Integer Forward Differencing of Cubic Polynomials: Analysis and Algorithms

R. VICTOR KLASSEN
Xerox Webster Research Center

Two incremental cubic interpolation algorithms are derived and analysed. Each is based on a known linear interpolation algorithm and modified for third order forward differencing. The tradeoff between overflow avoidance and loss of precision has made forward differencing a method which, although known to be fast, can be difficult to implement. It is shown that there is one particular family of curves which represents the worst case, in the sense that if a member of this family can be accurately drawn without overflow, then any curve which fits in the bounding box of that curve can be. From this the limitations in terms of step count and screen resolution are found for each of the two algorithms.

Categories and Subject Descriptors: G.1.1 [**Numerical Analysis**]: Interpolation—*spline and piecewise polynomial interpolation*; G.1.2 [**Numerical Analysis**]: Approximation—*spline and piecewise polynomial approximation*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*curve, surface, solid, and object representations, geometric algorithms, languages and systems.*

General Terms: Algorithms

Additional Key Words and Phrases: Bézier curves, parametric curve plotting.

I. INTRODUCTION

Nearly every first course in computer graphics includes the simple DDA algorithm for scan conversion of lines [10]. This well-known technique is fundamental to much of raster graphics, and generalizes to the forward differencing method by which parametric spline curves may be drawn. These methods are normally taught as floating point methods, requiring rounding to integers at each pixel. (In the first edition of *Principles of Interactive Computer Graphics* [9], a fixed point method for symmetric DDA is sketched, but it was dropped for the second edition.) Bresenham's algorithm for lines uses only integer arithmetic [3], but in its usual presentation it requires unit

This work was begun while the author was being supported by the Natural Sciences and Engineering Research Council, Digital Equipment Canada, and the University of Waterloo Institute for Computer Research.
Author's Address: Xerox Corporation, Webster Research Center, 800 Phillips Rd., Webster, New York 14580.

step size and generates lines in octants other than the first by reflections [6, 10, 11].

Field has provided two algorithms for incremental linear interpolation [5]. One of these uses fixed point (Algorithm A), while the other (Algorithm B) uses a generalization of Bresenham's algorithm. This generalization allows arbitrary step size and places no restrictions on the octant containing the line. Their relative speeds depend on the processor. Algorithm B is perfectly accurate, although Algorithm A is sufficiently accurate for most computer graphics applications. Both algorithms require integral endpoints, are reversible (producing the same interpolants regardless of direction), and produce interpolants at a number of uniform steps independent of the separation of the endpoints (in Bresenham's algorithm the number of steps is given by the distance in one dimension between the endpoints). PThe algorithms presented here are generalizations of those provided by Field, designed to handle low degree polynomials. The first new algorithm is a generalization of Field's Algorithm A, motivated by the approach of Bartels et al. [1, pp. 400–406]. The degree of the polynomial dictates a limit on the number of interpolants that may be safely obtained: errors accumulate more rapidly in higher degree polynomials. The bound for cubics allows sufficiently many interpolants for most purposes, but care must be exercised to avoid exceeding those limits. The second algorithm is a generalization of Field's Algorithm B. Here the degree of the polynomial also limits the number of steps that may be taken, but the limitation is caused not by accumulated error, but by overflow in the calculation of the initial values.

The algorithms presented in this paper do not require the use of extended precision. In order to guarantee that overflow and lost precision are prevented, some constraints are needed on the curves. These take the form of limitations on the number of steps and limitations on the screen resolution, and hence the size of the coordinates of the points on the curve. If the guarantee holds only for very short curves, the efficiency of forward differencing does not pay for the initialization overhead. If the constraints are met for all curves of moderate length, then longer curves must be split prior to rendering, and the length for which a curve must be split is needed. For each algorithm derived below, the limitations on the step count and screen resolution are given, as functions of the word size. In addition, it is shown that on a screen of height (or width) $2h$, no more than $18h$ steps of equal parametric separation are needed to render a parametric cubic that entirely fits within the screen, taking no step greater than one pixel in x or y.

