Draft: Final version published in Journal of Graphics Tools, 13(2), pp. 37-37, 2008.

Efficient Curve Fitting

Sarah F. Frisken
Tufts University

Abstract.
We propose a method for fitting a piecewise parametric curve to a sequence of digitized points such as those acquired from a computer mouse or digitizing pen. The fitting is done on-the-fly rather than after a complete sequence of points has been acquired, thereby providing high quality parameterized curves for immediate display during drawing. The method is fast, accurate, and robust, handles complex input paths, maintains corners, and ensures at least G^1 continuity at non-corner points. The method uses *vector distance fields*, which we describe here, to represent the intended input path of the digitized points in order to achieve its quality and performance. Detailed pseudocode is available online.

1. Introduction

Drawing 2D curves using a computer mouse or digital pen is important for applications ranging from sketching for illustration and design to signature capture for identity verification. While such input devices typically provide a sequence of digitized points, applications frequently require input in the form of an analytic curve. Unfortunately, current methods for fitting curves to points can be complex, often require significant preprocessing of the digitized points, and can fail, especially when the path of the input points is complicated and self-intersecting. In addition, current methods typically require a full sequence of digitized points (e.g., all the points recorded along the path of a digital pen between pen-down and pen-up events) prior to determining the analytic curve. This forces applications to draw an approximation of the analytic curve until the full sequence of points is available (e.g., Adobe Illustrator 10 draws the sequence of digitized points, while Microsoft's PowerPoint 2003 draws a polyline connecting the sequence of digitized points) and may result in a noticeable delay between the pen-up event and generation of the analytic curve and/or a noticeable shape change when the generated analytic curve replaces the approximation.

Here we present a new method for fitting an analytic curve to a sequence of digitized points. The fitting is done on-the-fly, i.e., an analytic curve is initialized and then updated as each digitized point is acquired rather than after a full sequence of points has been recorded. The method is robust and fast, thereby providing immediate feedback during curve drawing. The method is also accurate – the application can specify the maximum deviation of the analytic curve from the input path to balance competing requirements for accuracy, smoothness, and representation size – and it handles complex input paths while automatically detecting and maintaining corners and ensuring at least G^1 continuity at non-corner points.

2. Background

When drawing with a computer mouse or digital pen, the path of the input device is sampled, the sampled points are typically quantized to integer pixel locations, and these digitized points are made available to the application. While some applications simply represent the input path by the sequence of digitized points, converting the digitized points to an analytic curve has several advantages: 1) an analytic curve requires less memory than a list of digitized points; 2) the resultant analytic curve can be scaled, rotated, deformed, etc. without degrading the quality of the rendered path; 3) the application can enforce smoothness and continuity constraints on an analytic curve (e.g., to eliminate hand jitter during drawing or to reduce quantization errors); and 4) users can edit an analytic curve via control point manipulation rather than by erasing and redrawing points along the input path.

Although there are many different analytic curve representations (e.g., see [2]), curve drawing applications typically use piecewise polynomial curves. A piecewise polynomial curve is composed of multiple polynomial curve segments. When fitting a piecewise polynomial curve to a sequence of digitized points, the goal is to determine an optimal set of curve segments, where optimal may mean some combination of a minimum number of curve segments, a minimum error between the curve segments and the input path, and curve segments that enforce a number of other constraints such as curve continuity or maintaining intended corners of the input path.

The method we present can be applied to general analytic curves but here we focus on piecewise parametric curves whose curve segments are cubic Bezier curves. Cubic Bezier curves provide a good balance between complexity and flexibility and are used in many commercial drawing and printing systems. A cubic Bezier curve is defined by four control vertices, two on-curve control vertices C_0 and C_3 at the endpoints of the Bezier curve, and two off-curve control vertices C_1 and C_2. A point on the Bezier curve can be expressed in terms of these four control vertices by the cubic polynomial, $B(t) = C_0(1-t)^3 + 3C_1t(1-t)^2 + 3C_2t^2(1-t) + C_3t^3$, where the parameter t varies from 0 at the first curve endpoint C_0 to 1 at the second curve endpoint C_3.

Ahn presents a review of standard approaches for fitting curves and surfaces to a set of digitized points [1]. In the case of piecewise polynomial curves, the curve fitting problem reduces to finding a set of control vertices for the curve segments that minimizes the geometric, or Euclidean, distance between the digitized points and the fit curve. Ahn observes that the geometric distance is a non-linear function of the control vertices and that the task of computing and minimizing the sum of squared geometric distances is highly complex. He surmises that the curve fitting problem is essentially a non-linear optimization problem which should be solved using iteration. Given a sequence of digitized points $\{P_i\}$, i = 1, 2, … N, a typical iterative approach for curve fitting applies the following steps:

1. Start with a simple initial estimating curve (e.g., a straight line segment connecting the endpoints of the sequence) and an initial set of “minimum distance points” $\{Q_i\}$, i = 1, 2, … N, where each Q_i is a point on the estimating

curve that is ‘closest’ to its corresponding digitized point P_i.

2. Iteratively adjust control vertices of curve segments of the estimating curve to reduce the fitting error, where the fitting error is typically estimated as the sum of squared distances between each $\{P_i, Q_i\}$ pair. For each iteration, the set of minimum distance points $\{Q_i\}$ is re-computed (where, typically, the re-computation also requires an iterative approach).
3. If necessary, subdivide the estimating curve into additional curve segments.
4. Repeat steps 2 and 3 until the fitting error is acceptable.

