Lineogrammer: Creating Diagrams by Drawing

Robert Zeleznik, Andrew Bragdon, Chu-Chi Liu, Andrew Forsberg
Box 1910, Brown University
Providence, RI 02912 USA
{bcz, abragdon, cliu, asf}@cs.brown.edu

ABSTRACT

We present the design of Lineogrammer, a diagram-drawing system motivated by the immediacy and fluidity of pencil-drawing. We attempted for Lineogrammer to feel like a modeless diagramming "medium" in which stylus input is immediately interpreted as a command, text label or a drawing element, and drawing elements snap to or sculpt from existing elements. An inferred dual representation allows geometric diagram elements, no matter how they were entered, to be manipulated at granularities ranging from vertices to lines to shapes. We also integrate lightweight tools, based on rulers and construction lines, for controlling higher-level diagram attributes, such as symmetry and alignment. We include preliminary usability observations to help identify areas of strength and weakness with this approach.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces.

General terms: Design, Human Factors, Algorithms

Keywords: Drawing, alignment, diagram, pressure, snapping, recognition, gesture, sketching, beautification, ruler, pen, pen-centric, symmetry, handwriting, disambiguation.

INTRODUCTION

Diagrams drawn with pencil on paper, spanning a vast domain of styles, can be created in seconds using only a lightweight erase-and-redraw editing model; however, they are also characteristically imprecise. Alternatively, diagrams created on computers can leverage sophisticated, precise editing techniques, but typically require a more deliberate, strategy-oriented "choose-a-primitive" approach which can be effortful, limiting and distracting. Our hypothesis, however, is that the familiar, fluid and lightweight line-based interaction and editing style of pencil and paper can be seamlessly combined with higher-level shape-based interactions to provide a best of breed system.

Perhaps in the idealized form of such a system, users would freely sketch their diagram, as if using pencil and paper, and, as desired, see a cleanly formatted result. However, since freehand drawings can be ambiguous even to their creators and no technology offers perfect recognition results, such an idealized *deferred beautification* system may be unattainable. This raises a research question: under what conditions is imperfect deferred beautification advantageous over beautifying immediately? As an initial step toward an answer, we are pushing the boundary of immediate beautification to create a baseline for measuring future progress in deferred beautification. Our expectation is that immediate beautification may be less desirable during conceptualization phases, but may be potentially superior to deferred beautification when entering or refining a well-understood diagram due to a greater sense of control.

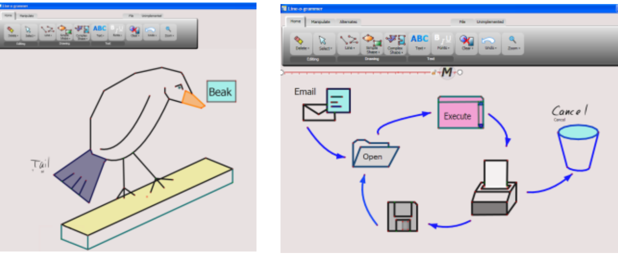

Figure 1. Screenshots. Simple drawings composed of lines, curves, text and polygons. Drawings were made without an explicit mode switch. The GestureBar discloses gestures and drawing strategies.

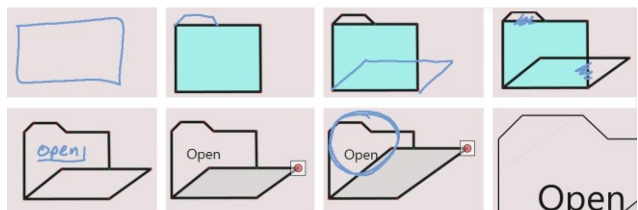

Figure 2. Drawing sequence. Top: Drawn lines are beautified and snapped. Unwanted segments are scribbled out. Bottom: text is recognized and typeset, a vertex is moved, and a zoom gesture is made.

Lineogrammer (Figure 1) reflects five design criteria that we feel address limitations of prior approaches:

- Disclose gestures and their nuances
- Avoid mode switches for text, geometry, gestures
- Simplify snapping with lightweight editing
- Enable interaction at different perceptual scales
- Make lightweight, special-purpose tools available

Permission to make digital or hard copies of all or part 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. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'08, October 19–22, 2008, Monterey, California, USA.
Copyright 2008 ACM 978-1-59593-975-3/08/10...$5.00.

By blending aspects of various known and novel interaction styles (Figure 2), we believe Lineogrammer is a novel, best-of-breed approach to creating diagrams on a computer. Specifically, Lineogrammer highlights contributions in four synergistic areas, including:

- Heuristics for modelessly disambiguating between text, geometric drawing elements and command gestures
- A snapping engine that can adjust the input stroke and existing drawing elements
- UI mechanisms for manipulating the diagram at granularities ranging from vertices and individual line segments through inferred primitives
- Lightweight tools, including construction lines and a ruler, which support alignment and symmetry

PRIOR WORK

There are numerous commercial systems for creating precise, domain-specific diagrams, such as Microsoft Visio. However, these systems are tuned to create complex but rigid structures and not the general diagrams facilitated by Lineogrammer's literal drawing metaphor. There are also commercial systems for creating general purpose diagrams, such as Microsoft PowerPoint. With a few exceptions, like Corel Grafigo 2, these systems require users to interact modally to find and instantiate primitives, and do not allow low-level shape editing of primitives at the level of lines. Although quite powerful, these systems do not support the familiar strategies and techniques of pencil and paper drawing. There are also several systems that provide sketch-based interactive beautification only for domain-specific diagrams, such as a modeless system for sketching directed graphs [1].

In contrast, relatively few general-purpose diagramming systems have been developed with sketch based interfaces. Saund's image editor [20] exploited perceptual structure to simplify selection and manipulation of visual structures in bitmap images. We apply similar notions to facilitate the manipulation of structured vector drawings at different perceptual scales ranging from the vertex, to line segment, to polygon. Igarashi [8] presented the most similar system to ours, Pegasus which introduced the term *interactive beautification*, and reviewed the relative merits of interaction beautification to other approaches [5] [11] [16]. In essence, interactive beautification amortizes the complex errors and error recovery associated with batch processing a sketch through an incremental, monotonic snapping engine that considers local and global context, but is much simpler than constraint-based approaches.