The remainder of this paper consists of two parts. The first part contains useful information about Bézier curves and forward differencing of cubics, which is independent of implementation. There are three sections in the first part. The first section presents known results about Bézier curves, as background for the second and third. The second section presents the assumptions used in defining the problem. In the third section, the properties in the first section, along with the assumptions of the second are used to find the "worst-case" family of curves. This curve family is the most likely to cause overflow either in the initialization or while either forward differencing

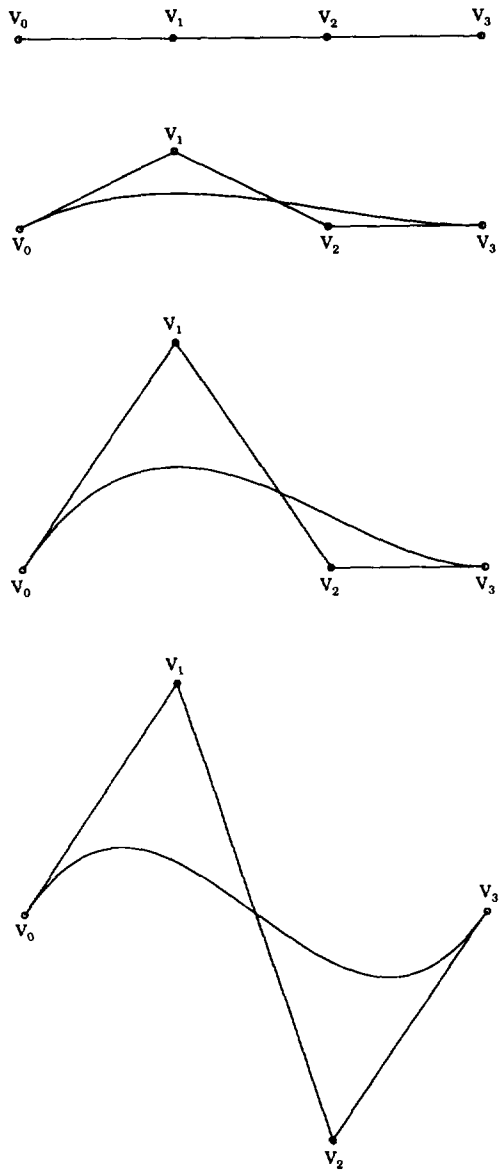


Fig. 1. Behavior of a Bézier curve. (a) With four colinear control points all four are interpolated. (b) As v_1 is raised, the curve is raised, except at the endpoints, which continue to be interpolated. (c) The further v_1 is raised, the further it is from the curve. (d) As v_2 is lowered, the highest point on the curve is lowered as well, so the curve becomes further from v_1.

algorithm is being used to step through the curve. The second part of the paper contains the two algorithms for integer forward differencing and discussions of their limitations.

2. BÉZIER CURVES

Any cubic basis may be used to represent the curve prior to initialization for forward differencing. The Bézier representation has some properties that are

useful for finding the curve most likely to cause overflow. This section introduces those properties.

Figure 1 shows a series of Bézier curves along with their control polygons. In Figure 1a, the control vertices are colinear and consequently the curve interpolates all four. As v_1 is raised, (b and c), the curve moves upward, except at the end control points, which are always interpolated. Also, as v_1 is raised, the curve moves further from v_2. Finally, as v_2 is lowered, the curve is pulled further away from v_1.

If the x coordinates of the control points are equally separated in the interval $[0, 1]$, $x(t) = t$. This is an easy way of plotting $y(t)$ as $y(x)$ for clarity. Because the two coordinates of points on a parametric curve are calculated independently, numerical properties of algorithms for computing them apply to each dimension independently. In the remainder of this paper, the curves discussed will be reduced to one-dimensional cases, by making $y(t) = y(x)$ as above.

