

Fluid Sketching of Directed Graphs

James Arvo* and Kevin Novins†

*University of California, Irvine, USA, arvo@uci.edu, +1 (626) 577-9648

†University of Auckland, novins@cs.auckland.ac.nz, +64 9 373-7599

Abstract

We describe a prototype sketch-based system that allows users to draw and manipulate directed graphs using gestural input exclusively. The system incorporates the notion of *fluid sketching*, which attempts to recognize and beautify what the user is drawing *while* it is being drawn. This concept applies to both the drawing of vertices, which are morphed to circles, and to the drawing of edges, which are approximated on-the-fly by a constrained projection onto low-order polynomials. Consequently, all user-drawn strokes are cleaned up by continuously morphing or projecting them to the nearest geometrically precise shapes. The system has a unique look and feel in that the currently-displayed graph is always at or in transition toward a clean and precise representation of what the user has drawn or is in the process of drawing. When vertices of the graph are dragged to new locations or edges are reshaped, the original graph connectivity is maintained while simultaneously retaining some of its original user-drawn character, such as vertex size and placement, and overall edge shape.

Keywords: Fluid sketching, graph drawing, human-computer interaction, eager recognition, pen-based interface, sketch-based interaction.

1 Introduction

In this paper we present a prototype system that allows users to construct and manipulate directed graphs using pen input alone. This work explores the idea of combining modeless pen-based interaction with eager recognition and both gradual and instantaneous beautification (Arvo & Novins 2000a, Arvo & Novins 2000b) to achieve a new style of interaction.

Traditional graphical user interfaces rely upon user-selected *modes* to determine the semantics of subsequent input events, such as button and pointer events. For example, interactive drawing systems often require the user to select a circle-drawing tool from a menu or control panel in order to draw a circle. The selected tool effectively changes how the system interprets and responds to subsequent mouse clicks and gestures. Mode-based systems have the advantage that the user's intentions are always explicit. However, there is also a price paid in terms of ease-of-use. In particular, the user is forced to punctuate all interaction with mode-selection operations; for example, by selecting new tools using menus or buttons.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Seventh Australasian User Interface Conference (AUIC2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 50. Wayne Piekarski, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

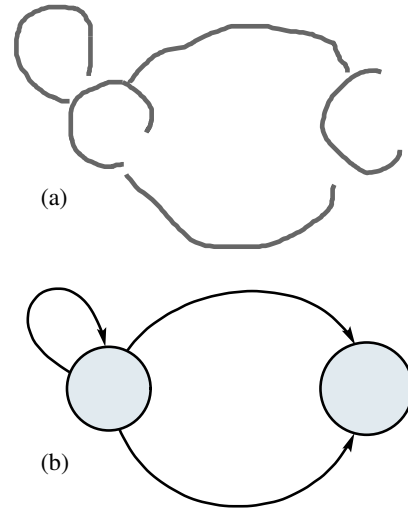


Figure 1: *The user's input strokes are immediately projected onto the best-fit circles and polynomial curves. (a) The raw input strokes generated by the user; first the two strokes representing the vertices were drawn, then the three edges. These strokes are never seen directly by the user of the system; they are shown here to illustrate that very rough approximations suffice. (b) The resulting graph, with cleaned up vertices and edges. This is the style that is shown to the user at every step of construction and manipulation.*

This can become tedious, especially for skilled users. For novice users it also presents the challenge of learning the appropriate modes and how to access them.

Interpretive interfaces offer an alternative to modal systems by using context and/or gesture recognition to infer the user's intentions as best they can. Such interfaces free the user from issuing explicit mode-changing commands and place the onus on the computer. Interpretive interfaces are gaining popularity due to the wide availability of pen-based computing platforms and the steadily-growing computational power of modern machines, which makes sophisticated recognition algorithms practical. Interpretive interfaces have been demonstrated for applications such as geometric and mechanical modeling (Zelevnik, Herndon & Hughes 1996, Stahovich 1998), mathematical notation (Joseph J. LaViola & Zelevnik 2004), and interface design (Landay & Meyers 1995).