Lineogrammer both extends Pegasus and adopts a different approach to snapping which reduces display clutter. Lineogrammer's extensions include: recognizing text, curves, and single-stroke polygons, a multi-function ruler widget, and a range of gestural and direct manipulation techniques for multi-scale editing and view control. However, Lineogrammer approaches snapping differently than Pegasus. When Pegasus displayed more than one snapping alternate, we observed that users often were distracted or confused and did not choose a correct alternate when it was displayed or spent considerable time searching for the correct alternate whether or not it was actually displayed. Thus, with Lineogrammer, we explore the notion that the overall cost of a complex alternates display is greater than the cost of more demand-driven correction techniques, such as joining lines with a connecting stroke, dragging vertices, or just erasing and redrawing. Like Pegasus, Lineogrammer makes the best-fit snap line be the default interpretation, but unlike Pegasus, Lineogrammer will show at most one alternate. In addition, Lineogrammer's snapping techniques are speed dependent which enables users to effectively override snapping by drawing slowly and carefully.

PaleoSketch [15] provides interactive beautification by best-fitting an input stroke with one or more of eight classes of higher-level geometric primitives. Lineogrammer complements a simplified version of this technique (that only supports complex fits for polylines) with incremental techniques that use the existing drawing as context for snapping simple primitives entered one at a time.

The alternative of deferred beautification has shown promise for 3D modeling particularly using oversketching techniques [9]. For 2D drawing, Yu [22] combined interactive (similar to [15]) and deferred beautification, but most work has focused either on imprecise or domain-specific structured drawings[4][12]. Plimmer, for example, used deferred beautification with interactive feedback of the recognition state when sketching the domain-specific, highly structured geometries needed for UML diagrams and GUI forms [17].

More recently, Ohki presented a 2D drawing system [13] that supports a related notion of constructing drawings based on set operations over primitive shapes; however it is not modeless in the sense that shapes are not drawn naturally with a pen and interaction requires an explicit menu selection to change tools. Ohki's system also demonstrated some higher-level suggestions, for instance to create symmetric objects and to perform iterative placement, that would be interesting to explore within Lineogrammer.

There are several techniques for distinguishing between handwritten text and drawings [14][21], including the Microsoft Windows InkDivider. Based on Patel's studies [14], we hand-optimized a variant of her binary classification scheme. Our technique supports the interactive beautification constraint that stroke classifications are fixed after 1/2sec by combining our interactive symbol recognizer [23] with rules based on other spatial and temporal features. PostBrainstorm [6] is related in that new strokes are associated with existing objects and typeset recognition is displayed below all "text-like" strokes.

Lineogrammer's rulers are closely related to Alignment Sticks [18], but differ because Lineogrammer's interactions are designed for uni-manual interaction and provide additional functionality, including symmetry operations.

SYSTEM DESIGN

The virtue of pencil drawing derives from its unique combination of expressiveness and lack of modes–in short, it is transparent. In designing Lineogrammer, we attempted to

replicate aspects of this style by replacing the cognitively heavyweight UI management tasks, such as foraging through toolbars, with functionality centered on drawing elements. Thus, Lineogrammer has no conventional toolbars; instead it is a blank drawing surface that, for discoverability of strategies and functionality, is docked to an explanatory "toolbar". As with pencil and paper, users draw "commands, " text and geometry interchangeably, but with assistance provided by a snapping and shape inference engine and formatting tools, to create refined results.

As with any complex system, numerous interdependent design choices were made at varying levels of detail. For simplicity, however, we present the design in terms of the following major categories:

- Making techniques and gestures learnable
- Classifying input strokes
- Snapping lines
- Editing with gestures and widgets

ITERATIVE USABILITY TESTS

Over the course of the development of Lineogrammer, we conducted three rounds of usability testing. We include preliminary observations gleaned from observing 10 pilot users (6 female/4 male,aged 18-30, right-handed), recruited from the general population of Brown University, including four novices. Eight subjects had no Tablet PC experience. Tests were conducted on a Toshiba Portégé M200 Tablet PC with 1 GB of RAM and a 60 GB hard drive. In the first pilot round of testing, the software was still very early and two subjects participated, with the goal of getting rough feedback on the overall approach. Five subjects were then brought in for the second round; this version did not yet include stroke alternates, zoom, the floating toolbar, or selection repetition. Finally, three subjects took part in the third round with a much-improved version of the software reflecting many refinements based on previous testing.

Subjects were given a series of tasks of two types: verbal and visual. Verbal tasks consisted of a verbal description, such as "draw your family tree using boxes, lines and text" or "draw a cartoon house with a door, window, and a chimney." Visual tasks consisted of printed diagrams, which users were asked to replicate. No training or hints were given, however for some tasks we asked users to make use of a specific feature, referencing it by name.

In the sections that follow, we include an observations subsection detailing anecdotal results from these tests.

MAKING TECHNIQUES AND GESTURES LEARNABLE:

We have explored a variety of techniques for training novice users in pen-centric and gestural systems, including providing heads-up displays, online reference manuals, tutorials, and menu shortcuts. However, our observation is that users do not seem receptive either to a priori learning, or having information pushed at them. Instead, they seem to want to explore information narrowly, on-the-fly that matches their immediate task or mental goal.

Thus, we designed a GestureBar[2], inspired by [10][24], to represent all system operations, including some critical multi-step techniques, as toolbar items (Figure 3). Unlike conventional toolbars, GestureBar items do not perform functions but rather indicate *how* to perform them. Toolbar items display one or more annotated animations and provide a 'Practice' area for exploring gestures or techniques with the assistance of nuanced, targeted feedback.

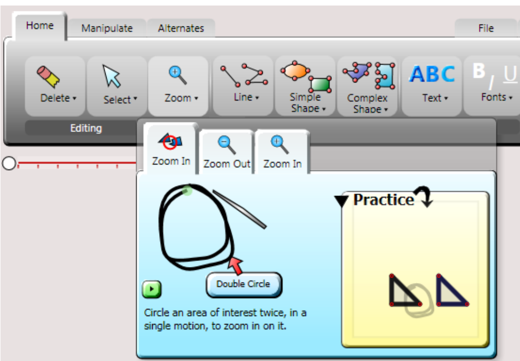

Figure 3. Gesture Bar. The user chooses Zoom to see related gestures along with a Practice area.

