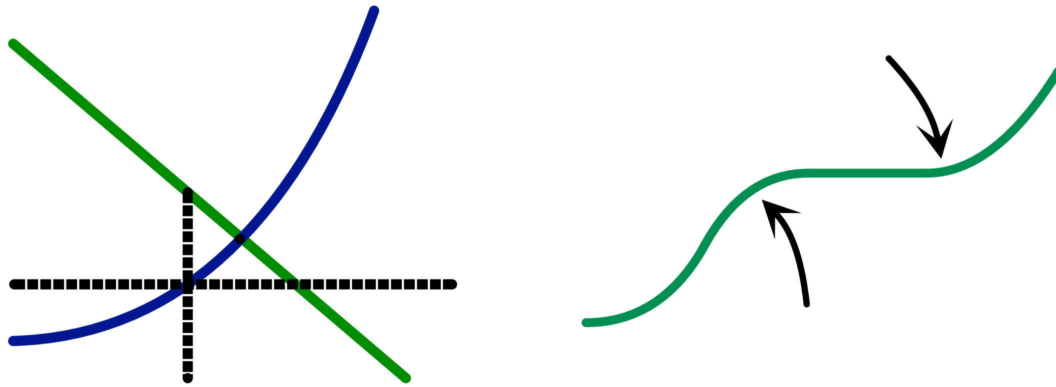
2.2. Graph "Sketchier": a pen-based interface

To move further towards a pen-based computing approach, I implemented a stroke recognizer specifically designed for line graphs. To design this domain-specific recognizer I first looked at the kinds of lines that are drawn in common line graphs. This was based on my observations of the types of graphs that users of Graph Sketcher ask about when they email me with questions or comments. I found that the following two graphs contain the representative line types:

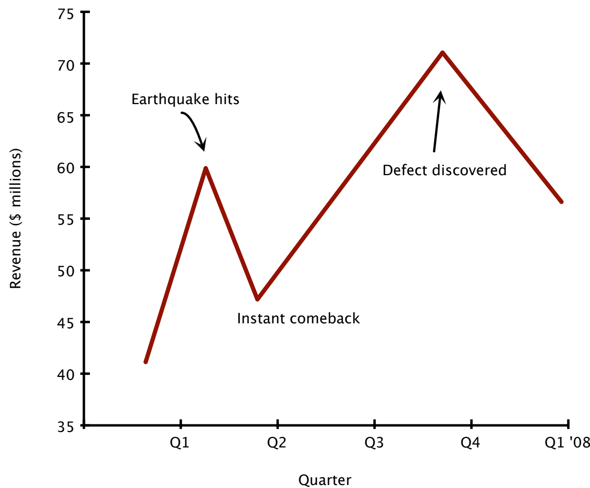


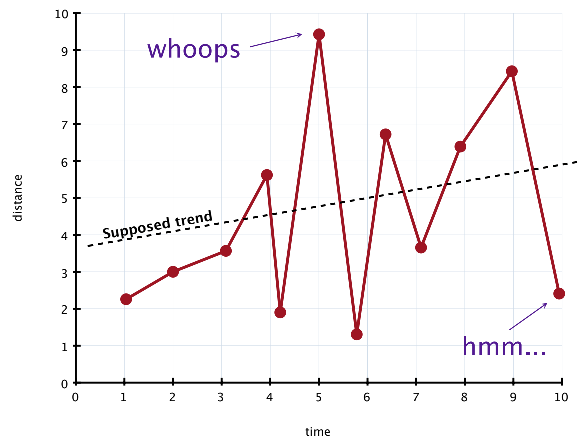


If we focus in on just the lines being drawn, we can abstract away to the following picture:



There are three categories of lines represented: straight lines, arcs, and more complex, smoothly curving lines. Conversely, notice that jagged lines are **not** commonly drawn. There are several likely reasons for this. First, real world trend lines don't usually have such discontinuities as are displayed in the first figure below. Second, for the purposes of rough forecasts or trends, smoothly curving lines tend to simply look better. Finally, although jagged lines often accompany noisy data (second figure below), such lines are not drawn by hand — they are plotted automatically by the software.