There are several inherent challenges in building a good interpretive interface. First, recognition algorithms can never be perfect, even in principle. Recognition errors are inevitable, and must be dealt with quickly and unobtrusively (Smithies, Novins & Arvo 1999). Second, from a human social perspective, an interpretive system can easily be perceived

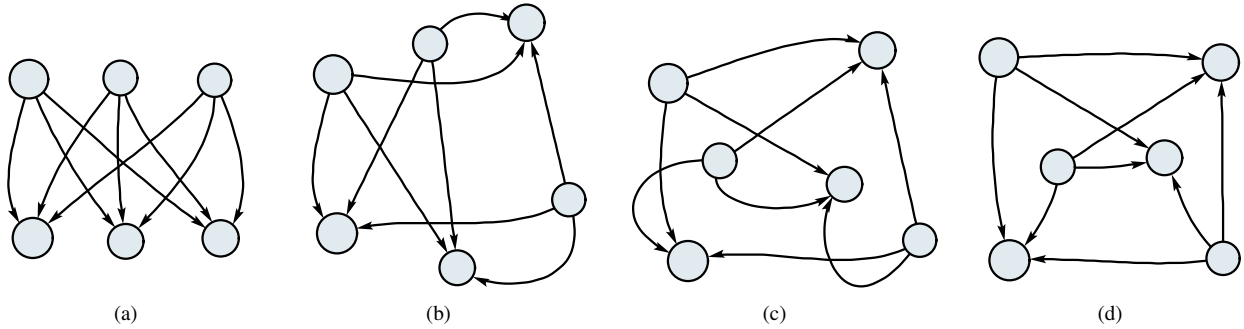


Figure 2: *Once drawn, the graph can be easily manipulated gesturally, again with no modes. (a) A well-known complete bipartite graph known as $K_{3,3}$, as drawn using our system. (b) The same graph after picking and dragging the four right-most vertices. (c) The same graph after further manipulating several vertices and edges. (d) After adjusting more vertex positions and edge shape. The entire process required less than a minute, and was accomplished using pen input exclusively.*

as “rude” if it unilaterally replaces the user’s input with its own representation. This is especially troublesome if the user has no recourse, or if overriding the choices made by the system is cumbersome.

Fortunately, there are several ways in which an interpretive user interface can be given a more natural feel. The approach that we explore here is based on providing a tight recognition-feedback loop that stresses rapid but continuous beautification (Arvo & Novins 2000b, Arvo & Novins 2000a), in addition to providing simple and intuitive methods for changing the appearance of the resulting output. We chose to explore these principles in the context of drawing directed graphs, as such diagrams are ubiquitous in computer science and discrete mathematics, yet they provide an extremely simple domain in which to explore various modes of interaction. Our use of eager beautification in this work stands in sharp contrast to other systems we have developed that seek to maintain the user’s hand-drawn appearance verbatim (Arvo & Novins 2005).

2 Modeless Gestural Interaction

The graph drawing system that we describe in this paper was designed to explore the feasibility of modeless sketch-based interaction. That is, with the exception of several global parameters that are set via a traditional control panel, the user interacts with the system exclusively through pen input on a drawing surface; there are no modes needed to distinguish between drawing and editing, for example, and no clicking on buttons or menus is required. See Figures 1 and 2 for samples of output from the system; in each case the resulting graphs were generated and manipulated using pen input alone.

Our system is based on four basic operations, which suffice for creating and manipulating directed graphs. These are, 1) creating vertices, 2) creating edges, 3) moving vertices and 4) reshaping edges.

Each operation is initiated by a *pen down* event; that is, by touching the stylus to the drawing surface, whether it be a tablet computer or a digitizing pad. The four operations are uniquely identified by the context of the pen-down event; that is, by the proximity of the event to other elements that are already displayed. For example, if the pen-down event occurs at a point that is not adjacent to any existing vertices or edges, the ensuing gesture is interpreted as the specification of a vertex. If a gesture starts on the boundary of an existing vertex, an edge is created that approximately follows the path of the pen. If the gesture starts in the interior of a vertex, it is interpreted as a pick-and-drag operation on that vertex.

Finally, if the gesture starts on an existing edge, it is interpreted as a edge reshaping operation.