Re-computing the minimum distance points in step 2 must be done for each iteration of the control vertex adjustment. Unfortunately, this inner loop is generally the most time consuming part of the algorithm. As described by Ahn [1], there are two basic approaches for finding the minimum distance points: 1) determine the closest point Q_i on the estimating curve for each digitized input point P_i directly (which requires solving a 5th order polynomial for each cubic Bezier curve segment of the estimating curve) using an iterative polynomial root finder (e.g., Newton-Raphson or Bezier clipping [10]), or 2) determine a parameter value t_i for each digitized point P_i so that $Q(t_i)$ is the ‘closest’ point on the piecewise parametric estimating curve to P_i. When using the second approach, the parameterization of the digitized points is typically initialized using chord length parameterization of the estimating curve and then adjusted iteratively using a polynomial root finder (e.g., see [12] and [13]).

The standard curve fitting approaches suffer from a number of drawbacks. One such drawback is that standard approaches are designed to operate only on a full sequence of digitized points, e.g., all of the points recorded along a single input path. Because the fit curve is not determined until the input path has been completed, an approximation of the input path, such as the digitized points themselves or a polyline connecting the digitized points, must be drawn to provide feedback to the user. This can result in a delay after the input path is complete and/or a noticeable change in the shape of the drawn path when the approximation of the input path is replaced by the fit curve. A second major drawback is the costly inner loop for determining minimum distance points. The computation involved in the inner loop is proportional to the number of points in the sequence of digitized points. Preprocessing can be used to reduce the number of digitized points as well as to remove noise such as hand jitter from the input data. However, preprocessing is also time consuming and can result in the loss of intended detail. (Ideally, the preprocessing should detect and preserve intended corners in the input path as in [12].) Other problems with standard approaches occur because of a lack of robustness in the iterative methods for determining minimum distance points. These methods are sensitive to local minima, particularly for complex, self-intersecting input paths, and require a good initial set of minimum distance points.

Our method for fitting a piecewise polynomial curve to a sequence of digitized points uses *distance fields* for representing the input path in order to reduce time spent in the inner loop, thereby allowing us to fit a curve to the input path *on-the-fly* to provide immediate feedback as the curve is drawn, rather than after a full sequence of digitized points has been recorded. A distance field of a shape measures, for any point in space, the distance from that point to the closest point on the shape. Distance fields have been used for many applications in computer graphics, CAD/CAM, computer vision, and robotics. See [7] for a general review of the use of distance fields in computer graphics and vision and [6] for a review of the use of distance fields in shape modeling. By representing the input path by a sampled distance field, we can determine the distance from any test point on the estimating curve to the input path simply by interpolating the sampled distance field at the test point, a trivial operation when compared to the iterative searches of standard curve fitting methods.

Conventionally, distance fields are *scalar fields* (e.g., representing the scalar Euclidean distance from a point to a shape). We use *vector distance fields* (described in section 3.1) which represent the distance at any point as a vector value. Vector distance fields are more suitable than scalar distance fields for representing shapes that do not have a well defined inside and outside (e.g., our input paths) and for algorithms that require the gradient of the distance field (e.g., our approach for adjusting control vertices). Finally, we present a fast method for updating the vector distance field of the input path as each digitized point is provided to the application.

3. Efficient Curve Fitting Using Vector Distance Fields

Our goal is to fit a piecewise polynomial curve composed of cubic Bezier curve segments to a sequence of digitized points representing an input path on-the-fly, i.e., as each digitized point is provided to the application. During curve fitting, the following objectives are considered: the fit curve should satisfy a maximum curve fitting error specified by the application; a minimum number of curve segments is desired; the fit curve should pass through endpoints of the input path but does not need to pass through other points in the sequence of digitized points; intended corner points in the input path should be detected and preserved; and, except at corner points, the fit curve should have at least G^1 continuity. Because curve segments along the path are finalized before the end of the path is reached, our method provides a set of curve segments that are locally optimal, satisfying accuracy and smoothness requirements while attempting to generate as few curve segments as possible, but may not be globally optimal.

Standard curve fitting approaches measure the distance *from each digitized point on the input path to the estimating curve*, which requires an expensive iterative step inside the inner loop for finding the closest point on the estimating curve for each digitized point. In our approach, we reverse this paradigm and measure the distance *from a set of test points on the estimating curve to an approximation of the input path*, i.e., to the polyline connecting the sequence of digitized points. This paradigm shift increases the efficiency of curve fitting because: 1) the polyline is represented as a sampled vector distance field so the distance from test points to the polyline can be computed directly and efficiently via linear interpolation, thereby avoiding the costly inner loop for determining minimum distance points; 2) the polyline's vector distance field changes only incrementally as each new digitized point is added to the polyline and does not change during iterative adjustment of the estimating curve (i.e., it does not change until the next digitized point is acquired), and 3) the number of test points along the estimating curve can be significantly smaller than the number of digitized points without compromising quality, thereby avoiding the need for preprocessing to reduce the number of digitized points. As long as the digitized points are closely spaced (common for typical input devices), the polyline is an effective approximation of the input path. We can ensure that the fit curve will smoothly approximate the input path by enforcing G^1 continuity of the estimating curve and using a non-zero maximum curve fit error (e.g., an error of 1-2 pixels) so that the estimating curve is not forced to fit the polyline exactly.

Our method for curve fitting applies the following steps which are fully described in sections 3.1 through 3.4. Detailed pseudocode code is available online at the address listed at the end of this paper.

1. Initialize a first 2D cubic Bezier curve segment of the piecewise polynomial estimating curve by setting the position of all of its control vertices to the position of the first digitized point in the sequence.
2. Clear a 2D vector distance field for representing the portion of the polyline corresponding to the current curve segment.
3. Repeat the following sub-steps as each new digitized point is made available
    - Test for a corner between the previous endpoint of the current curve segment and the new digitized point by comparing the angle between the tangent vector at the endpoint of the current curve segment and the line segment from the endpoint to the new digitized point to a maximum corner angle (typically 45 to 60 degrees). If a corner exists, finalize the current curve segment, clear the 2D vector distance field representation, and initialize a new curve segment. Require only C^0 continuity at the first endpoint of the new curve segment.
    - Update the vector distance field representation to incorporate the new digitized point.
    - Adjust control vertices of the current curve segment to reduce the distance between the estimating curve and the polyline. Enforce G^1 continuity with the previous curve segment if required.
    - Test for an inadequate curve fit. If, after adjusting the control vertices, the error of the estimating curve exceeds a maximum fitting error, undo the control vertex adjustments, finalize the current curve segment, clear the 2D vector distance field representation, and initialize a new curve segment. Require G^1 continuity at the first endpoint of the new curve segment.