Based on these observations, I designed the stroke recognizer with several key constraints in mind:

1. It should be able to recognize smooth changes in curvature. Clearly, in order to recognize the shape of the complex green trend line, the recognizer must be able to detect changes in curvature despite the lack of sharp corners.

2. It does not need to perform corner detection. For the rare cases when users do want jagged lines, they can simply lift the mouse button or pen momentarily and start a new stroke.

3. It should not over-fit noisily drawn strokes. Many such strokes are probably intended to be long, straight segments or gently curving arcs. If recognized as such, they will look smoother and will be easier to adjust later.

This stroke recognizer enables an interface where the user can draw a stroke of arbitrary complexity which is interpreted as a series of segments. As in the original interface, the user can subsequently go back and adjust all aspects of the resulting segments. Alternatively, they can delete the recognized line and try sketching it again.

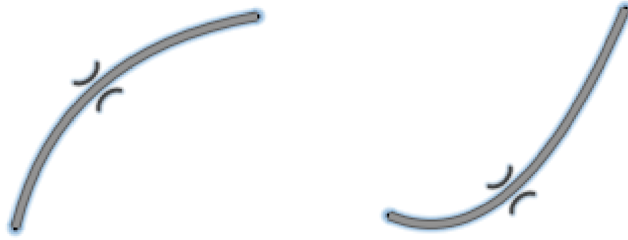
2.3. A medium-constrained approach: arcs

There is plenty of middle ground between the line-constrained original interface and the full stroke segmentation interface. To explore this intermediate territory, I implemented a third version of Graph Sketcher. The drawing process looks exactly the same as it does with the stroke recognizer interface — the raw pen stroke still appears on the graph surface as the user draws — but in this case each stroke is constrained to be a single line segment. When the mouse button is lifted after drawing a stroke, a very simple recognizer decides whether the stroke should be straight or arched, and in the latter case how far and in what direction the arc bulge should go.

2.4. Shared functionality

All three interfaces support several other important features. First, the ends of drawn lines always snap to existing points and points on the "grid" (defined by axis tick marks). This allows lines to be easily extended by drawing from an existing endpoint out to a further location. Second, straight line segments become horizontal, vertical, or 45-degree diagonal if the underlying stroke is nearly so. This makes it much easier to draw perfectly horizontal, vertical, or 45-degree lines in all three of the interfaces.

The three interfaces also share common controls for making adjustments to lines. All line segment endpoints can be selected and freely dragged around. Entire line segments too can be moved in this way. Multiple segments can be selected and adjusted together by holding down the "shift" key and clicking each component. To change the curvature of a line segment, the user moves the cursor over the segment and a "curve point" control appears (see first figure below). Dragging the control point moves the "bulge" of the curve around. All of these changes are displayed in real time, making it easy to fine-tune adjustments.



An example interaction with one of the interfaces is as follows. The user goes into draw mode by clicking the toolbar button or holding the “ctrl” key. They then draw the line they want (with results varying depending on the interface being used). Switching back into the “modify” mode, the endpoints and curvatures can then be adjusted to perfection. Alternatively, the user can stay in draw mode, delete the whole line, and try drawing it again.

3. Implementation

I designed and implemented the entirety of Graph Sketcher, so I had full access to the source code and a full understanding of its underlying model-view-controller architecture. As mentioned above, the original Graph Sketcher interface already existed at the time of this project and did not need significant modification. To support the more advanced pen-based interfaces, I implemented a number of additions to the software (comprising about 1,000 additional lines of Objective-C code).