The vector-valued derivative of a curve is known as its *hodograph*. The hodograph of a Bézier curve is given by a Bézier curve of the next lower order. The "control points" of the hodograph are proportional to the difference vectors between adjacent control points of the original curve. In particular, the derivative of a Bézier cubic curve is the quadratic curve specified by Bézier control points that are three times the differences of the control points of the original curve. The figures on the following pages show many examples of one-dimensional parametric curves and their derivatives as functions of t. (Strictly speaking, the x component of the hodograph is constant; because of the one-dimensional nature of the derivative in a functional curve, the term hodograph is used loosely in this paper to mean a plot of the derivative as a function of t. For parametric curves this is a vector-valued parametric curve; for functional curves, it is a scalar functional curve.)

3. ASSUMPTIONS

In computer graphics, it is customary to consider parametric cubic curves with the parameter restricted to the interval $[0, 1)$. Bézier curves are known to be more numerically stable within this interval than elsewhere [13].

Since the curve is parameterized in the $[0, 1)$ interval, the parametric step size is assumed to be no greater than $\frac{1}{2}$. Any curve that is to be drawn on an actual display must never have an x or y coordinate outside the display. Curves that cross the boundaries of the display may be split into multiple curves, having new coefficients, but again parameterized over the interval $[0, 1)$.

Requiring that the curve be completely on screen is not a major limitation. Once the entire curve is clipped, none of the individual pixels on the curve need clipping. If clipping is done in software, clipping the entire curve is cheaper than clipping the pixels generated. If hardware pixel clipping is available at no performance cost, there are trade-offs between the cost of generating a few points off screen and that of clipping the curve to eliminate such points. If the screen size and curve step count are at the limits of the

machine word (as derived in Sections 7 and 8), leaving clipping to the hardware is not an option.

Lastly, the control points are assumed to lie on an integer grid. The grid need not be the same resolution as the display grid, but the points on the curve generated by the algorithms are rounded to this grid. For each algorithm it is possible to relax this restriction; this is discussed near the end of the paper.

4. FORWARD DIFFERENCING IN GENERAL

Forward differencing was known to Newton, who used it for polynomial interpolation of functions. Newton's method begins with calculating a number of equally spaced points at the start of the interval to be interpolated, and using those values in the initialization of a set of difference values. This is followed by a loop in which successive values are obtained by adding the differences to each other and to the result. The initialization appropriate for polynomials is slightly different from Newton's: it is discussed later in this section. Calling the differences $\delta_0 \cdots \delta_3$, the inner loop for cubic interpolation is as follows:

```
for i ← 0 to n
    output δ0
    δ0 ← δ0 + δ1
    δ1 ← δ1 + δ2
    δ2 ← δ2 + δ3
```

Using exact arithmetic, this procedure interpolates a cubic polynomial passing through $n + 1$ equally spaced points.

For two reasons, it is helpful to have a closed form expression for δ_j as a function of the parametric step size, $\Delta = 1/n$, and the parameter, t. It is needed for the analysis of overflow prevention, and it provides an easy way to derive the initialization of the δs for arbitrary initial representations of the cubic. By inspection of the above loop,

$$\delta_j(t_{i+1}, \Delta) = \delta_j(t_i, \Delta) + \delta_{j+1}(t_i, \Delta),$$

for $0 \le i \le n$, with $\delta_j \equiv 0$ for $j \ge 4$. Noting that $t_i = i\Delta$, the values assigned to δ_j sample the continuous function

$$\delta_j(t + \Delta, \Delta) = \delta_j(t, \Delta) + \delta_{j+1}(t, \Delta).$$

Thus, $\delta_{j+1}(t, \Delta) = \delta_j(t + \Delta, \Delta) - \delta_j(t, \Delta)$. This, along with the fact that δ_0 samples the function being interpolated, gives

$$\begin{aligned} \delta_0(t) &= s(t) \\ \delta_1(t, \Delta) &= s(t + \Delta) - s(t) \\ \delta_2(t, \Delta) &= \delta_1(t + \Delta, \Delta) - \delta_1(t, \Delta) \\ \delta_3(t, \Delta) &= \delta_2(t + \Delta, \Delta) - \delta_2(t, \Delta) \end{aligned} \tag{1}$$

where $s(t) = [x(t), y(t)]$ is the curve being interpolated. If these expressions are expanded with $t = 0$, the initialization for $s(t)$ results. In the remainder

of this paper, $\delta_j(t, \Delta)$ is often written as $\delta_j(t)$, or even δ_j, with t and Δ being assumed.