3.1 Vector Distance Fields

The 2D *vector distance* (dx, dy) from any given sample point (x, y) in $\Re^2$ to a closest point (u, v) on a 2D shape is defined to be the 2D vector from the sample point to the closest point, i.e., (dx, dy) = (u - x, v - y). The extension to 3D and higher dimensional

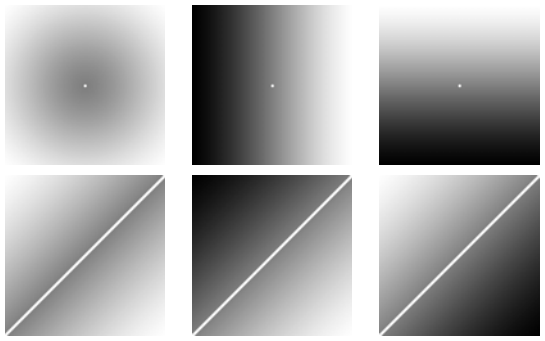

Figure 1. In these figures, points and lines are rendered white, negative distances increase in magnitude from mid-gray to black and positive distances increase in magnitude from mid-gray to white. Mid-grays represent distance values close to zero. Top left: the scalar distance field of a point is non-linear everywhere and non-differentiable at the point. Top center and right: the x and y components (respectively) of the vector distance field of a point are linear and differentiable everywhere. Bottom left: the scalar distance field of a line is non-differentiable on the line. Bottom center and right: the x and y components (respectively) of the vector distance field of a line are linear and differentiable everywhere.

vector distances is straightforward. The magnitude of the vector distance, i.e., $(dx^2 + dy^2)^{½}$, is the minimum Euclidean distance from the sample point to the shape, and the vector distance itself is equal to the unit gradient vector of the Euclidean distance field at the sample point scaled by the minimum Euclidean distance to the shape.

A vector distance field of an object represents, at any point in space, the vector distance from that point to the object. Vector distance fields were introduced for evolving surfaces via level sets in [3] and for shape representation in [5]. Vector distance fields are particularly well suited for representing shapes that do not have a well defined inside and outside (e.g., points, lines, curves, and infinitely thin surfaces) because each component (e.g., dx, dy) of the vector distance varies smoothly from negative to positive from one side of the shape to the other. In contrast, scalar distance fields of such shapes are non-differentiable at points on the shape so that, for example, we could not use linear interpolation to locate points *on* the shape (where the scalar distance is zero) from sample points that *span* the shape since sampled values on opposite sides would have the same sign.

Vector distance fields are also well suited for applications that require the gradient of the distance field. In contrast to scalar distance fields, where the gradient is determined using a higher order and less accurate gradient interpolation function such as the central differences operator, with vector distance fields the gradient can be interpolated directly and more accurately from sampled vector distances using bilinear interpolation.

The 2D vector distance field of the input path can be represented by a regularly sampled 2D array of vector distances, which we refer to as a *vector distance map* (other representations could be used to decrease memory requirements and increase processing speed). Vector distances between sample points in a vector distance map can be reconstructed using bilinear interpolation of each component of the vector. When computing the vector distance map of a polyline representing the input path, we exploit the fact that, for 2D lines and points, the x and y components of the 2D vector distance field are linear and differentiable everywhere as illustrated in Figure 1. In contrast, the scalar distance field of a point is non-linear everywhere and non-differentiable at the point, and the scalar distance field of a line is non-differentiable on the line. (We also note that for points, lines, and infinitely thin planes in 3D, the x, y, and z components of the 3D vector distance field are linear and differentiable everywhere.)

3.2 Adjusting Control Vertices Using the Vector Distance Field

The fitting error, E, of each curve segment in the estimating curve is approximated as the average squared distance from a set of test points $\{Q_i = B(t_i)\}$ on the curve segment to the polyline connecting the sequence of digitized points of the input path:

$$E = \frac{1}{N}\sum_{i=1}^{N} d(Q_i)^2 ,$$

where *d(Q)* is the distance from Q to the polyline.

We constrain the endpoints of each curve segment in the estimating curve to lie on a digitized point of the input path and use an iterative algorithm to reduce the fitting error of the curve segment by adjusting its off-curve control vertices. Recall that a point B(t) on a cubic Bezier curve with endpoints C_0 and C_3 and off-curve control vertices C_1 and C_2 can be expressed as a cubic polynomial of parameter t:

$$B(t) = C_0(1-t)^3 + 3C_1 t(1-t)^2 + C_2 t^3 (1-t) + C_3 t^3 \text{, where } t \in [0, 1]. \qquad (1)$$

In each iteration, we move off-curve control vertices in the direction that reduces the curve fitting error most quickly, i.e., in the direction of the derivative of the error with respect to the position of the off-curve control vertex. Taking this derivative for the first off-curve control vertex and using the chain rule, we determine the adjustment 'force' $\vec{f}_1$ acting on C_1 as:

$$\vec{f}_1 = \nabla_{C_1} E = \frac{1}{N}\sum_{i=1}^{N} 2d(Q_i)\cdot \nabla_{C_1} d(Q_i)$$
$$= \frac{1}{N}\sum_{i=1}^{N} 2d(B(t_i))\cdot \nabla_{C_1} d(B(t_i)).$$

Note that

$$\nabla_{C_1} d(B(t_i)) = \left(\frac{\partial d(B(t_i))}{\partial x_1}, \frac{\partial d(B(t_i))}{\partial y_1} \right) \text{ for } C_1 = (x_1, y_1)$$

$$= \left(\frac{\partial d(x(t_i))}{\partial x(t_i)} \cdot \frac{\partial x(t_i)}{\partial x_1}, \frac{\partial d(y(t_i))}{\partial y(t_i)} \cdot \frac{\partial y(t_i)}{\partial y_1} \right) \text{ for } B(t_i) = (x(t_i), y(t_i)),$$

using the chain rule.