To support pen stroke recognition, I first implemented a stroke-capture mechanism within Graph Sketcher. When the draw tool is selected and the mouse is dragged, the system checks the cursor location and the time using the finest temporal resolution possible: once per cycle of the event loop. However, data points are only recorded if the cursor falls on a different pixel than the previous data point. Using this method, a typical sketched arc contains about 30 to 100 digitized points. The current stroke is also displayed on the graph as it is being sketched. Once the mouse button is released, the stroke recognizer is invoked and the raw stroke is replaced with a smooth, recognized line. There is no discernible delay; even with very long and complicated strokes, the full segmentation algorithm finishes within two milliseconds on a 2.16 Ghz Macbook Pro. This low latency actually makes it possible to view the recognized line in real time; this functionality proved useful for iterating the stroke recognition algorithm, but is disorienting for regular program use. (A version of Graph Sketcher with this functionality enabled is provided in the supporting materials.)

3.1 Arc recognition

For the simple arc-based recognizer, no segmentation is needed. This recognizer simply calculates the perpendicular distance of each ink point from the diagonal between the stroke’s start and end. The furthest ink point (above the black line in the figure below) is chosen as the loca-

tion of the curve point, resulting in a single smooth curve (shown in blue). If the distance of the curve point from the diagonal is less than a set proportion of the diagonal length, the segment is made perfectly straight instead. Based on my own experimentation with trying to draw straight lines, I set this cutoff at 8% of the diagonal length.

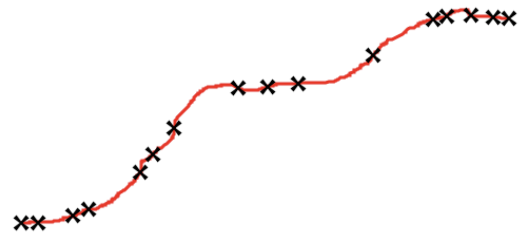
3.2 Full stroke segmentation

Substantially more machinery is necessary for the full stroke segmentation interface. Recall from section 2 the three constraints on the stroke recognizer motivated by observation of commonly drawn lines:

1. It should be able to recognize smooth changes in curvature.
2. It does not need to perform corner detection.
3. It should not over-fit noisily drawn strokes.

Constraints 1 and 2 suggest that segmentation should be based on the direction of curvature rather than the presence of sharp corners. To accomplish this, my recognizer starts by computing the arc length, smoothed slopes, and smoothed curvature at each ink point, following the method in Stahovich (2004). I used a window size of 7 ink points on either side (15 total) for computing the slopes with a linear regression, and 10 ink points (21 total) for similarly deriving the curvature. These window sizes were settled on during early experimentation; their exact values do not greatly affect the final accuracy of the segmentation.

Using the smoothed curvature values, the algorithm sets candidate segmentation points everywhere the curvature changes sign (resulting in the figure at right). As you can see, if the recognizer stopped here it would create far too many segments. Constraint 3 above specifies that as few as possible segments should be generated, while still doing a good job of fitting the curve. So I needed a way of pruning down the set of candidates.



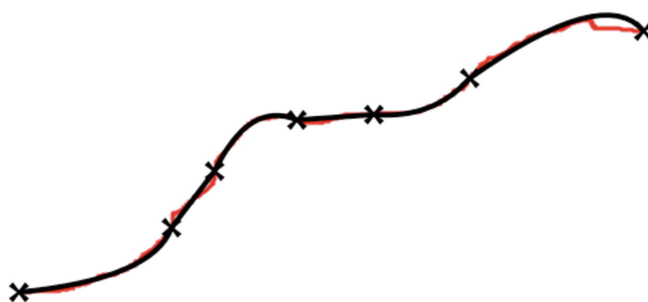
To determine whether neighboring segments should be merged, the algorithm computes the **adjusted total curvature** (ATC) of each proposed segment. This measure adds up the curvature values at each point in the proposed segment, multiplies this sum by the segment's arc length, and divides by the number of ink points in the segment.

adjusted total curvature = sum(curvatures) * arc length / number of ink points

The rationale behind this measure is as follows. The sum of the curvature values determines how substantial the curve is, and in what direction (positive or negative curvature); for noisy, squiggly lines, the curvature values will roughly cancel each other out. Multiplying by the arc length boosts the score for curves that are longer, so that it is more likely to retain long segments than shorter ones. Finally, the slower the user draws a line, the more ink points get recorded, which artificially increases the summed curvature; dividing by the number of ink points in the segment cancels this effect.