Although GestureBar is still being formally evaluated, our pilot tests indicate that it is more effective at disclosing gestural interactions than conventional crib sheets, at least for novice users. Users with no experience whatsoever with gestural UIs have been able to find, learn and perform all of Lineogrammer's gestures without human assistance over the course of half hour to hour long sessions.

CLASSIFYING INPUT STROKES:

There is no a priori distinction between drawn strokes that represent geometry (line, polyline, and curve) and those that represent text (print or cursive) or gestural commands. Additionally, the same stroke instance can logically fall into more than one classification (*i.e.*, a circular stroke can be text or circle geometry). Thus we developed a set of empirically-derived recognition heuristics for classifying a stroke when it is entered. These heuristics are encoded in an ad hoc rule base of procedural methods, although other implementations are possible and might be superior.

Gesture Design and Classification

To assist determining whether an input stroke indicates a gesture, we designed our gestures as a balance between visual/motor appropriateness and machine recognizability. This design strategy by definition makes it straightforward to distinguish gestures from other input strokes since we avoid gestures which are hard to disambiguate using only a stroke segmentation algorithm similar to Calhoun's [3] and a set of simple geometric rules. Since gestures are visually transient, we attempt to increase awareness of their invocation by fading an icon away at the location where they were recognized that matches the toolbar icon for their function.

The delete gesture is discussed in detail the Editing section. The lasso gesture is related to the pigtail lassos used in Scriboli[7], but is recognized procedurally without needing a modifier key by three successive features: a nearly closed loop, a high-curvature cusp, and a straight tail. This definition recognizes both pigtails and non self-intersecting tails, and facilitates control over the exact boundary of the lasso.

Table 1 summarizes design issues of the full gesture set.

	Delete – defined broadly as stroke with >= 3 sharp cusps. Broad definition is necessary because of its frequent use. Details above.
	Join lines – is essentially not a gesture, but what might be done on paper to join lines.
	Lasso select – defined broadly as a closed loop with a sharp, straight hook-like retrace at the end.
	Typeset text (top) – underline was our first choice, but confusable for math. Edit properties (bot.) – almost any gesture would do, but ‘e’-like loop was mnemonic.
	Undo (top); Redo (bot.) Symmetric gestures to be memorable. Straight lines were desired but were ambiguous w/ text & geometry.
	Zoom in (top) – flattened diagonal zigzag from bottom-left to top-right that transitions well to interactive zoom rectangle. Zoom in (bot.) – Some users found double-circle to be more natural and more reliable.
	Zoom out – to be memorable, this is the Zoom In gesture reversed.
Press & hold	Pan – not considered ideal because of pause, but commonly used in PDAs
Line, curve, ellipse, polygon	Literal gestures for creating geometry are really shape recognition, not gestures. These need to be disambiguated from text.

Table 1. List of Gestures. Red dots indicate the start of the gesture when important. Green indicate existing diagram context.

Observations. We found users were for the most part aware of all the systems gestures and were in fact eager to learn them. In initial testing, some users performed the Typeset Text gesture without a sufficiently sharp hook at the end or without a hook at all. We addressed this by adding text tool tips in the GestureBar which point out important details of the gesture. In later testing, we noticed no systemic barriers with gestures, although the zigzag zoom gestures were harder for users to perform consistently correctly.

Text/Geometry Classification

Input strokes that are not gestures must either be text or geometry. Although sophisticated techniques are the subject of ongoing research [21][14], they are not the focus of our work so we use a simple but workable set of heuristics which consider the size, geometry, and spatio-temporal relationship of the input stroke to pre-classified diagram elements. An initial criterion that trivially handles many cases is to restrict the maximum size of text input to 2cm, which seems reasonable since larger text is unusual but can still be easily achieved by choosing a large font or interactively scaling. In addition, simple, single-stroke closed polygons and many character symbols, can be easily distinguished using a recognizer trained to distinguish the salient differences of polygons and text symbols. However, circles, straight lines and polylines present specific disambiguation challenges, for instance, from symbols like ‘0’, ‘1’, and ‘w’ respectively – discussed in more detail below.

Existing Context. Existing diagram context provides a foundation for disambiguating text and geometry (Figure 4). For example, if a straight vertical line stroke is drawn next to a known text elements and of a similar size, we will classify it as a text symbol (‘1’, or ‘l’). Similarly, if that same stroke were drawn starting on, ending on, or intersecting an existing line stroke, then we will classify it as a line. Neither of these heuristics is guaranteed to work, but empirical observations indicate that leveraging local context, when it exists, significantly reduces mis-classifications.

Figure 4. Text/drawing disambiguation. A leading ‘0’ or the initial stroke of a ‘T’ requires a 500ms delay during which an additional text stroke must be input.

Isolated strokes. The real disambiguation challenge occurs when a stroke is drawn in isolation, such as the start of a word or the beginning of a multi-stroke shape. In these situations, we try two metrics: the ratio of the stroke length to its number of cusps, and a yes/no classification provided by our single character recognizer. Cursive words will be classified as text based on the first metric. The second metric matches input strokes to a broad, writer-independent set of character symbol templates and classifies the stroke as text if it matches any template. We intentionally omit templates for ‘0’, ‘1’, and ‘i’ because they would also match common strokes intended as geometry. Instead, for fundamentally ambiguous strokes in isolation, we rely on a temporal recognition strategy where slow input is biased toward drawing elements and fast input toward text. For example, a straight line (or a circular stroke) would be placed on a deferred recognition queue. After 500 ms, the queued stroke would be classified as a drawing element (line or circle) unless a new stroke is drawn nearby that the recognizer labels as text with high confidence. In this case, the queued stroke would be recognized as a ‘1’. This implies that some characters, like ‘0’, cannot be recognized in isolation. When text is misclassified as geometry, the Typeset gesture can be used to coerce the geometry back to text.

We support curves, circles and ellipses, although our implementation of them is primitive and would benefit from Paulson’s approach [15]. The curved geometries we do support are treated inefficiently as polylines with unselectable vertices. In addition, our heuristics for distinguishing complex curves from text requires more work.