Consequently,

$$\nabla_{C_1} d(B(t_i)) = 3t_i \cdot (1-t_i)^2 \cdot \left(\frac{\partial d(x(t_i))}{\partial x(t_i)}, \frac{\partial d(y(t_i))}{\partial y(t_i)} \right)$$
$$= 3t_i \cdot (1-t_i)^2 \cdot \nabla d(B(t_i)).$$

Thus,

$$\vec{f}_1 = \frac{1}{N}\sum_{i=1}^{N} 2d(B(t_i)) \cdot 3t_i \cdot (1-t_i)^2 \cdot \nabla d(B(t_i))$$
$$= \frac{6}{N}\sum_{i=1}^{N} t_i \cdot (1-t_i)^2 \cdot d(B(t_i)) \cdot \nabla d(B(t_i)).$$

Similarly, the adjustment 'force' $\vec{f}_2$ acting on C_2 is

$$\vec{f}_2 = \nabla_{C_2} E = \frac{1}{N}\sum_{i=1}^{N} 2d(Q_i) \cdot \nabla_{C_2} d(Q_i)$$
$$= \frac{6}{N}\sum_{i=1}^{N} t_i^{\,2} \cdot (1-t_i) \cdot d(B(t_i)) \cdot \nabla d(B(t_i)).$$

To reduce the fitting error, we iteratively adjust the positions of the off-curve control vertices C_1 and C_2 by adding vectors proportional to the adjustment forces $\vec{f}_1$ and $\vec{f}_2$:

$$C_1^{j+1} = C_1^{j} + \alpha \vec{f}_1$$
$$C_2^{j+1} = C_2^{j} + \alpha \vec{f}_2 \qquad (2)$$

In general, the proportionality constant $\alpha \in [0,1]$ determines the stability and convergence properties of the curve fitting algorithm, with a smaller α providing more stability but slower convergence. However, because components of the vector distance field are nearly linear close to the polyline, our approach is reasonably insensitive to α; an α value of 1 provides both fast convergence and good stability.

Both $\vec{f}_1$ and $\vec{f}_2$ are functions of the Euclidean distance to the polyline d(Q) and the gradient of the Euclidean distance field *$\nabla d(Q)$* at the set of test points $\{Q_i = P(t_i)\}$ on the curve segment. In particular, *$d(Q) \cdot \nabla d(Q) = \| (dx, dy) \| \cdot (dx, dy)$*, where *(dx, dy)* is the vector distance at point Q. Thus, the adjustments $\vec{f}_1$ and $\vec{f}_2$ can be computed directly by interpolating the vector distance map of the polyline at the test points $\{Q_i\}$.

3.3 Maintaining G^1 Continuity of the Estimating Curve

The smoothness of an analytic curve can be described in terms of its *continuity*, a function of its differentiability at points along the curve (see [2]). Because polynomial curves are everywhere infinitely differentiable, the continuity of a piecewise polynomial curve is determined by the continuity at the joints between curve segments: C^0 continuity simply implies that curve segments are connected at their endpoints; C^1 continuity implies that the tangent vectors of connected curve segments are parallel and have equal length at the point where they are joined; C^2 continuity implies that the curvature of connected curve segments is equal at the point where they are joined, and so on. Geometric continuity, G^N, is somewhat less restrictive than algebraic continuity, C^N. In particular, two curve sgments are G^1 continuous if their tangent vectors are parallel but not necessarily equal in length at the point where they are joined. Note that G^1 continuous curve segments are smooth enough for most applications but the proposed method can be extended to achieve higher order algebraic or geometric continuity when higher order Bezier curves are used in the piecewise polynomial curve.

The tangent vector of a cubic Bezier curve (see eqn. (1)) is $\vec{t}(t)$ = (dB(t)/dx, dB(t)/dy) = $3(C_1 – C_0)(1 – t)^2 + 6(C_2 – C_1)\, t\, (1 – t) + 3(C_3 – C_2)\, t^2$. At the curve's first and last endpoints, $\vec{t}(0) = 3(C_1 – C_0)$ and $\vec{t}(1) = 3(C_2 – C_3)$, respectively. These endpoint tangent vectors lie on the lines connecting the first endpoint to the first off-curve control vertex and the second endpoint to the second off-curve control vertex, respectively. Hence, in order to maintain G^1 continuity at non-corner points between a new curve segment and its preceding curve segment, the first off-curve control vertex of the new curve segment is restricted to lie on the line **L** passing through the second off-curve control vertex and the second endpoint of the preceding curve segment. Because the first off-curve control vertex of the new curve segment originates on **L** (it is initialized to lie at the endpoint common to the new curve segment and the preceding curve segment), the first off-curve control point can be constrained to lie on **L** by restricting the displacement $\vec{f}_1$ of the first control vertex to be parallel to **L**. Thus, in order to maintain G^1 continuity at the first endpoint of the new curve segment, we replace $\vec{f}_1$ with the restricted displacement $\vec{f}_1^* = \left(\vec{f}_1 \circ \vec{l}\right) \cdot \vec{l}$, where $\vec{l}$ is the unit direction vector of **L** and '$\circ$' is the vector dot product, and adjust the first off-curve control vertex of the new curve segment in equation (2) using $C_1^{j+1} = C_1^j + \alpha \vec{f}_1^*$.