Thus, each distinct category of gesture can be robustly and unambiguously recognized based on its starting point alone. Additional input and editing capabilities will make this task more challenging, as they will no doubt entail recognition of the subsequent gesture and thus introduce more opportunities for recognition errors. However, these four categories, based on starting point, will remain an important element of the recognition phase. We describe each of these operations in the following subsections.

2.1 Drawing Vertices

The edges of a graph are defined by their adjacent vertices. Thus, the first step in creating a graph is to create one or more vertices. In our system one creates a vertex by drawing a circle, or a rough approximation to a circle. Any user-drawn stroke that is drawn in an empty portion of the canvas, away from other vertices and edges that may already exist, is assumed to be a vertex. The system immediately performs a least-squares approximation to find the most likely circle that corresponds to what the user has thus far drawn, and then begins to morph the user-drawn line toward that ideal circle. This is a form of eager recognition in that it occurs well before the user has drawn a complete circle. In fact, only a portion of the circle is necessary to define it, as shown in Figure 1. This technique of eagerly recognizing a shape and beginning to morph the user-drawn line to that shape *while* the user is still drawing it is referred to as *fluid sketching*. The authors previously introduced this technique to allow users to draw perfect geometric shapes using only approximate sketches, and to get immediate and continuous feedback while doing so (Arvo & Novins 2000a). While this general approach can easily accommodate a variety of shapes, in the current system we utilize only its ability to simplify the drawing of circles, as shown in Figure 3.

In the context of our current system, the notion of fluid sketching is a convenient approach for drawing vertices, which are depicted by circles of arbitrary radius. As the user begins to draw a vertex, the curving stroke is continuously adjusted toward a perfect circle whose radius and center are the best approximation to what has been drawn thus far. This is accomplished by continually migrating each point of the user-drawn stroke toward the corresponding point of the current ideal shape. By the time the user has drawn half the circle, the system has already morphed it and partially shaded it. When the user lifts the pen, the system completes the vertex boundary

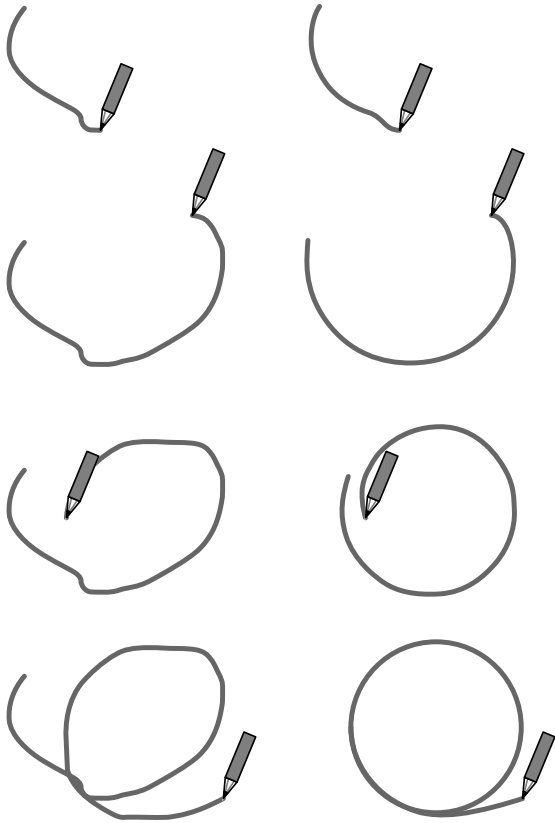


Figure 3: An example of fluid sketching. (left) The raw input drawn by the user. (right) The very same input continuously morphed toward the current least-squares approximation of a circle.

and fills it. This approach is commensurate with the goal of having the system always present a clean and precise diagram, despite any sloppiness or imprecision on the part of the user. The contrast between what the user actually draws and what the system displays can be seen in Figure 1. We use a related strategy for drawing edges, which we describe in the next section.

2.2 Drawing Edges

In our system, edges are specified by beginning a gesture near the edge of an existing vertex, then drawing toward another vertex. We approximate the path of the user’s pen with a low-order polynomial, thus smoothing it while retaining something of its general character. The result of this smoothing can be seen in Figure 1.