Observations. We initially allowed text to be drawn at any size which meant that all isolated lines, no matter how large, would be subject to the 500ms recognition delay and

misrecognition as text. Pilot users, however, never drew large text, so we restricted text classification to strokes <2cm high. This significantly reduced false positive text recognition, although false positive geometry recognition can be a problem in dense input that necessitates zooming.

Since we deliberately support the strategy that slow input is biased away from text recognition, we observed that some novice users had initial difficulties because they block printed text in a slow, overly deliberate manner, as compared to their writing speed when we asked them to write on paper. We expect that this problem can be mitigated by adding explicit detail in the GestureBar.

SNAPPING LINES

Although natural and expressive, hand-drawn input is often imprecise, particularly as writing speed increases. Consequently, Lineogrammer attempts to snap input geometry to existing diagram features (vertices, edges, midpoints, and polygon centers) using a tolerance which is dynamically adjusted based on drawing speed. However, since snapping occurs while the diagram is being constructed when there is limited context, we cannot expect a unique, monotonic (*i.e.*, affecting only the input stroke) snap result to be sufficient. Suggestive techniques attempt to ameliorate the uniqueness problem by presenting multiple potential snap results *in situ* for the user to select from [8]. Since the range of alternates for any given line can be combinatorial quite large, culling must be done to achieve a manageable set. Even when Pegasus presents only three alternates, the display can look cluttered and be confusing or distracting; adding more alternates increases the odds of the intended result being available but is visually even less acceptable.

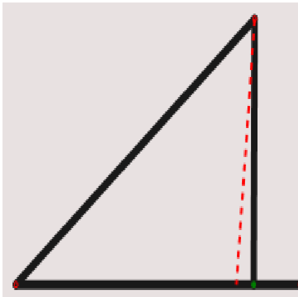
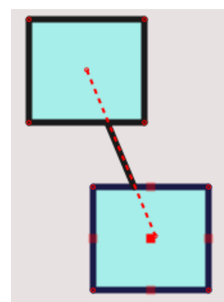

Figure 5. Stroke alternates. (Left) The dashed line alternate closely matches the input stroke; the solid vertically-snapped line is the default. (Right) The alternate is the line between the two centers; the default is the same line showing only the segment *between* the rectangles. Tapping a dashed line switches the line to that alternate.

We instead treat snapping optimistically as a lightweight aid to clarified drawing, not as a sole solution, and provide at most a single alternate, represented by a dashed line (Figure 5). Our experience is that simple editing options, such as erasing and carefully redrawing or interactively manipulating, are preferable to the distraction of complex alternate displays which may not contain the correct result.

The snapping algorithm can snap both an input line's orientation to the principal axes or to other lines and their mirror images, and perpendiculars, and also its endpoints to existing vertices or lines (midpoints and shape center targets are highlighted upon approach.) The algorithm is non-monotonic and thus, after snapping the input line, the algorithm may modify existing lines; however, the algorithm will only extend, but not re-orient, existing lines to snap to the input line. When a single input stroke is recognized as a polyline or polygon, additional considerations apply to ensure the integrity of the resulting shape.

We consider this approach optimistic because we produce an alternate only for the nearest orientation snap, but not for different endpoint snaps. Our expectation is that when snapping targets are clustered, users will move more slowly causing the snapping threshold to be reduced; when snap targets are isolated, they will be able to move faster, increasing the snapping thresholds, and still benefit from snapping. The dynamic snapping radius for vertices is:

```
speed  = arclength of last 10 samples (~ 1/10th sec)
radius = min(stroke.Length/4, max(.1in, min(.3in,speed)))
```

Controlling line orientation, however, uses an alternate, since orientation targets are often not visible or are not localized to the stylus path which makes unintentional snapping more likely. Thus, when a line orientation is snapped, we show an alternate, defined by the line's original start and end point, which overrides the snap. In either case, the line's endpoints will be snapped if within the threshold of a feature; however, when the line orientation has been snapped, the endpoints can only snap to features that are collinear with it.

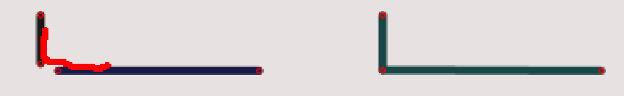

Figure 6. Joining lines. Mis-snapped lines can be joined by overdrawing (shown in red for emphasis).

Observations. We initially used a static snapping threshold, but found that it, like grids, too often caused undesired snapping. Turning down this threshold tended to avoid unwanted snapping but forced users to draw everything with deliberation. With the dynamic threshold, users qualitatively seem to have control snapping better, although more sophisticated techniques are no doubt possible. When intended snaps are missed, overdrawing joining lines is natural and usually fixes the problem (Figure 6.) In other cases, users instinctively seem to adopt the habit of scribbling out mis- or un-snapped lines and redrawing them more slowly to gain the advantage of the tighter snapping threshold. Visually indicating this threshold by highlighting the nearest target might also improve the experience but is complicated since snapping tolerances are a function of the entire stroke. A caution is that several users found feedback of even a single alternate was distracting.

EDITING WITH GESTURES AND WIDGETS

Lineogrammer supports a suite of additional interactive functionality which complements the basic drawing/snapping input model. Although some users are sufficiently skilled at drawing that they may be able to accurate-

ly enter a final design, more typical users would need to manipulate their designs to gain better control and perspective over the design space. The techniques we describe in this section work collectively to facilitate lightweight exploration and flexible workflows and include:

- Scribble "drawing"
- Lightweight selections
- Moving, rotating, scaling, Pressure Snapping
- Symmetry, alignment and distribution
- Working at different levels of detail
- Formatting text

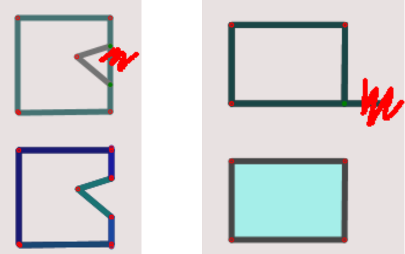

Figure 7. Deleting. (Left) A notch is "sculpted" from a rectangle. (Right) Deleting the intersected line segment causes a rectangle to be inferred.