Based on my own experimentation with a range of line shapes and drawing speeds, and observation of one other user, I set a threshold of 30 as the cutoff between labeling segments as straight vs. curved. This means that segments with an adjusted total curvature above 30 are deemed to be positively curved, those below -30 are deemed negatively curved, and those between -30 and 30 are deemed straight. If two subsequent proposed segments have the same 3-way line type (positive, negative, or straight), then they are merged into one segment.

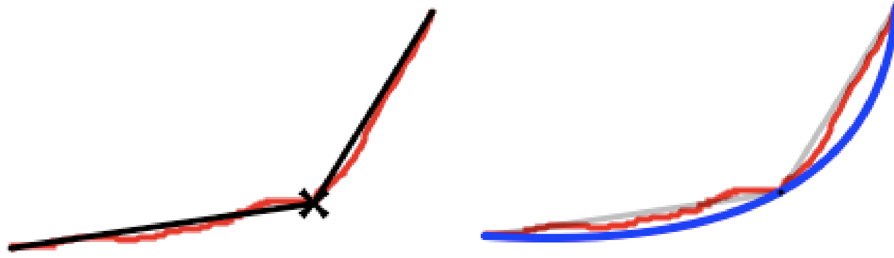
This process of pruning is continued recursively until no more segmentation points can be removed. Finally, the "arc recognition" algorithm from the previous interface (see section 3.1) is used to determine the actual curvature of each displayed segment. The resulting final segments are shown at right (along with the segmentation points as X's and the original input stroke in red).



During further self-experimentation, I noticed that a peculiar situation (slightly exaggerated in the figure below) would sometimes arise. Although the user intended to sketch a straight line, the algorithm would segment the stroke into multiple pieces (below, a piece with negative curvature followed by one with positive curvature). However, the recognizer would output a line which was essentially straight but with a segment point in the middle that was baffling to users:

This situation occurred because the negative and positive pieces both had an ATC value outside the cutoff (so they did not get merged) but they were below the "bulge percentage" cutoff for the arc recognizer (so they were drawn as straight segments).

To eliminate this situation, I added an additional directive to the recursive pruning algorithm: if neighboring segments are both judged as straight by the arc recognizer, the segment point between them should be removed. It's possible that the resulting merged segment will end up being drawn as curved (as in the example below; original segments are black, merged curve is blue), but this behavior is desirable given our constraint of smoothly curving lines.