High-order forward differencing is known to be prone to accumulation of round-off error. If the differences are initially calculated using floating point, they may have an error associated with them that is as large as the unit round-off error, ϵ. Even if the additions in the loop are exact, at each step any error in the value of one difference is added to the error in the next lower difference.

Suppose the initial errors corresponding to δ_i are $e_i(0) = \epsilon$ for $i = 0..3$. The error in each difference may be defined recursively as a sum of its initial error and the progressive error in the next higher term:

$$e_i(n) = \epsilon + \sum_{j=0}^{n-1} e_{i+1}(j).$$

Thus,

$$\begin{aligned} e_3(1) &= \epsilon \\ e_2(n) &= (n+1)\epsilon \\ e_1(n) &= (2 + n + n^2)\epsilon/2 \\ e_0(n) &= (6 + 5n + n^3)\epsilon/6. \end{aligned}$$

Shantz and Chang, recognizing that the cubic term is dominant, and that only the initial error of δ_3 contributes to the cubic term, chose to ignore the error contributed by other initial errors [14]. This reasoning leads to the formula

$$e_i = \epsilon_3 \frac{n!}{i!(n-i)!},$$

where $\epsilon_3 = e_3(0)$ is the initial error in the third difference. Omission of the initial errors in the other terms leads to an underestimate of the final error by

$$\epsilon\left(1 + \frac{n + n^2}{2}\right),$$

assuming the initial errors in all three terms are ϵ. This discrepancy is large enough that, in some instances, unanticipated single pixel errors can occur.

The effect of accumulated errors can be avoided by retaining sufficient fractional precision to prevent them from appearing in the result. For $n = 1024$, 25 fractional bits are required in the high order differences. This leaves only 7 bits to specify the integer position in a 32-bit (integer) word. Because 32-bit floating point typically has 24 or fewer bits of mantissa, floating point implementations must use double precision. Because the magnitudes of the differences are not in the same range, fewer integer bits are required for the higher order differences. This strategy is used in the algorithms presented here. Floating point automatically keeps fewer integer bits if the higher order differences are smaller, but it allows insufficient control to provide a tight guarantee of accuracy.

5. A "WORST-CASE" CURVE

Of all possible parametric cubic curves $s(t)$ parameterized on the $[0, 1)$ interval that fit entirely on the screen, one family of curves is distinguished by the fact that no curve not in this family causes overflow unless this one does. That family of curves is identified in this section.

Avoidance of overflow dictates maximum magnitudes of the differences (δ_i), their computer representations (including fractional bits), and any intermediate values used in their initialization. Sections 7 and 8 deal with avoidance of overflow in the initialization. In this section the potential sizes of the terms involved are derived. As shown in Section 8, the number of fractional bits grows as three times the log of the step count. Keeping all computed values within single machine words provides a major speed improvement over double-word representations. To do so requires a decrease in the number of bits devoted to the integer part as the number of fractional bits is increased, leading to the possibility of overflow. Avoidance of overflow therefore dictates a limit on the number of steps that may be taken. Overflow is not a problem for small numbers of steps: few bits are needed for the representation of fractions. Thus the worst-case curve has the greatest potential for overflow in the limit of high step counts.

It is sufficient to consider one dimension alone. Since the differences in x and y are independent of each other, any discussion of the errors or overflow that might occur when they are calculated may be considered in only one dimension. When a restriction such as not stepping more than one pixel is applied, it should be taken with the usual meaning: the maximum of the changes in x and y should not exceed one. Any bounds on the number of steps required to meet such a restriction may be derived independently for $x(t)$ and $y(t)$.