3.4 Efficient Incremental Computation of the Vector Distance Field

Computing the control vertex adjustments required for on-the-fly curve fitting described in sections 3.2 and 3.3 requires vector distances to the input path at test points along the estimating curve. Computing these vector distances using a brute force approach (i.e., by first computing the vector distance to each line segment in the polyline representing the input path and then choosing the vector distance with the minimum magnitude) is prohibitively slow for complex input paths. While geometric data structures could be used to improve the brute force approach [9], better results are achieved by representing the polyline with a vector distance map that is incrementally updated as each new digitized point is acquired. Vector distances at test points can then be efficiently interpolated from sampled distances in the vector distance map, e.g., by using bilinear interpolation. Given the vector distance map, our curve fitting approach is simple, fast, and robust. Thus, achieving on-the-fly curve fitting during drawing requires an efficient method for computing the vector distance map. Fortunately, we can take advantage of the linear nature of the vector distance field near points and lines and a fast distance-based rendering method described in [11] to achieve the desired speed.

In our prototype implementation, we store the vector distance map in a 2D image with the same dimensions and resolution as the display window used by the drawing application. We store two 32-bit floating point values per pixel, i.e., dx and dy, but we have verified that using two 8-bit values reduces memory requirements without compromising accuracy. If required, using an adaptively sampled vector distance field representation could provide additional compression and/or reduced processing loads (e.g., see [5]). The vector distance field of the polyline is a CSG union of the vector distance fields of the polyline's individual line segments, where the CSG union of two vector distances chooses the vector distance with the smaller magnitude. Thus the vector distance map of the polyline can be constructed incrementally; when a new digitized point becomes available, the vector distance field of the line segment from the end of the existing polyline to the new digitized point is simply added to the existing vector distance map using a CSG union operation.

The vector distance field of a line segment is composed of the field closest to the line segment itself and the field closest to the digitized points defining its endpoints. In practice we only need to compute the vector distance field of one of the two endpoints for each line segment because endpoints are shared along the polyline. Computing the vector distance field for each line segment can be made very efficient for the following two reasons.

First, using the proposed incremental approach, the estimating curve is never very far from the polyline. Hence, the vector distance field is only required within a limited radius, *R*, from the polyline, where *R* is determined by the maximum allowable curve error and the spacing between input points. Consequently, we can define a limited region enclosing each line segment within which we need to compute its vector distance field. Because contributions from each line segment are added using the CSG union operator, the limited regions can overlap, allowing the use of regions with simple geometry. In our case, we choose the quadrilateral regions illustrated in Figure 2 – for an endpoint, we use an axis-aligned square centered on the endpoint with sides of length *2R* and, for a line segment, we use a rectangle centered along the line segment with width *2R*.

Second, as discussed in section 3.1, the x and y components of the vector distance fields of points and lines are linear. Thus, to add the vector distance field of a new line segment to the existing vector distance field, we rasterize two simple geometric shapes: a square limiting the vector distance field of the line segment's first endpoint, and a rectangle limiting the vector distance field of the line segment itself. During rasterization,

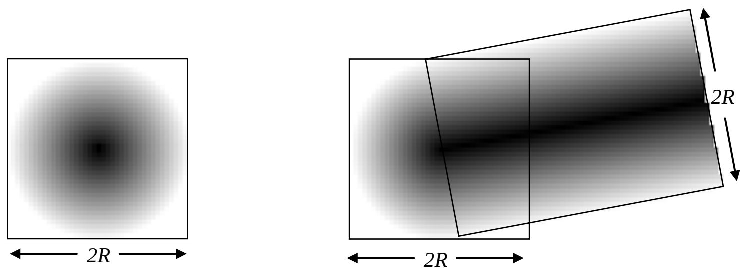

Figure 2. Left: the magnitude of the vector distance field of a point limited by an axis-aligned square region with sides of length *2R*, where *R* is the maximum expected deviation of the estimating curve from the polyline during curve fitting. Here, black indicates zero distance and black to white indicates an increasing distance from the point. Right: the magnitude of the vector distance field of a line segment and its first endpoint limited by their appropriate quadrilateral regions. Note that the computed distance field of the line segment is limited by a rectangle with its centerline along the line segment and a width of *2R*. The two limiting regions can overlap; their contributions are added to the vector distance map using a CSG union that chooses the vector distance with the minimum magnitude.

the x and y components of the vector distance fields are computed at corners of their respective quadrilateral regions and linearly interpolated (e.g., using a digital differential analyzer [4]) across the quadrilateral regions. The CSG union operator compares the magnitude of the interpolated vector distance at a particular raster location to the magnitude of the corresponding vector distance already stored in the vector distance map and chooses the vector distance with the smaller magnitude. In practice, we compare squared magnitudes to avoid taking square roots.

4. Performance

Our prototype implementation of the proposed curve fitting method is fast enough that curve segments of a piecewise polynomial curve can be fit to a sequence of digitized input points on-the-fly, i.e., as they are made available by the application. When drawing with a mouse or digitizing pen, there is no perceptible delay between the motion of the input device and the drawing of the piecewise polynomial curve. The code was written in C with reasonable care but was not optimized for speed. Although the speed of the proposed method is not an issue for desktop computers, it may be more important for processor constrained systems. In such cases, there are several optimizations possible. First, in our experience it is likely that optimized code would achieve a 2x speed improvement over the prototype implementation. Second, there are a number of algorithmic changes that would result in speed enhancements. For example, the prototype implementation currently clears the entire vector distance map each time a new curve segment is initialized. If the algorithm kept track of the minimum and maximum extent into which the distance fields were drawn, we could significantly decrease the area of the display window that needed to be cleared. A second algorithmic enhancement would