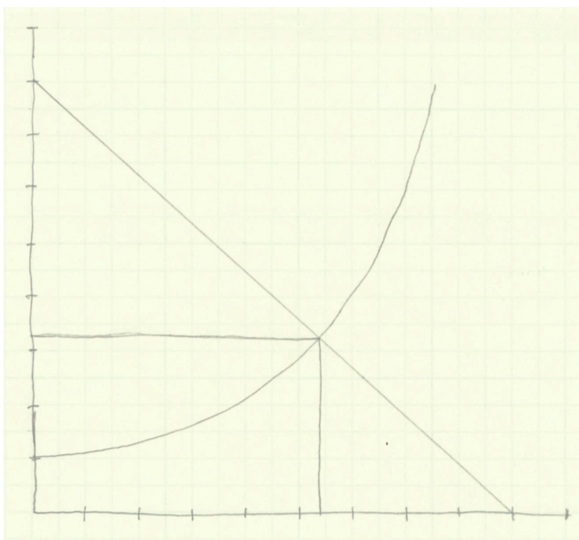


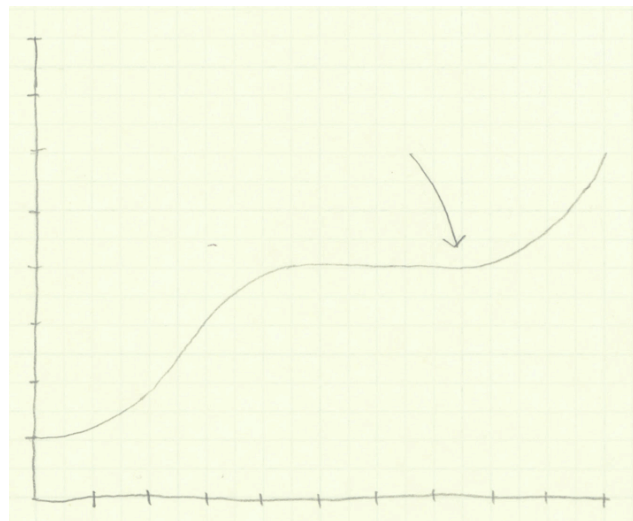
4. Usability Study

To evaluate the accuracy of the segmentation algorithm and better understand the usability of the three interfaces, I carried out a usability study with five subjects. Each subject used all three of the interfaces to draw the same line graphs, and I recorded some quantitative observational measures as well as participants' responses to questionnaires for each interface.

I started each usability session by very briefly describing what Graph Sketcher is for and what the purpose of the study is: "to help me better understand the usability of an experimental interface for sketching line graphs." I also asked participants to speak their thoughts aloud so that I could better follow what they were trying to do. I did not provide any instruction on how to use Graph Sketcher or any of the interfaces, nor any information about how the interfaces differed.

I then provided participants with a pair of hand-drawn graphs (below) and asked them to recreate the lines using Graph Sketcher, "keeping in mind that this is just a sketch: you do not need to do any calculations or precision measurements. Overall, speed is more important than perfection." They were instructed to ignore axes, grid lines, and any other formatting.





While the subject worked, I watched closely and recorded how many times lines were drawn, deleted, adjusted in position, and adjusted in curvature. I captured these counts separately by line type (straight, arc, and complex) to form the following grid:

	draw attempts	curvature adjusts	endpoint ad-justs	deletions
straight				
arc				
complex				

After the subject finished a pair of graphs, I had them fill out a short questionnaire evaluating the interface version they had just used.

Please rank on a scale of l to 5 how much you agree with the following statements:					
	disagree		neutral		agree
I thought Version ____ was:					
• Intuitive: using it felt natural.	1	2	3	4	5
• Efficient: it allowed me to work quickly.	1	2	3	4	5
• Fun: I enjoyed using it.	1	2	3	4	5
• Better than sketching by hand.	1	2	3	4	5

Finally, this process was repeated with the next version of the interface. (The order in which interfaces were presented was different for each subject.) At the very end, each subject was also asked to rank the three interface versions according to which they liked most.

5. Results and Future Work

5.1 Stroke recognition accuracy

Stroke recognition accuracy is a tricky thing to measure because it is not always clear what the "correct" stroke is. Indeed, in the context of sketching graphs, it is sometimes the case that a whole range of lines is "good enough" for the purposes of the sketch. Because of this user-dependent basis for accuracy, I measure accuracy by recording the number of adjustments and deletions of drawn lines performed by users. Below I report the average number of adjustments

in each category, per final line. In the pair of graphs there were three straight lines, one arc, and one complex line. (I will discard results for the arrows in this analysis because there were too many compounding factors affecting their results.)