Scribbling as a Drawing Technique

Besides drawing lines, perhaps the most important interaction technique in Lineogrammer is the scribble delete gesture. The idea is not just that scribbling is an easy almost reflexive way to correct imprecise drawings, but more importantly that it affords a powerful subtractive sculpting strategy for creating shapes by successive refinement (Figure 7). Unlike conventional shape palette approaches, Scribble Drawing allows users to draw a simple approximate shape, intersect it with refinement lines, and then remove line segments that are inferred from the intersection points. For this approach to be effective the transition between drawing and deleting must be fluid and so we rely on a very broadly defined scribble gesture that allows users to perform it in arbitrary contexts in ways that feel natural to them. This approach is similar to [8], except we delete all inferred line segments intersected by the scribble, not just the first. Additionally, we need to disambiguate scribbles, defined broadly as >=3 somewhat sharp cusps from potentially similar cursive text. Instead of relying on ambiguous geometrical measures, we use temporal and spatial context. If a scribble occurs within 500ms of a previously classified, neighboring text stroke and does not intersect it more than 5 times, we treat it as additional text. Scribbles that do not overlap existing elements will always be text.

Observations. The scribble gesture is typically the first thing mentioned when users are asked to name something they like in the system; they frequently use it reflexively.

Lightweight Selections

To interact with a diagram, the user must communicate a selection scope. To maximize expressivity, we wanted it to be easy to select arbitrary contiguous and disjoint collections of vertices, lines and polygons, but at the same time we wanted selection to be cognitively lightweight. Previous work demonstrated a powerful perceptually-based approach to trace out selection paths or poses [20]. Although we like this approach, it presents integration challenges for our modeless interaction environment, and we felt that for many, if not most, common cases, a simpler tap-based multi-selection mechanism [7] would be preferable. Still, we augment tap selection with a Lasso gesture can for selecting larger sets of neighboring elements. In either case, selections are additive such that tapping on unselected drawing elements adds them to the current selection; tapping on a selected element de-selects it, and tapping on the background de-selects everything. Tap selection of vertices is straightforward, so we will only discuss tap selection of higher-level diagram features.

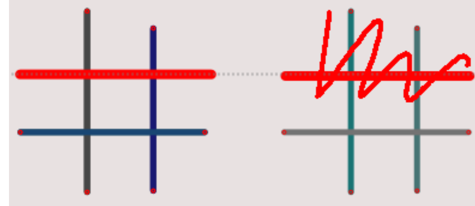

Figure 8. Selecting lines. (Left) Tapping selects whole lines ignoring intersections. (Right) Scribbling then deletes only the selection.

Line Selection. Tapping on a line can be ambiguous when the line intersects with other lines. For selection, we ignore the inferred line segment boundaries and select the entire line when any part of it is tapped (Figure 8). To restrict selection to just an inferred line segment, the line must be explicitly split by tapping on the point of intersection to create a vertex. The reason we chose this behavior is that collinear artifacts generally do not occur unless intended and thus we want to preserve collinearity across selection/manipulation actions. We note that the opposite assumption is made for deletion which is typically used to clean up errant line segments and to perform subtractive drawing. The difference between selection and deletion can be exploited: if the user wants to delete an entire line, instead of trying to scribble over each inferred line segment, they can instead tap anywhere on the line to select the whole line, and then scribble over any part of the selection to delete the entire selection.

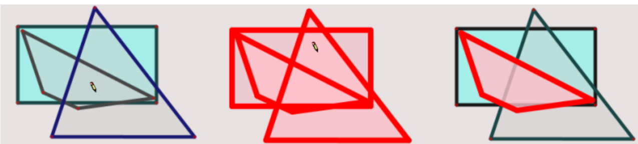

Figure 9. Polygon selection. There is no front-to-back ordering for inferred polygons. Clicking the overlapped trapezoid will select one or more other polygons which can be deselected by tapping them.

Polygon Selection. Although, a polyline or polygon can be selected by sequentially tapping on all of its edges, tapping inside a polygon is more convenient. Since users don't explicitly create polygons – they are inferred automatically by detecting closed loops of edges – issues of ambiguity again arise. The dominant heuristic applied is to consider

any closed loop of lines that does not traverse any line intersections to be a polygon. However, ambiguities may still exist because inferred polygons may overlap one another (Figure 9). If a tap falls within more than one overlapping polygon, then they all are selected; tapping on polygons that were not intended de-selects them. The exception is that we do not select any polygon that completely contains another polygon that was also selected. Alternatively, selecting a containing polygon automatically selects all contained diagram elements, which we believe is generally desirable for manipulation. Nonetheless, selection of certain types of tiled shapes, among others, can require multiple taps or lassoing. Going forward, particularly as we extend support for filling inferred polygons, we believe that additional design consideration is warranted.

Figure10. Floating toolbar. A marking menu (green triangle) that was expanded to show toolbar items.

Associated with every selection, is a local floating toolbar (Figure 10) for accessing less critical functionality, including: setting colors and thicknesses, grouping the selected elements, making all selected objects be of the same size, and a Copy handle for copying the selection. A drag handle supports repeated copying (Figure 11).

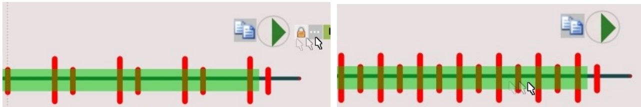

Figure 11 Repeated copying: Dragging the '…' tool item creates default-spaced copies (left.) Releasing and dragging in the green area changes the number of copies (right.) Tapping ends the action.

Observations. We had observed that users often created a selection, moved it and then started drawing or attempted to select something else. In the former case, they did not usually notice that the selection remained as they started to draw which would later generate confusion. In the latter case, they would inadvertently find themselves extending the original selection instead of replacing it. In response to this, we now automatically deselect everything when a new stroke is drawn or after a selection is moved and another primitive is selected. We also noticed that several users who were explicitly informed that they could press the stylus button to select objects with a conventional lasso (*i.e.*, no gestural 'hook' required) preferred instead to learn the lasso gesture. A more thorough follow-up investigation of the preference for gestures over stylus buttons is warranted.