For the purpose of approximating such a curve, and getting it to meet its adjacent vertices approximately orthogonally, we first augment the user-drawn line at the beginning and end with the centers of its adjacent vertices. The augmented line has several important roles internally; for example, it is useful in stretching and compressing edges as vertices move, as it allows the edges to “pivot” about the center of its adjacent vertices which, in turn, allows the points at which the edge meets them to drift as needed (Arvo & Novins 2005). This can be observed in Figure 6. The additional line segments with which the edges are augmented are never displayed, however.

The user-drawn edges are continuously projected onto a low-order polynomial basis, even as they are being drawn. This is accomplished by matching a collection of points on the polynomial curve with corresponding points on the user curve as closely as possible, as depicted in Figure 4. This approxima-

tion process would amount to nothing more than a trivial least-squares projection if it were not for the constraints that must be imposed at each end point. That is, because the curve is required to meet the beginning and ending points with which it was augmented, the problem becomes a *constrained least squares projection*. This problem can nonetheless be expressed as least-squares, but only after the hard constraints are explicitly accounted for. A complete derivation of the resulting constrained least-squares projection is provided in Appendix A.

The parameter k , which specifies the order of the approximating curves, must be set by the user and is one of the only exceptions that currently requires traditional control panel interaction in our system. Low-orders, such as 3 and 4, produce extremely smooth curves which are incapable of bending multiple times. Higher-orders, such as 6 and above, allow for much more detail to be retained, as shown in Figure 5. In our current system, this order is a global parameter that applies to all curves, and reflects the user’s preference for smoothness versus expressiveness.

2.3 Repositioning Vertices

The repositioning of vertices in themselves presents no difficulty at all; it requires nothing more than a pick, drag, and drop sequence. The challenge, however, is in dealing with edges that may be adjacent to the moving vertex, as we wish to retain the semantics of the graph as well as some of the original character of the edges as they are reshaped.

The majority of graph editing systems allow only straight edges, thus edge geometry is completely determined by its adjacent vertices. When dealing with curved edges, however, there are more degrees of freedom. Thus, when a vertex is moved, there are infinitely many ways in which the adjacent edges can be altered so that connectivity is retained. We must decide how best to utilize this extra flexibility.

As with our earlier work (Arvo & Novins 2005), we have found it useful to perform edge transformations in a coordinate system that is defined by the *baseline* of the edge, which in this context means the straight line connecting the centers of its adjacent vertices. As a vertex is translated, the baseline of each adjacent edge is rotated, stretched, or compressed. Expressing the user’s edge in a baseline coordinate system allows us to effectively factor out the rigid part of the transformation. What remains is handling stretching and compression of the baseline.

The fundamental rule that we shall impose is that the transformed edge should approximately preserve its arc length as it is transformed, as this results in a physically plausible and intuitive behavior. As an edge is stretched, it must therefore become smoother, ultimately flattening into a straight line. Compression should cause the edge to bend into a loop that takes up the slack. See Figure 6.

These behaviors can be obtained quite easily by expressing the points along the original user-drawn line in terms of offsets from corresponding points along the baseline. As the curve is stretched, the points along the baseline are also stretched, and the offsets from these points are simultaneously down-weighted, reaching zero when the distance between the vertices reaches the original arclength of the curve. This strategy causes the line to gradually flatten in a way that keeps arc length roughly constant. The strategy is similar when an edge is compressed by moving its adjacent vertices closer together. In the this case, however, we ramp the offset weights toward a value that creates a loop of the correct arc length. The use of offsets that are adjusted as the edge is