The Bézier basis happens to be convenient for identifying the curve family first to cause overflow. The choice of representation of a curve in this family does not matter, since it is the curve itself that determines the values of the differences. If the screen has its centre at 0, and limits at $\pm h$, this curve family is specified by the Bézier control points $[-h, 5h, -5h, h]$, in this order or reversed, in one dimension, and any set of points that leaves the curve on the screen in the other dimension. A curve with these control points in y is shown in Figure 11. The power basis coefficients for a Bézier curve on the control points $[v_0, v_1, v_2, v_3]$ are given by

$$\begin{aligned} c_0 &= v_0 \\ c_1 &= 3(v_1 - v_0) \\ c_2 &= 3((v_2 - v_1) - (v_1 - v_0)) \\ c_3 &= 3(v_1 - v_2) + (v_3 - v_0) \end{aligned} \tag{2}$$

so this Bézier curve is given by

$$-h + 18ht - 48ht^2 + 32ht^3, \qquad 0 \le t < 1.$$

To prove that this is the worst-case curve, it is sufficient to show that it maximizes the initial values of the differences, and any intermediate values used in computing them, and that no curve has greater values of its differences than this one, at any value of t.

For a general cubic defined in the power basis, the values of the differences from (1) are

$$\begin{aligned}\delta_0 &= c_0 + c_1 t + c_2 t^2 + c_3 t^3 \\ \delta_1 &= c_1\Delta + 2c_2 t\Delta + c_2\Delta^2 + 3c_3 t^2\Delta + 3c_3 t\Delta^2 + c_3\Delta^3 \\ \delta_2 &= 2c_2\Delta^2 + 6c_3 t\Delta^2 + 6c_3\Delta^3 \\ \delta_3 &= 6c_3\Delta^3.\end{aligned} \tag{3}$$

Each of the differences is considered in turn, beginning with the first (δ_1).

For small values of Δ, the first difference is closely related to the derivative. In the limit of small Δ, $\delta_1 \to \Delta\, ds/dt$. This follows from the mean value theorem of calculus. Since $s(t)$ is continuous over any interval $(t, t + \Delta)$ for which $\delta_1(t)$ is evaluated in the algorithm, there exists $\xi \in (0, \Delta)$ such that

$$ds(t + \xi)/dt = \frac{s(t + \Delta) - s(t)}{\Delta} = \delta_1(t, \Delta)/\Delta.$$

As $\Delta \to 0$, the interval containing ξ is correspondingly reduced.

Let the maximum value of the derivative over all possible curves that fit on the screen be $D_{\max}$. No curve s can then have

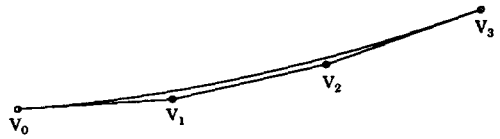
$$\delta_1(t, \Delta) > \Delta D_{\max}, 0 \le t < 1 - \Delta$$

and still fit on the screen. If it did, there would be a point $t + \xi$, $0 < \xi < \Delta$, for which $ds(t + \xi)/dt > D_{\max}$. Since the point $t + \xi$ is in the interval $[0, 1)$, $D_{\max}$ would not be the maximum possible value the derivative can take in the $[0, 1)$ interval. Thus, $\delta_1(t, \Delta) \le \Delta D_{\max}$.

Consider the first derivative. The curve which has the largest first derivative also has the largest first difference, at least in the limit of small Δ, and $D_{\max}$ is a good upper bound on the first difference. If the Bézier control points of the curve $s(t)$ are $[v_0, v_1, v_2, v_3]$, the derivative $\dot{s}(t)$ is specified by the control polygon P with control points $[D_0, D_1, D_2]$ with $D_i = 3(v_{i+1} - v_i)$. The configurations of $\dot{s}$ can be broken into two cases.

(1) P, and hence $\dot{s}$ are monotonic, that is, $D_0 \le D_1 \le D_2$ or $D_2 \le D_1 \le D_0$. By symmetry, we need consider only one of these two subcases, so we assume they are nondecreasing. Figure 2 shows a possible curve in this class, along with its hodograph.