As for the floating toolbar, we noticed that it sometimes got in the way, especially after we extended its functionality with more buttons. We tried to address this scalability issue by replacing the toolbar with a marking menu. However, we then noted that the nature of the marking menu made people less aware of its functionality. We now collapse the toolbar, but bias interaction toward expanding it since it looks and acts like an expandable ribbon. That is, the East marking menu item allows users to seemingly drag the toolbar out of the menu (Figure 10.) More advanced users can still benefit from the arguably more fluid and efficient traditional way of invoking a marking menu item.

Moving, Rotating, Scaling and Pressure Snapping

We facilitate interactive adjustments to a diagram by allowing selections to be moved by dragging them; the exception is selected polygons which can be moved by dragging on their interior and scaled along a principle axis if one of their edges is dragged. To apply a scale or rotate transformation to one or more lines, vertices, or polygons, a "nail" needs to be created before dragging on the selection (Figure 12). This nail, created with a downward Flick (equivalent to the Flicks used by the Microsoft Windows Vista OS), serves as the center of rotation or uniform scaling. A circular and linear constraint guide is created during this interaction to facilitate only rotating or only scaling the object.

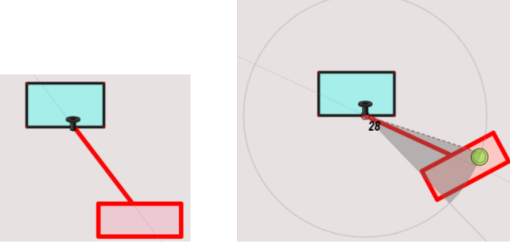

Figure 12. Nails. A nail is input with a flick. Snapping to the circular guide forces rotation about the nail; the linear guide constrains scaling; otherwise both are done. Light pressure disables snapping.

We also support a novel Pressure Snapping technique that allows users to explicitly enable or disable snapping while interacting. The notion is that with light pressure, objects "slide over" snapping detents, but stick with firm pressure. Pressure Snapping transitions are detected and disclosed using a pressure meter widget based on [19] but augmented with a more explicit ToolTip message.

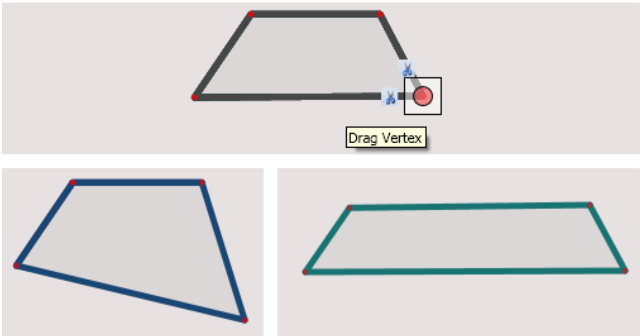

Figure 13. Vertex dragging. Dragging the inner red circle (above) distorts the shape (left). Dragging the white outer square (top) Rubber Stretches it (right).

A selected vertex has a compound widget with a red center circle and white outer square which provides direct access to two different dragging behaviors (Figure 13). Dragging

the inner circle rubber-bands the edges incident on the vertex. Dragging the square does *Rubber Stretching*, a form of local scaling, in which edges incident on the vertex move without changing their orientation and stretch to maintain intersections with other lines. Rubber Stretching is equivalent to scaling when the vertex of a rectangle is selected.

We also attempt to preserve perceptually salient characteristics of line intersections when lines are moved because we expect that these characteristics are intended and should only be broken explicitly by the user. For instance, if a line that ends in a 'T' junction with an unselected line is moved, we preserve the topology of the drawing by sliding the line along the unselected line (Figure 14); however, the 'T' junction is broken if the line is dragged off the end of the unselected line. In non-'T'-junction cases, we stretch unselected lines to maintain the drawing topology. In either case, we are attempting to localize the editing effects.

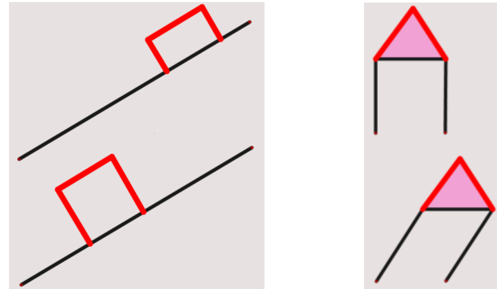

Figure 14. Moving lines (shown in red). (Left) When lines terminating in 'T' junctions move, they stretch and slide. (Right) Otherwise the unselected incident lines stretch to stay attached.

Observations. We initially experimented with placing a rotation handle on selections, but were frustrated that simple rotations around object corners were not possible. We considered extending the widget with a movable center, but felt that the Flick gesture offered the same functionality more efficiently. In practice, we have found some users are adept at using nails, whereas others have difficulty with its unique timing requirements. Further investigation of widget vs. gestures for rotation and scaling is warranted.

Pressure Snapping addresses the frustration of unintended snapping, although the concept of using pressure to control functionality was not expected by our users. Initial users, exposed only to the pressure meter, were unaware that they could control snapping to guides. Subsequent users, exposed to the ToolTip, were able to pick up the technique.

Disclosing rubber-banding and rubber-scaling functionality needs improvement since users are generally unaware either that both functions exist or of what they both do. Currently there is no GestureBar entry for the vertex widget. In contrast, we found that dragging lines ending with and without 'T' junctions was well understood; however, complex selections containing multiple 'T' junctions along different axes behave less predictably.

Symmetry, Alignment and Distribution

Most of the techniques described so far address local problems of precision; we also use a notion of *rulers* to facilitate control over more global spatial relationships. Our rulers are similar to the *sticks* [18] with straightforward adaptations, using rubber-band resize handles, to be completely controllable with only the stylus tip-switch. Dragging the ruler's body translates it; and dragging the white handles (Figure 15) resize or rotate the ruler about a sliding pivot. Our rulers also use a marking menu to switch between behaviors, including shape and vertex alignment, and symmetric mirroring about or even distribution along the ruler's axis. A toolbar item on the floating toolbar, appearing when a line is selected, can be used to automatically snap and orient the ruler to the selected line.