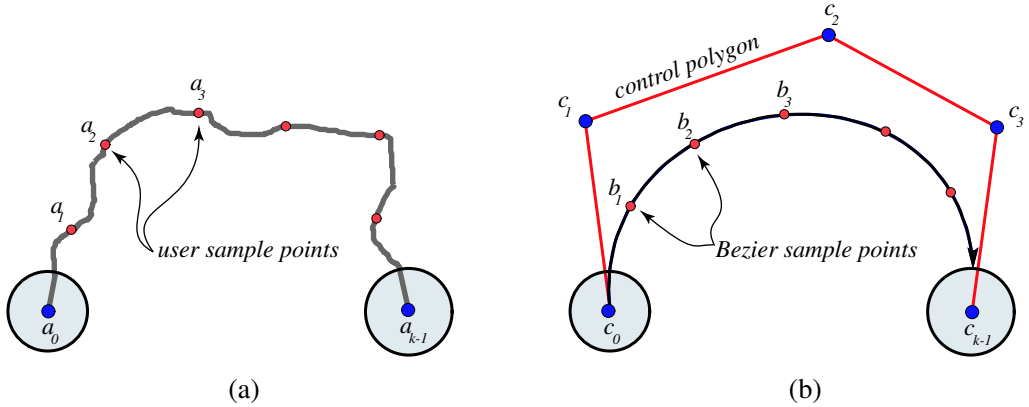


Figure 4: User-drawn edges are projected onto low-order polynomial curves to make them smooth and clean. (a) The user’s stroke is first sampled at n positions that are uniformly spaced with respect to arc length. (b) The best-fitting Bézier curve of order k is then computed by performing a least squared fit to the user line based on m samples that are uniformly spaced in parameter space. The result of this process is a collection of k control points, the first and last of which correspond to the centers of the adjacent vertices.

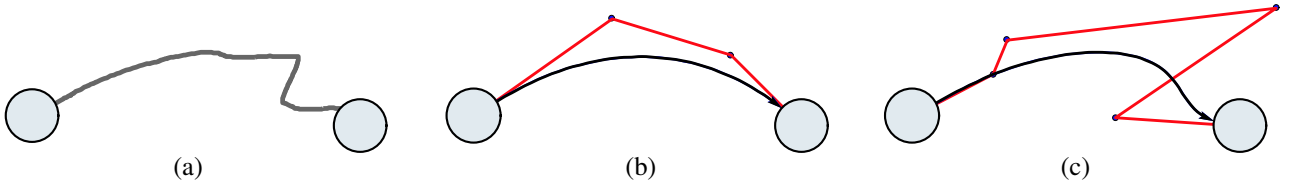


Figure 5: The order of the Bézier projection can be set so as to follow the user’s curve more or less closely. (a) The user’s stroke representing an edge. (b) A Bézier curve of order four (with two intermediate control points) filters out almost all of the variation. (c) A Bézier curve of order six (with four intermediate control points) captures more of the original shape.

stretched or compressed is equivalent to the the point-wise linear interpolation strategy that the authors employed in a similar graph drawing system (Arvo & Novins 2005).

A desirable side effect of our edge morphing scheme is that edges will regain their original shape when their adjacent vertices return to their original separation distance. That is, an edge will retain a “memory” of its original shape, to which it returns given the opportunity. One advantage of this *shape memory* is that all stretching and compression is exactly reversible, so that repeated adjustments of vertex positions do not cause the shape of adjacent edges to drift. It also adds to the intuitive behavior of edges as vertices move, as it is reminiscent of a spring returning to its rest shape. These effects can be observed in Figure 6.

2.4 Reshaping Edges

While the edge projection algorithm described in Section 2.1 fits curves to the user input, further editing of the edge shapes is often necessary as a result of adding more edges, moving the vertices, etc. One common approach to curve editing is to allow the user to adjust the control points. However, direct manipulation of the curves is much better suited to our system, which encourages direct manipulation in every other context.

In keeping with the modelless philosophy of our system, we allow the user to simply pick any point on an existing edge and drag it. Unfortunately, such a manipulation is ambiguous; there are infinitely many changes that one could make to the curve that would be consistent with such a dragging gesture. We therefore opted for a simple two degree-of-freedom transformation comprised of a one-dimensional shear and a one-dimensional scaling transformation, which are applied to the entire curve.

As a point on an existing edge is dragged, the starting point p , the current pen position q , and the baseline of the edge completely determine the shearing and scaling that are to be applied to the curve. As with the vertex movement operation in Section 2.3, our transformations take place in the baseline coordinate system, with respect to which p and q become p' and q' . Once in this coordinate system, we seek the unique transformation that carries p' to q' using only scaling perpendicular to the baseline and shearing parallel to the baseline. This amounts to solving for a and b using the matrix equation

$$\begin{bmatrix} 1 & a \\ 0 & b \end{bmatrix} p' = q', \quad (1)$$

where a is the skew, and b is the scale factor. The resulting 2D matrix, which is trivially solved for, is then applied to the entire edge in baseline coordinates, which does not alter the position of its end points. The resulting transformation is surprisingly effective on low-order curves (Figure 7), but is less so on higher-order curves for which local degree-lowering or degree-raising operations would be desirable.