(2) P is nonmonotonic, that is, $D_0 \le D_1$ and $D_2 \le D_1$, or $D_0 \ge D_1$ and $D_2 \ge D_1$. For quadratic Bézier curves, a nonmonotonic control polygon implies a nonmonotonic curve, so $\dot{s}$ is nonmonotonic. By symmetry, only one of these subcases needs consideration, so we consider only the first, and further assume $D_0 \le D_2$. Figure 3 shows a possible curve in this class.



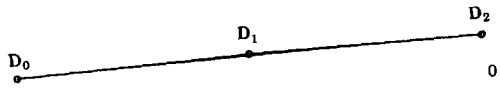


Fig. 2. A curve with monotonic derivative control graph.

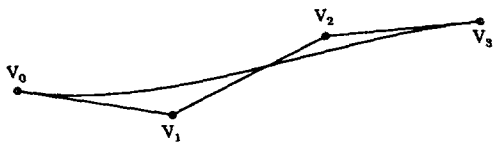
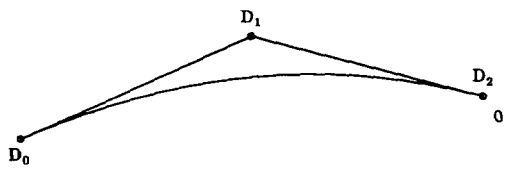


Fig. 3. A curve with nonmonotonic derivative control graph.



Each of the cases is broken into several subcases, which are discussed in turn.

(1) *Monotonic Derivative.* There are two subcases.

(i) *Monotonic curve.* In this case assume that $D_i \geq 0$. The largest slope occurs at the end of the curve segment, where it has the value D_2. The assumption that s fits on the screen implies that $v_0 \geq -h$ and $v_3 \leq h$. To make D_2 as large as possible, set $v_0 = -h$, $v_3 = h$, $v_1 = v_2 = -h$. Figure 4 shows this curve. $\dot{s}(1) = D_2 = 6h$.

(ii) *Nonmonotonic curve.* Figure 5 shows a curve in this class, along with its hodograph. No assumption is made about which is larger: v_2 or v_1. By symmetry, we may assume that $v_3 = h$ and that the maximum slope is at $t = 1$, corresponding to D_2. Assume that the change in sign of P occurs somewhere in the interval between D_0 and D_1, so $D_1 \geq 0$ and $D_0 \leq 0$. Clearly, D_2 is largest when v_2 is as low as possible. As v_2 is lowered, the lowest point on s moves down, but the lowest point must not be lower than $-h$. Moving v_1 up raises the lowest point on s, so raising v_1 makes it possible to lower v_2 further than otherwise possible. Since D_1 is nonnegative, $v_1 \leq v_2$. Hence, $v_1 = v_2$. The two center control vertices may be lowered further if $v_0 = h$ than for any lower position of v_0. With $v_0 = v_3 = h$ and $v_1 = v_2 = -5/3h$, s comes tangent to the $y = -h$ line at $t = \frac{1}{2}$. Figure 6 shows this curve. $\dot{s}(1) = D_2 = -D_0 = -\dot{s}(0) = 8h$. (Since the change of sign of P is at D_1, the assumption that it is between D_0 and D_1 has no effect).

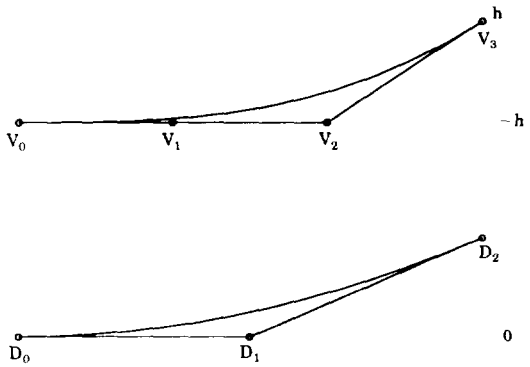
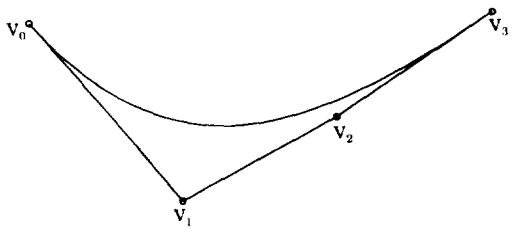


Fig. 4. The curve with monotonic control graph and monotonic derivative having maximum derivative. The derivative at $t = 1$ is $6h$.