Averages per line for the full stroke recognition interface

	draw attempts	curvature adjusts	endpoint adjusts	deletions
straight	1.3	0	0.7	0.6
arc	1.2	0.4	1	0.8
complex	2	3.2	4.8	1.2

Straight lines were recognized extremely accurately, with none of them being interpreted as curves, though occasionally they would be split into two segments, causing a deletion or a redraw. Arcs were also recognized correctly most of the time, though there were also some unwanted segmentations and on average users corrected the curvature or endpoints almost every time. Finally, complex lines almost never turned out right the first time, and usually most of the segments were adjusted later in both curvature and endpoints.

These results are for new users who had never used the interface (or the software) before, so some of the deletions and adjustments can be classified as learning effects. In addition, in all cases subjects used a mouse — an input device which is fairly difficult to accurately draw with. Some users also experimented with a digitizing tablet, but in the end were more comfortable using a mouse for the study. Given these experience and hardware limitations, these results are quite impressive.

There is also significant room for future work. Users first attempts at drawing a complex curve would probably be more acceptable if the software automatically smoothed the curving segments in such a way that the slopes at endpoints always aligned. This was not implemented for the current study because the underlying geometry problem proved to be worthy of its own project. Similarly, when making adjustments to curves, these tangencies could be maintained. This would allow fewer adjustments because the smoothness of the line would be maintained automatically, rather than having to be constantly restored by the user.

Given that the primary adjustment required for the straight and arched lines consisted of deleting extra segments, worthy future work would also look into decreasing the false segmentation rate. This is a difficult matter because the thresholds have to balance curve fitting accuracy with robustness to noisy input. Ultimately, individual user modeling may be required to accurately set these thresholds for each user, depending on the noisiness of their input and their likelihood of actually drawing complex lines.

5.2 Comparative Usability

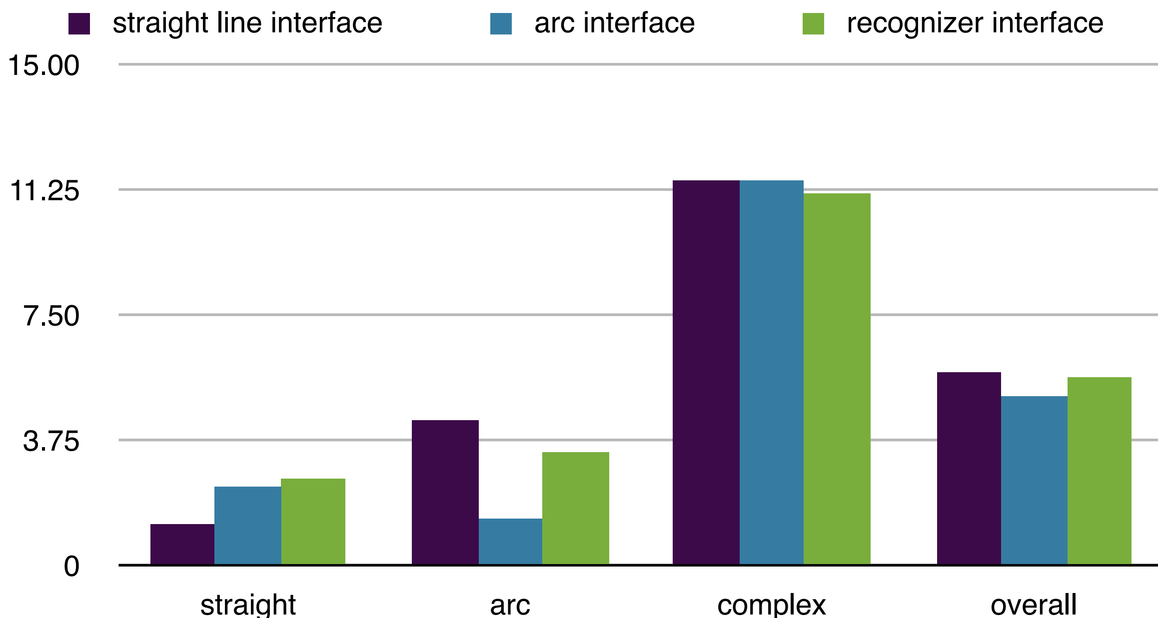
The three interfaces I tested roughly correspond to the three types of lines which users were asked to draw: in the original interface, strokes are constrained as lines; in the intermediate interface, they are confined to single arcs, and in the full stroke recognition interface they can be complex curves. A reasonable hypothesis, then, is that each interface is best matched to drawing its corresponding type of line.