3 Conclusion

We have demonstrated a modelless sketch-based system for creating and manipulating directed graphs, or *digraphs*. The system employs a previously-introduced approach called *fluid sketching* to continuously recognize and morph strokes that are intended to represent circular vertices. In a similar spirit, edges are continuously cleaned up by projecting them onto the closest approximating polynomial curve, whose order is specified by the user as a global parameter. As a consequence of these policies, the graph that is displayed is never far from the best-fitting clean digram, where vertices are perfect circles, edges are

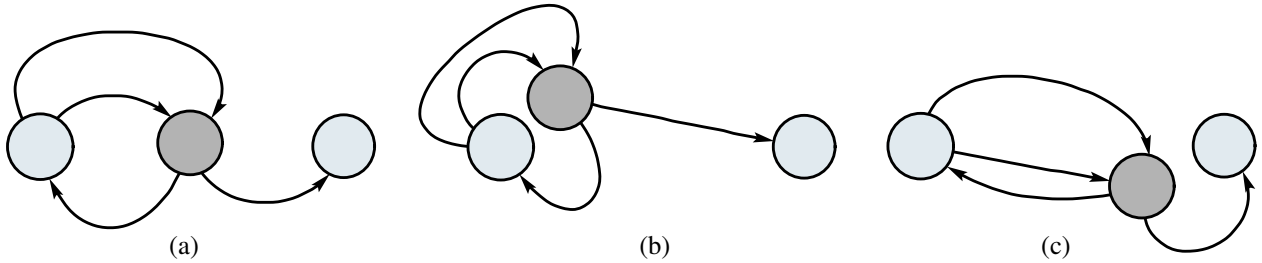


Figure 6: When a vertex is moved, all adjacent edges are adjusted so as to maintain the original connectivity, retaining as much of the original edge shape as possible. (a) A simple graph before moving a vertex. (b) The same graph after moving the shaded vertex up and to the left. Three edges get compressed, and one is stretched. (c) The same graph after moving the shaded vertex down and to the right. Three edges get stretched, and one is compressed.

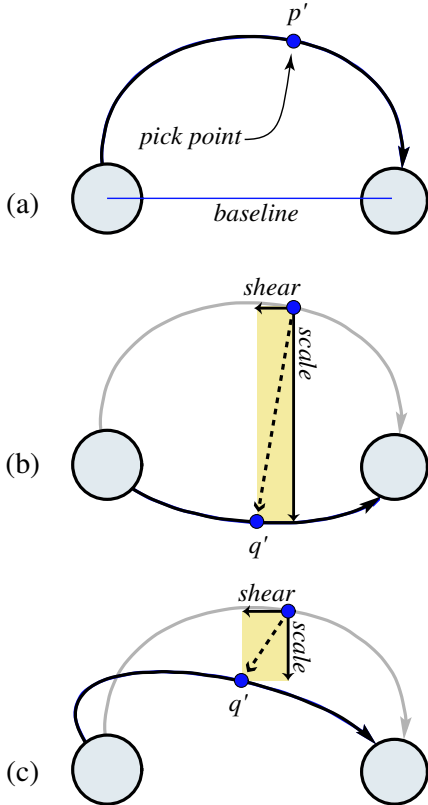


Figure 7: The shapes of edges can be modified in a limited way by picking and dragging a point on the curve. (a) The user picks a point p on an existing curve, which is p' in baseline coordinates. (b) Dragging p' to a new position q' uniquely defines an affine transformation consisting of a vertical scale and a horizontal shear. (c) A configuration with a very pronounced shear.

smooth curves, and vertex-edge adjacencies are precise. No modes are required during drawing, as the intent of each stroke is inferred from the current context; isolated strokes are interpreted as vertices, while strokes beginning near an existing vertex are assumed to be edges.