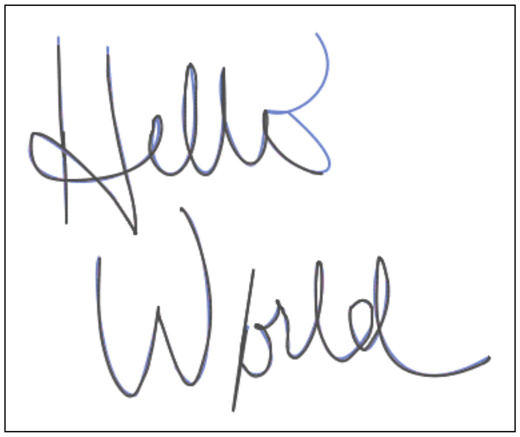

Figure 3. A comparison between text written with the proposed method (light blue curves) and text written with a standard existing curve fitting method [10] (black curves). The proposed method matched the input path within the user specified error tolerance in all observed cases. The two typical failure modes of the standard method are illustrated in this example: sections of the input path that are not fit at all (e.g., the 'o' in 'Hello') and 'wild' curves where the fit curve differs significantly from the input path (e.g., the line through the 'o' in 'World').

adjust the number of test points along each curve segment according to the length of the curve segment (currently the number of test points is fixed). Finally, graphics hardware could be used to rasterize the quadrilateral regions limiting the distance fields contributed by each new line segment. By storing the x and y components of the vector distance field as color components of quadrilateral vertices, the vector distance field could be linearly interpolated across each quadrilateral region during rasterization. A shader function could be used to perform the CSG union in order to add the contribution from a new line segment to an existing vector distance map.

We compared our prototype implementation of the vector distance field method to an implementation of Schneider's method which we had previously adapted and optimized from code available in [13]. In general, while both methods produced curves of similar quality and accuracy, the proposed vector distance field based method proved to be significantly more robust. The only observed failure mode for the proposed method occurs if the width of the quadrilateral regions used to limit the vector distance fields of points and line segments is too small. By setting this width sufficiently large (e.g., we used 20 pixels for the accuracy and timing tests reported here), the generated piecewise polynomial curve fits the input path to within the user specified maximum error. In contrast, although Schneider's method behaves well for simple curves, we observed frequent failures when the input path resembled typical cursive handwriting. Schneider's method exhibited two failure modes (see Figure 3), both of which appear to be due to poor initialization of the root finder used to locate minimum distance points on the estimating curve. The frequency of these failure modes did not seem to be correlated with the user-specified maximum allowable curve error. When comparing the two methods on

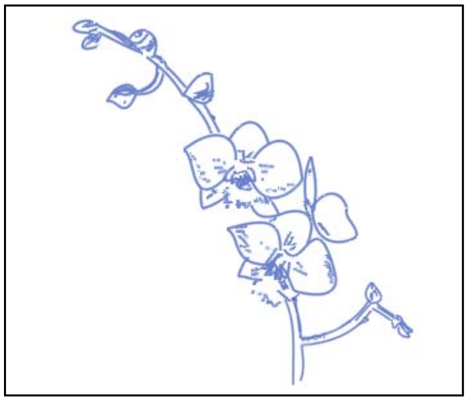
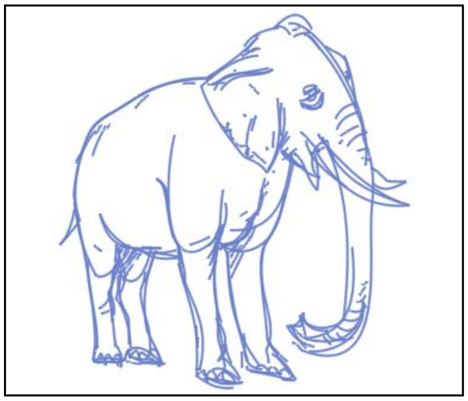

Figure 4. Illustrations drawn using the proposed curve drawing method. The method provides immediate feedback of sketched curves as they are drawn and comfortably handles both smooth curves and small detail.

54 handwritten examples of the test phrase ‘Hello World’, it was found that the proposed vector distance field method never failed while Schneider’s method failed in 16 of the 54 cases.

We also compared the two algorithms on a set of 25 randomly drawn input paths (see Table 1). The 19 input paths that were fit without failure by Schneider’s method had an average of 89.4 digitized input points (with a range of 41 to 194 points). The generation times reported for Schneider’s method represents the time required to apply his algorithm only *after all of the digitized points were recorded*. The time reported for the proposed method represents *the total fitting time during drawing*, i.e., it includes the time taken to refit the curve each time a new point is acquired. These metrics are clearly different – they favor Schneider’s approach which was not designed to be applied on-the-fly and total times would be significantly longer if it were reapplied with each new input point – but the comparison confirms the speed of the proposed method. Finally, although the maximum allowable error for both methods was set to approximately the same value (i.e., 1 pixel), the proposed method produced analytic curves with significantly fewer cubic Bezier curve segments (an average of 9.1 vs. 31.6, with ranges of 5 to 19 cubic Bezier curve segments for the proposed method vs. 8 to 77 cubic Bezier curve segments for Schneider’s method). Note that producing fewer curve segments has advantages for applications that require transmission of curve segments over limited bandwidths or for drawing applications that allow curve editing.

Method	Avg. #points	Avg. total time	Avg. time per point	Avg. #curve segs
Schneider	89.4	0.334 sec.	0.00366 sec.	31.6
VDFs	89.4	0.261 sec.	0.00303 sec.	9.11

Table 1. A comparison of Schneider’s method and the proposed method for 19 randomly drawn input paths.