To simplify the analysis, I will collapse the different types of adjustments into a summed measure which can be taken to be the total number of mouse operations required to create an acceptable line. The raw data is as follows:

Average number of operations to create a line of each type

	straight line interface	arc interface	recognizer interface	average
straight	1.3	2.4	2.6	2.1
arc	4.4	1.4	3.4	3.1
complex	11.6	11.6	11.2	11.5
overall	5.8	5.1	5.7	5.5

As predicted, the lowest number of operations per line occurs in the diagonal — when the interface corresponds with the line type. However, this difference is much more significant for straight lines and arcs than it is for complex lines. A chart gives a better sense of this data:



Overall, the arc interface (blue) requires the least number of operations, which suggests that it is the most efficient. This may also suggest that it is the most intuitive interface, because many of the strokes were drawn as part of the learning process, to find out how the interface would behave (since participants had not used the interfaces before and were given no prior information about them).

5.3 Questionnaire results

As described previously, users were asked to rate the intuitiveness, efficiency, and "fun" of each interface. Four of the five participants completed the questionnaire. The average ratings (on a scale of 1 to 5, where 5 is best) are as follows:

	straight line interface	arc interface	recognizer interface
intuitive	4	4	4.25
efficient	4	3.75	4.25
fun	4	3.5	4
average	3.9	3.6	**4.2**

According to these user judgments, the arc interface is the least usable while the full recognizer is the best! This directly contradicts what we found with the observational results. But the story becomes even more interesting when we look at how users **ranked** the interfaces. ("Overall, which was your favorite version? Please rank them.") Assigning each interface a rank of 1, 2, or 3, the average "scores" were:

	straight line interface	arc interface	recognizer interface
score	1.8	**1.3**	3

Here lower is better, so the arc interface is the clear winner. In other words, users rankings of the interfaces completely contradict their ratings of the individual interfaces' usability! Every user ranked the full recognizer dead last, yet most also gave it the highest usability scores.

What is going on here? At one level, this adds weight to the adage, "never trust your users". More deeply, I think that the usability ratings were colored by users beliefs of what they *should* prefer. The full recognition interface is clearly more powerful and impressive than the other two, so people automatically assume that it is more intuitive and efficient — and even more fun! In contrast, the direct rankings seemed to reflect people's intuitive judgments about which interfaces they actually liked using. Most users already knew their opinions and filled out that

section very quickly. They probably did not know **why** they felt the way they did, but their opinions were clear in a way that is impossible for judging abstract qualities like "intuitiveness" given their lack of expertise in the field of human-computer interaction. This could explain why the users' direct rankings of the interfaces better match the empirical results, which showed that the arc interface indeed required the fewest adjustments overall.

These effects are subtle — I would not have noticed the differences in adjustment frequencies without the detailed coding results, and the questionnaire results may not be statistically significant. Doing a larger usability study is the only way to be sure.