As soon as any portion of a graph is drawn, it can be manipulated using pen-input alone; again, no modes are necessary as the intent of each stroke can be inferred from the context of the diagram. When vertices are dragged, all adjacent edges are adjusted so that adjacency is maintained, yet the original shape is still approximated. This is accomplished with a simple heuristic for stretching and compressing edges that keeps arc length approximately constant. Edges

are also imbued with shape memory, so that they tend to return to their original shape even after repeated stretching and compression. When an edge is dragged, its shape is altered by means of scaling and shearing. New elements can be drawn at any time; thus drawing and editing operations can be freely mixed, as the system can easily disambiguate them without user-selected modes. We believe that the system, even in its current prototype form, offers a unique style of interaction that is conducive to quickly drawing and manipulating high-quality representations of digraphs.

4 Future Work

While our current system provides an interesting demonstration, there are many features we wish to add to make it more robust and useful. First, before the system can support a real application, we will require several new editing operations and finer user control over vertex and curve shapes, as well as the ability to supply annotations for vertices and edges. It will be challenging to maintain our modeless credo as the range of user input and editing options grows (Rossignol, Willems, Neumann & Vuurpijl 2004).

The system could also be more clever about its beautification. Ideally, the system should automatically determine the degree of the Bézier curve to use based on the complexity of the user-drawn curve, and allow the user to make adjustments that raise or lower the degree of each curve. The latter will require that the system distinguish between local and global shape editing operations. It also should be able to assist the user in finding vertex placements and edge paths that avoid intersections or meet other aesthetic criteria.

Finally, the system should be connected to a computational back end that will interpret the graph as a finite state machine, or other automaton. One of our ultimate aims in pursuing this work is to produce a convenient tool for teaching automata theory, in which labeled directed graphs are the preferred representation for both finite state machines, pushdown automata, and Turing machines.

Within the HCI community opinion is still divided as to whether recognition should be eager or lazy, and whether beautification should be performed eagerly, lazily, or at all (Plimmer & Grundy 2005). In our own work we have experimented with several dramatically different approaches, and find that each has its own appeal (Smithies et al. 1999, Zanibbi, Novins, Arvo & Zanibbi 2001, Arvo & Novins 2005). Ultimately, the best solution may entail a plurality of styles, arbitrated either by context or by personal preference. In any case, combining modeless interaction with eager beautification, as we have demonstrated here, appears to be a fruitful direction to pursue.

References

- Arvo, J. & Novins, K. (2000a), Fluid sketches: Continuous recognition and morphing of simple hand-drawn shapes, in ‘Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology’, San Diego, California.
- Arvo, J. & Novins, K. (2000b), Smart text: A synthesis of recognition and morphing, in ‘AAAI Spring Symposium on Smart Graphics’, Stanford, California, pp. 140–147.
- Arvo, J. & Novins, K. (2005), Appearance-preserving manipulation of hand-drawn graphs, in ‘Proceedings of GRAPHITE 2005’, Dunedin, New Zealand.
- Joseph J. LaViola, J. & Zeleznik, R. C. (2004), ‘Mathpad2: a system for the creation and exploration of mathematical sketches’, *ACM Transactions on Graphics (TOG) archive* **23**(3), 432–440.
- Landay, J. & Meyers, B. (1995), Interactive sketching for the early stages of user interface design, in ‘Proceedings of Computer-Human Interaction ‘95’, pp. 34–50.
- Plimmer, B. & Grundy, J. (2005), Beautifying sketching-based design tool content: Issues and experiences, in ‘Proceedings of the Sixth Australasian User Interface Conference (AUIC2005)’, Newcastle, Australia.
- Rossignol, S., Willems, D., Neumann, A. & Vuurpijl, L. (2004), Mode detection and incremental recognition, in ‘Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition (IWFHR-9 2004)’, Tokyo, pp. 597–602.
- Smithies, S., Novins, K. & Arvo, J. (1999), A handwriting-based equation editor, in ‘Proceedings of Graphics Interface ‘99’, Kingston, Ontario, pp. 84–91.
- Stahovich, T. F. (1998), ‘The engineering sketch’, *IEEE Intelligent Systems* **13**(3), 17–19.
- Strang, G. (1986), *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, Massachusetts.
- Zanibbi, R., Novins, K., Arvo, J. & Zanibbi, K. (2001), Aiding manipulation of handwritten mathematical expressions through style-preserving morphs, in ‘Proceedings of Graphics Interface 2001’, Ottawa, Ontario, pp. 127–134.
- Zeleznik, R. C., Herndon, K. P. & Hughes, J. F. (1996), SKETCH: An interface for gestural modeling, in ‘Computer Graphics Proceedings’, Annual Conference Series, ACM SIGGRAPH, pp. 163–170.