5. Discussion

The proposed method for fitting a piecewise parametric curve to a sequence of digitized points is robust, relatively simple, and fast enough to fit curves on-the-fly during drawing. However, the method is not ideal under all circumstances. First, the approach uses a local optimization; if achieving the best possible fit, the smallest number of curve segments, or other global optimizations are more important than on-the-fly curve fitting, the proposed method may not be optimal (although use of the vector distance field during curve fitting would still be advantageous). Second, the proposed method approximates the input path by a polyline. If the digitized points are too far apart, the polyline, and hence the fit curve, may not represent the intended input path very well. Third, because the vector distance field becomes complex when the input path crosses itself, the proposed method tends to split curve segments to avoid self-intersections thereby resulting in more curve segments than may be absolutely necessary.

There are several ways that the proposed method could be modified or extended. For example, if jittery input data would benefit from smoothing (as described in [12]), a windowed smoothing filter could be applied to the digitized points before curve fitting. Although this could result in a delay during curve drawing of half the width of the smoothing filter, a predictive filter (e.g., see [8]) could be used to maintain on-the-fly curve fitting (this approach would require some backtracking at corners). Finally, in order to achieve better global curve fitting, the method could be extended to adjust both the current and the previous curve segment as each new digitized point is acquired. Finally, as suggested in [12], higher order Bezier curve segments could be used to achieve higher order continuity at the endpoints of curve segments.

References

[1] S. J. Ahn, *Least Squares Orthogonal Distance Fitting of Curves and Surfaces in Space*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, 2004.

[2] G. Farin, *Curves and Surfaces for CAGD: A Practical Guide*, Morgan Kaufmann Publishers, Academic Press, 2002.

[3] O. Faugeras and J. Gomes, "Dynamic Shapes of Arbitrary Dimension: The Vector Distance Functions", Proceedings IMA Conference on Mathematics of Surfaces, pp. 227-262, 2000.

[4] J. Foley, A. vanDam, S. Feiner, and J. Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley, 1992.

[5] S. F. Frisken, R. N. Perry, and T. R. Jones, "Detail-directed Hierarchical Distance Fields", U.S. Patent 6,396,492.

[6] S. F. Frisken and R. N. Perry, “Designing with Distance Fields”, in *Interactive Shape Editing*, ACM SIGGRAPH 2006 Course Notes, ACM Press, 2006.

[7] S. F. Frisken and R. N. Perry, “Efficient Estimation of 3D Euclidean Distance Fields from 2D Range Images”, in *Proc. Symposium on Volume Visualization and Graphics*, 2002.

[8] M. L. Honig and D. G. Messerschmitt, *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic Publishers, 1984.

[9] E. Jakubiak, S. Frisken, and R. Perry, “Proximity Cluster Trees”, submitted to J. Graphics Tools, January, 2007.

[10] T. Nishita, T. W. Sederberg, and M. Kakimoto, “Ray Tracing Trimmed Rational Surface Patches”, ACM SIGGRAPH 1990 Conference Proceedings, pp. 337-345, 1990.

[11] R. N. Perry and S. F. Frisken, “Method and Apparatus for Rendering Cell-Based Distance Fields Using Texture Mapping”, U.S. Patent 6,917,369.

[12] P. Schneider, *Phoenix: An Interactive Curve Design System Based on the Automatic Fitting of Hand-sketched Curves*, Master’s Thesis, University of Washington, 1988.

[13] P. Schneider, “An Algorithm for Automatically Fitting Digitized Curves”, in *Graphics Gems*, ed. Andrew Glassner, Academic Press, p. 408-415, code: p. 787, 1990.

Web Information: Detailed pseduocode is available online at the following web address (pseudocode is included below for review purposes only).