6. Discussion

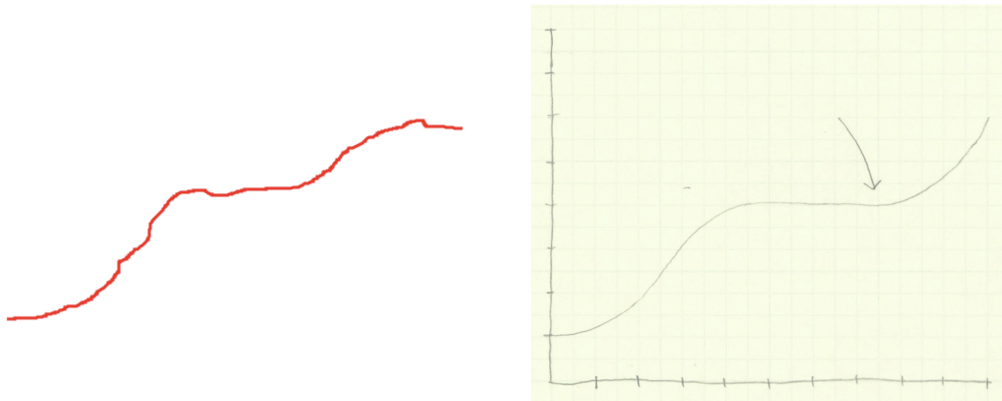
Somewhat ironically, the most usable interface was the one created as an afterthought, and the sophisticated recognizer-driven interface was not significantly better than the low-tech alternatives. This conclusion does have its caveats. As mentioned above, improvements to the stroke recognizer may increase its usability and likability. Additionally, hardware improvements will also likely make a difference: sketching on a tablet computer is considerably easier and more precise than using a mouse, trackpad, or digitizer tablet, so the stroke recognition interface may be accurate enough in a tablet context to substantially boost its usability. Unfortunately, it is currently impossible to run Graph Sketcher on a tablet computer because no Mac tablets are sold (though apparently there is a way to hack/modify a Macbook Pro into tablet form).

Still, given that the vast majority of computer users do not have a tablet PC, the usability study presented here is very relevant for designing and evaluating contemporary software. In this context, I think it is safe to conclude that the study shows that the full stroke recognition interface, as presently implemented, is not a good interface choice. This failure helps to explain why pen interfaces have not yet become widely used. Even for an application domain that blatantly lends itself to a sketch-like interface, and even with a fairly accurate stroke recognizer, the recognition approach was a clear loser. For one thing, users seemed disconcerted by the unpredictable nature of the stroke recognition; they were downright annoyed when the system failed to read their mind. Users also seemed stressed about having to perform accurately in order for the system to correctly recognize their intentions.

Although improvements in software and hardware interfaces could lessen both of these problems, I think the deeper issue here is that of appropriate **constraints**. Pen interfaces tend to be highly unconstrained, which gives them flexibility and power but also makes them overwhelming, stressful, ambiguous, and often inefficient. The most obvious example is with text input: typing is faster, more satisfying, and more accurate than tablet PC handwriting precisely because typing is so much more constrained. Each button does precisely one thing: insert a particular character into the event stream. Even if there existed a futuristic handwriting recognizer that recognized with human accuracy, most people would still rather use a keyboard for the task of inputting characters.

A similar argument can be made for the graph sketching domain. The reason I think the arc interface turned out to be most efficient (and enjoyable) was that it provided the correct degree of constraint for the task at hand. Curves, even complex ones, are really just a series of segment endpoints and curve points (which specify the amount and direction of bulge). The arc interface in effect let users precisely and easily specify these three points to create each arc segment. If they knew what they wanted the first time around, there was no need to go back and adjust anything, and there were no surprises from the recognizer. Creating complex curves only required lifting the mouse button momentarily to indicate upcoming changes in curvature.

The underlying problem with sketch recognition is that there is true ambiguity in every pen stroke, and even the most advanced stroke recognizer will not be able to read minds. For example, should there be a short, straight segment in the middle of either of the drawn lines below?



Depending on the context, it might not matter, or it could be very important one way or the other. The only clear way to resolve this ambiguity is to increase the number of constraints by letting the user point out exactly what they want. One approach is to display “n-best” lists of the potential options the recognizer thinks you might mean. But given the ease of simply specifying one’s intentions the first time around, and the fact that *every* segment is potentially ambiguous, I think there is a strong case to be made that the arc interface will be the best approach for this line-graphing task no matter the improvements in software and hardware.

References

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

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

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