A Polynomial Approximation

Here we provide the details of the projection algorithm used to approximate user-drawn edges. Two parameters, k and n , affect the process and are assumed to be provided; k denotes the order of the polynomial that we wish to project onto, and n denotes the number of samples along the user-drawn curve that we use to approximate the curve, as shown in Figure 4. We require that $n > k$, which results in an over-determined system that we solve using least squares. Our task is to find an order- k polynomial curve that interpolates the centers of two vertices

while approximating a sequence of $n - 2$ points along the user-drawn curve. We shall do this by finding the $k - 2$ interior control points of an order- k Bézier curve.

Let a_0, \dots, a_{n-1} denote n samples along the user-drawn curve that are uniformly spaced according to arc length, and let b_0, \dots, b_{n-1} denote n samples along an order- k polynomial curve that are uniformly spaced according to the parameter $t \in [0, 1]$. We wish to find k control points that generate the polynomial curve such that the resulting points b_0, \dots, b_{n-1} best approximate a_0, \dots, a_{n-1} . See Figure 4. To express this as a system of linear equations, we define the $n \times k$ matrix \mathcal{B} by

$$\mathcal{B}_{n \times k}(i, j) \equiv \binom{n-1}{j} t_i^j (1-t_i)^{k-1-j} \quad (2)$$

for $i = 0, \dots, n-1$, and $j = 0, \dots, k-1$, where

$$t_i \equiv \frac{i}{n-1} \quad (3)$$

are uniformly spaced points in $[0, 1]$. This matrix maps the k control points c_0, \dots, c_{k-1} to n samples along the resulting Bézier curve. That is, if \mathbf{C} denotes the $k \times 2$ matrix whose rows are c_0, \dots, c_{k-1} , then $\mathcal{B}_{n \times k} \mathbf{C}_{k \times 2}$ is an $n \times 2$ matrix whose rows are the coordinates of points along the Bézier curve that are uniformly spaced with respect to $t \in [0, 1]$. To solve for the unknown control points, we write the matrix \mathcal{B} in block form,

$$\mathcal{B}_{n \times k} = \begin{bmatrix} 1 & 0 & 0 \\ u & M & v \\ 0 & 0 & 1 \end{bmatrix}, \quad (4)$$

where both u and v are column vectors. Thus, we have

$$\begin{bmatrix} 1 & 0 & 0 \\ u & M & v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_0^\top \\ C \\ c_{k-1}^\top \end{bmatrix} = \begin{bmatrix} b_0^\top \\ B \\ b_{n-1}^\top \end{bmatrix}, \quad (5)$$

where C is an unknown $(k-2) \times 2$ submatrix of control points, and B is an $(n-2) \times 2$ submatrix encoding points on the resulting curve. We wish to find the matrix C that minimizes the norm of the difference

$$\begin{bmatrix} b_0^\top \\ B \\ b_{n-1}^\top \end{bmatrix} - \begin{bmatrix} a_0^\top \\ A \\ a_{n-1}^\top \end{bmatrix}. \quad (6)$$

Imposing the constraints that $a_0 = b_0 = c_0$ and $a_{n-1} = b_{n-1} = c_{k-1}$, which force the Bézier curve to pass through the end points a_0 and a_{n-1} , allows us to isolate the matrix C . It follows from Equations (5) and (6) that we must choose the matrix C so that

$$MC \approx A - uc_0^\top - vc_{k-1}^\top. \quad (7)$$

The problem can now be solved using classic least squares. Forming the well-known normal equations (Strang 1986) we obtain

$$C = (M^\top M)^{-1} M^\top (A - uc_0^\top - vc_{k-1}^\top), \quad (8)$$

which specifies the unknown control points in terms of known quantities. Coupled with the given control points, c_0 and c_{k-1} , we obtain a control polygon for the order- k Bézier curve that interpolates the end points while approximating the user-drawn curve at the intermediate points.