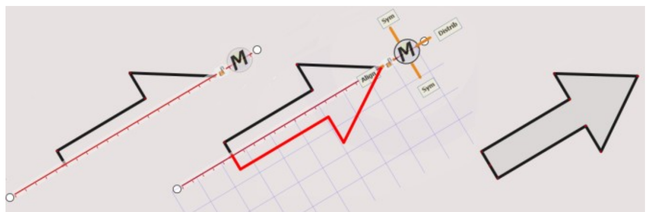

Figure 15. Symmetry ruler. Ruler oriented along symmetry axis (left.) Hovering over bottom menu item previews symmetry (middle.) Result (right.)

Rulers can be used to make symmetric shapes in two ways. After drawing half of a symmetric shape, the ruler can be placed along the axis of symmetry, and a marking menu aligned with the ruler can be used to reflect the shape across the axis of symmetry (Figure 15). It is important that the marking menu be aligned with the ruler, so that the two symmetry functions are located along an axis perpendicular to the ruler; thus choosing the appropriate menu option feels like "turning a page." In typical cases when only some drawing elements are part of the intended symmetric shape, the scope of the symmetry operation must be restricted by first selecting the edges to include in the symmetry action. The ruler can also be locked into a symmetry mode in which all edges drawn on one side of the ruler are immediately mirrored to the other, and dragging a vertex or edge mirrors the interaction on the other side.

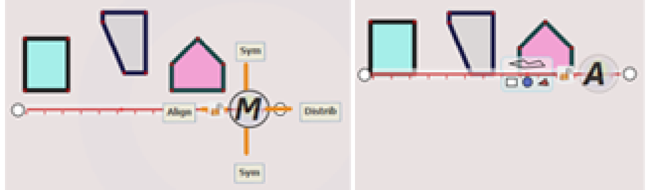

Figure 16. Alignment ruler. Hovering over the 'M' icon displays a marking menu (left). Choosing the Align option (West marking menu item) displays two handles on the ruler (right); dragging the ruler from these handles pushes shapes or vertices (right). The white circles rotate the ruler around a movable pivot.

By selecting 'Align' on its marking menu, the ruler adds shape and vertex alignment handles (Figure 16). Dragging the handles makes the ruler "grab" objects it passes over, while grabbing the body allows the ruler to be repositioned. Selecting vertices, edges or shapes prior to moving the ruler explicitly restricts the scope of alignment to those features.

In addition to alignment, the ruler also has an even spatial distribution function on its marking menu. Again, the placement of this menu item is significant, since it allows users to stroke from the menu along the axis of the ruler as if sweeping out a deck of cards. Any shapes, lines or vertices previously selected would be evenly distributed along the axis of the ruler.

Figure 17. Construction lines. Two construction lines are created from the left rectangle by tapping to select its bottom and top edges. Subsequent drawing lines may snap to the construction lines.

Observations. During demos, people overwhelmingly seem to thinks rulers are "very cool." However, as designers we are less sure of their utility. They often appear to be more heavy-weight than what seems needed to perform a task, particularly in the case of alignment. Thus, we added an additional lighter-weight construction line mechanism (Figure 17) for aligning shapes when they are being drawn. When any line is selected, an infinite construction line is created through it. Subsequent drawing operations will snap to construction lines as if they were diagram lines.

In an early implementation, we restricted alignment with the ruler to only move shapes in one direction, the direction it was initially pushed. However, it was not uncommon for users to overshoot their target requiring additional interaction to push the elements back in the other direction. We changed the behavior to be grabby so that shapes can now be pushed or pulled perpendicular to the ruler axis.

Working at Different Levels of Detail

It is frequently necessary to view a diagram at different scales, particularly to add local details; we support this with gestural zooming. Initially, we recognized one gesture, a diagonal line up to the right (for right-handers, and up to the left for left-handers) with a zigzag in the middle. This gesture is recognized as it is being drawn and switches to a rubber-banded rectangle indicating the zoom region. Based on feedback from some users, we added a second gesture, a double circle around the area to be zoomed. In either case, zooming in saves the original zoom location on a stack. To zoom out to the previous zoom level, the initial zoom gesture is drawn in reverse.

Panning is also useful, but appears to be needed less frequently, so we assigned it a simple, but less fluid, press-and-hold gesture. The timeout for this gesture is 500ms as suggested by previous research [7].

Observations. It is not clear whether both zoom in gestures are needed; initial user samplings are divided, but not strongly enough to identify a clear preference. However, despite the prevalence of press-and-hold gestures in PDA interactions, we do not feel that it is a fluid gesture for panning. Still, given the relative infrequency of panning, the gesture may be acceptable and is recognized robustly.

Formatting and Editing Text

Properly formatted text can often be the dominant part of a diagram, both visually and in terms of human effort. However, unlike geometry which we attempt to recognize and format as soon as it is entered, we only recognize text but do not typeset it when it is entered. This conscious decision is based on evaluation results for various interactive feedback techniques for entering mathematics [23] in which interactive typesetting was found to be the least efficient and most distracting technique for rapid input, particularly if there are no quality guarantees from the recognizer. Instead we display the output of the Microsoft Windows handwriting recognizer in a 10pt font below the ink strokes similar to [6]. Users can then choose to explicitly convert their handwritten ink into typeset text when they are ready to deal with recognition errors and typesetting details. The typeset gesture was designed to not conflict with common handwriting notations including mathematical equations. It allows users to join two neighboring text elements into a single label by drawing the gesture under both, or to separate joined text elements into two labels by drawing the gesture under each separately (Figure 18). Typeset text can be formatted using a pigtail gesture to invoke a conventional GUI of text formatting controls, or it can be manipulated like other geometric primitives.

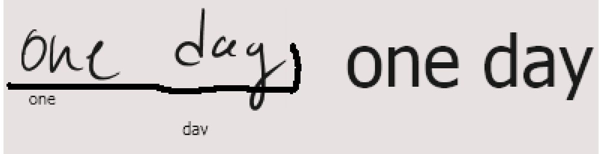

Figure 18. Formatting text. Drawing the format text gesture below two neighboring recognized text ranges unifies them into a single typeset range.

Observations. Although we provide a simple correction interface after text has been typeset, users seem instinctively more inclined to scribble erase and redraw their ink more neatly until the displayed recognition result is reasonable. Also, even though we feel that our current techniques are complete in the sense that text can be formatted quite generally, we think improvements can be made by studying workflow patterns more closely.