Pseudocode for Efficient Curve Fitting Using Vector Distance Fields

```
CreateWindow(width, height)
{
    //----Create and initialize a display window

    //----Allocate and clear a vector distance map associated with the
    //----display window for sampling and storing the vector distance field
    //----of a polyline connecting a sequence of digitized mouse points
    vectorDistMap ← CreateVectorDistMap(width, height);
}

DestroyWindow(window)
{
    //----Free the vector distance map associated with the display window
    Free(vectorDistMap);
    //----Destroy the display window
}

MouseDownCallback(x, y)
{
    //----Initialize a piecewise parametric curve
    curve ← InitParametricCurve()

    //----Initialize the first curve segment in the piecewise parametric
    //----curve
    InitCurveSegment(x, y, curve.curveSeg[0])

    //----Clear the vector distance map
    ClearVectorDistMap(vectorDistMap);
}

MouseUpCallback()
{
    //----Terminate the parametric curve
    TermParametricCurve(curve)
}

MouseMoveCallback(x, y)
{
    //----Attempt to update the current curve segment with the new
    //----digitized point (x, y)
    updateResult ← UpdateCurveSegment(x, y, currentSeg)

    //----Start a new curve if the current curve segment was not updated
    //----successfully
    if (updateResult != SUCCESS) {

        //----Terminate the previous curve segment at its previous endpoint
        //----(xPrev, yPrev)
        TermCurveSegment(currentSeg)

        //----Clear the vector distance map
        ClearVectorDistMap(vectorDistMap);

        //----Initialize a new curve segment beginning at (xPrev, yPrev)
        InitCurveSegment(xPrev, yPrev, nextSeg)

        //----Constrain the tangent vector at the first endpoint of the new
        //----curve segment to be parallel to the tangent vector at the end
        //----of the current curve segment in order to maintain G1
        //----continuity unless a corner was detected
        if (updateResult == FAILURE) nextSeg.constrained ← TRUE
        else if (updateResult == CORNER) nextSeg.constrained ← FALSE

        //----Update the new curve segment to include (x, y) and reset the
        //----current curve segment to be the new curve segment
        UpdateCurveSegment(x, y, nextSeg)
        currentSeg ← nextSeg
    }
}
```

```
InitCurveSegment(x, y, curveSegment)
{
    //----Initialize the curve segment’s 4 control vertices to (x, y)
    curveSegment.C0 ← (x, y)
    curveSegment.C1 ← (x, y)
    curveSegment.C2 ← (x, y)
    curveSegment.C3 ← (x, y)
}

UpdateCurveSegment(x, y, curveSegment)
{
    //----Test for a corner between (x, y) and (xPrev, yPrev). A corner is
    //----detected at (xPrev, yPrev) if the angle between the tangent at
    //----the end of the curve segment and the new line segment is greater
    //----than a minimum corner angle (e.g., 60 degrees).
    if (TestCorner(x, y, curveSegment) return(CORNER)

    //----Reset the positions of the 3rd and 4th control vertices. Move the
    //----4th control vertex from its previous position (xPrev, yPrev) to
    //----(x, y) and move the 3rd control vertex by the same amount (i.e.,
    //----by (x – xPrev, y – yPrev))
    (xPrev, yPrev) ← curveSegment.C3
    curveSegment.C3 ← (x, y)
    curveSegment.C2 ← curveSegment.C2 + (x, y) - (xPrev, yPrev)

    //----Update the vector distance map, adding contributions from the new
    //----line segment (i.e., the line segment from (xPrev, yPrev) to
    //----(x, y)) and the point (xPrev, yPrev). The fieldRadius limits the
    //----range from the line segment or point where the vector distance
    //----field is computed. The fieldRadius should be as small as possible
    //----to limit the number of distance computations required but large
    //----enough so that the estimating curve almost always lies within
    //----this range of the polyline.
    RenderLineCell(xPrev, yPrev, x, y, fieldRadius, vectorDistMap)
    RenderPointCell(xPrev, yPrev, fieldRadius, vectorDistMap)

    //----Adjust the positions of the off-curve control vertices, C1 and
    //----C2. Iterate until the curve error is below threshold or a maximum
    //----number of iterations has been performed
    while ((error > maxError) AND (nIteration < maxNIteration) {

       //----Determine the force vectors f_1 and f_2 acting on off-curve
       //----control vertices C1 and C2, respectively. Initialize both
       //----forces to the zero vector and then add force contributions
       //----from each sample point B(t_i) along the cubic Bezier curve
       //----segment, where t_i = i / N, i = 1, 2, … N, and N is the number
       //----of sample points.
       f_1 ← (0, 0)
       f_2 ← (0, 0)
       for (i = 0, i < N, i++) {

          //----Compute the vector distance (dx, dy) and the Euclidean
          //----distance, d = sqrt(dx^2 + dy^2), from the polyline at the
          //----sample point by interpolating the vector distance map
```

```
            (x, y) ← B($t_i$)
            (dx, dy) ← InterpVectorDist(vectorDistMap, x, y)
            d ← sqrt($dx^2$ + $dy^2$)

            //----Determine the force contributions to each force vector at
            //----this sample point and add them to $f_1$ and $f_2$
            $f_1(t_i)$ ← 6 * $t_i$ * $(1 – t_i)^2$ * d * (dx, dy)
            $f_2(t_i)$ ← 6 * $t_i^2$ * $(1 – t_i)$ * d * (dx, dy)
        }

        //----Constrain $f_1$ to be parallel to (xTan, yTan) if the tangent at
        //----the first endpoint is constrained
        if (curveSegment.constrained == TRUE) {
            $f_1(t_i)$ ← dotProduct((xTan, yTan), $f_1(t_i)$) * (xTan, yTan)
        }

        //----Move each of the off curve control vertices using the
        //----force vectors
        curveSegment.C1 ← curveSegment.C1 + $f_1$
        curveSegment.C2 ← curveSegment.C2 + $f_2$

        //----Compute the curve error, i.e., the sum of the squared
        //----Euclidean distances from sample points along the curve to the
        //----polyline. Initialize the error to zero and then add the
        //----contribution to the error from each sample point B($t_i$) along
        //----the cubic Bezier curve segment, where $t_i$ = i / N, i = 1, … N,
        //----and N is the number of sample points
        error ← 0
        for (i = 0, i < N, i++) {
            (x, y) ← B($t_i$)
            (dx, dy) ← InterpVectorDist(vectorDistMap, x, y)
            error ← error + $dx^2$ + $dy^2$
        }

        //----Test for a successful curve fitting
        if (error < maxError) return(SUCCESS)
    }

    //----The curve fitting failed. Reset the curve’s control vertices to
    //----their original values and return FAILURE.
    ResetControlVertices(curveSegment)
    return(FAILURE)
}


RenderLineCell($x_0$, $y_0$, $x_1$, $y_1$, fieldRadius, vectorDistMap)
{
    //----Determine the edges of a rectangular line cell centered on the
    //----line segment from ($x_0$, $y_0$) to ($x_1$, $y_1$) with width twice
    //----fieldRadius

    //----Determine the vector distance at the bottom-left vertex of the
    //----line cell (i.e., the vector distance from the vertex to the line
    //----line segment)
    //----Determine the change in the vector distance with unit changes in
    //----x and y

    //----Rasterize the line cell (e.g., using an active edge list
    //----algorithm), making use of the fact that the x and y components of
    //----the vector distance are linear inside the line cell and can be
    //----computed incrementally using a digital differential analyzer
    //----(see [4])
}

RenderPointCell(x, y, fieldRadius, vectorDistMap)
{
    //----Determine the edges of a square, axis-aligned point cell centered
    //---- on (x, y) with side lengths twice fieldRadius (see Figure 2)

    //----Determine the vector distance at the bottom-left vertex of the
    //----line cell (i.e., the vector distance from the vertex to (x, y)

    //----Determine the change in the vector distance with unit changes in
    //----x and y

    //----Rasterize the point cell using a digital differential analyzer
}
```