DISCUSSION/FUTURE WORK

Pilot users are generally receptive of the interaction style afforded by our Lineogrammer prototype. The GestureBar, although novel, provides familiar-looking support for discovering functionality. The modeless UI, despite occasional disambiguation and snapping errors, is usable in the sense that pilot users seem to enjoy it and are able to work around encountered problems. Pilot users also appear to enjoy "exploring" the snapping behavior and "discovering" strategies, such as scribble drawing. Reaction to the ruler is very enthusiastic and many users play with it at length.

With that said, Lineogrammer is only now nearing the point where it can support more formal usability evaluations in the hands of users who do not have someone to coach them through problems they encounter or about strategies they overlooked. Our input disambiguation, snapping, and ges-

ture recognition algorithms are usable for such preliminary testing, but warrant treatment as research areas in their own right. The GestureBar is also a research area, although our immediate focus is less on its design and more on its content. Initial evaluations revealed that specific choices of icons, words, etc. deeply impact whether users gained an effective understanding of essential gestures and strategies.

At a more fundamental level, we need to address the problem that one of the first things that people try to do with Lineogrammer is draw curved shapes that the system is not able to handle either efficiently or at all. We are well aware that considerable work is required to evolve from our current embryonic support for curves. Nonetheless, we are optimistic that the curve drawing techniques we are considering are at least consistent with our current snapping, selection and manipulation techniques.

CONCLUSION

We have presented Lineogrammer, a novel pen-centric system for creating refined diagrams that seamlessly blends the flexibility and fluidity of low-level, line-oriented drawing with the power of interacting at a higher-level with inferred shapes. We presented effective heuristic approaches for disambiguating gestures, text and geometric drawing elements and for snapping lines to the diagram using a simple, effective dynamic snapping threshold. We also designed a variety of complementary techniques for manipulating diagrams at the vertex, line and shape level. Preliminary testing indicates this is a promising direction and can serve as a baseline for evaluating the need for and progress in deferred beautification.

ACKNOWLEDGMENTS

A Special thanks to Sashi Raghupathy, Andries van Dam and the Microsoft Center for Research on Pen-Centric Computing at Brown University.

REFERENCES

[1]. Arvo, J and Novins, K. *Fluid Sketching of Directed Graphs.* 2006, In Proc. 7th Australasian UI Conference.

[2]. Bragdon, A. *GestureBar: A Training-Free Approach to Disclosing and Teaching Gestures*. Brown University, 2008. Tech. Rep. CS-08-06.

[3]. Calhoun, C., Stahovich, T., Kurtoglu, T., and Kara, L. *Recognizing Multi-Stroke Symbols.* 2002, In AAAI Spring Symposium on Sketch Understanding, pp. 15-23.

[4]. Davis, R. *Magic Paper: Sketch-Understanding Research.* 9, 2007, Computer, Vol. 40.

[5]. Gross, M. and Do, E. *Ambiguous Intentions.* 1996, In Proceedings of UIST'96, pp. 183-192.

[6]. Guimbretière, F., Stone, M., and Winograd, T. *Fluid interaction with high-resolution wall-size displays.* 2001, In Proceedings of UIST '01, pp. 21-30.

[7]. Hinckley, K., Baudisch, P., Ramos, G., and Guimbretiere, F. *Design and Analysis of Delimiters for Selection-Action Pen Gesture Phrases in Scriboli.* 2005, In Proceedings of CHI'05.

[8]. Igarashi, T., Matsuoka, S., Kawachiya, S., and Tanaka, H. *Interactive Beautification: A Technique for Rapid Geometric Design.* 1997, In Proc. of UIST'97.

[9]. Ku, D., Qin, S.F., and Wright, D. *Interpretation of Overtracing Freehand Sketching for Geometric Shapes*. 2006, In Proceedings of WSCG'2006.

[10]. Kurtenbach, G. and Moran, T. *Contextual Animation of Gestural Commands*. 1994, Graphics Interface '94.

[11]. Landay, J. and Myers, B. *Interactive Sketching for the Early Stages of User Interface Design.* 1995, In Proceedings of CHI '95, pp. 43-50.

[12]. Lank, E. *A Retargetable Framework for Interactive Diagram Recognition.* 2003, In Proceedings of ICDAR'03.

[13]. Ohki, Y. and Yamaguchi, Y. *2D Drawing System with Seamless Mode Transition.* 2005, In Proceedings of Smart Graphics, pp. 206-217.

[14]. Patel, R., Plimmer, B., Grundy, J., and Ihaka, R. *Ink Features for Diagram Recognition.* 2007, In Proceedings of SBIM'07, pp. 131-138.

[15]. Paulson, B. and Hammond, T. *PaleoSketch: Accurate Primitive Sketch Recognition and Beautification.* 2008, In Proceedings of Intelligent User Interfaces '08.

[16]. Pavlidis, T. and Van Wyk, C. *An Automatic Beautifier for Drawings and Illustrations.* 1985, Proceedings of SIGGRAPH'85, pp. 225-234.

[17]. Plimmer, B. and Grundy, J. *Beautifying Sketching-Based Design Tool Content: Issues and Experiences.* 2005, In Proceedings of the 6th Australasian Conference on User Interface, pp. 31-38.

[18]. Raisamo, R. *An Alternative Way of Drawing.* 1999, In Proceedings of CHI '99, pp. 175-182.

[19]. Ramos, G. and Balakrishnan, R. *Zliding: Fluid Zooming and Sliding for High Precision Parameter Manipulation.* 2005, In Proceedings of UIST '05.

[20]. Saund, E., Fleet, D., Larner, D., and Mahoney, J. *Perceptually-supported image editing of text and graphics*. ACM Trans. Graph. 23, 3 (Aug. 2004), pp. 728-728.

[21]. Ye, M., Viola, P., Raghupathy, S., Sutanto, H., and Li, C. *Learning to Group Text Lines and Regions in Freeform Handwritten Notes.* 2005, In Proc. of ICDAR'05.

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

[23]. Zeleznik, R., Miller, T., and Li, C. *Designing UI Techniques for Handwritten Mathematics.* 2007, In Proceedings of SBIM'07, pp. 91-98.

[24]. Zeleznik, R. and Miller, T. *Fluid Inking: Augmenting the Medium of Free-Form Inking with Gestures.* 2006, In Proceedings of Graphics Interface '06.