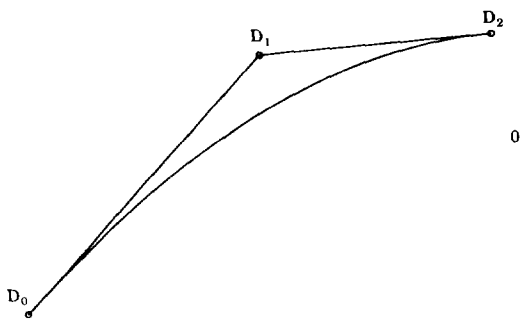


Fig. 5. A nonmonotonic curve with monotonic derivative.

(2) *Nonmonotonic derivative.* There are three subcases corresponding to the number of relative extrema in the curve segment.

(i) *Monotonic curve.* By symmetry, we can assume that $D_i \geq 0$. Figure 7 shows two possible hodographs in this class. By definition, the sum $D_0 + D_1 + D_2 = 3(v_3 - v_0)$, so if s remains on screen this sum must be no greater than $6h$. The configuration in Figure 7a is $[0, 6h, 0]$. If one of the end control vertices is raised, the center one must be lowered. Thus, in a configuration

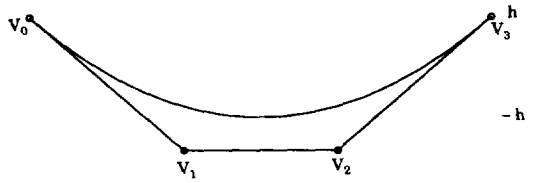
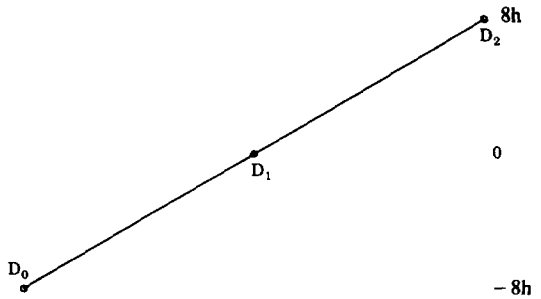


Fig. 6. The nonmonotonic curve with monotonic derivative, with maximum derivative of $\pm 8h$, at the ends.



similar to that of Figure 7a, the one shown has maximum derivative. It is $3h$, at $t = \frac{1}{2}$. In Figure 7b, the control vertices (of P) are $[8h, -4h, 2h]$. In order to raise one of the end control vertices, the other two must be adjusted, both to keep the sum no greater than $6h$ and to keep the minimum of $\dot{s}$ no less than zero (to keep the curve monotonic).

The hodograph shown in Figure 7b has the largest magnitude of curves which fall into case (2i). This can be shown by finding the maximum magnitude of the derivative for a general curve in case (2i), and then finding the curve which maximizes it. A derivative curve is given by

$$\dot{s} = D_0(1-t)^2 + 2D_1(1-t)t + D_2t^2. \tag{4}$$

Its derivative is $\ddot{s} = 2((D_1 - D_0)(1-t) + (D_2 - D_1)t)$, which is zero for

$$t = \frac{D_0 - D_1}{D_0 - 2D_1 + D_2} \equiv t_m. \tag{5}$$

For the configuration of this case, this is the parameter value at which $\dot{s}$ is maximized. Keeping the sum of the control points, and hence the maximum control point as large as possible,

$$D_0 + D_1 + D_2 = 6h,$$

or

$$D_1 = 6h - D_0 - D_2. \tag{6}$$

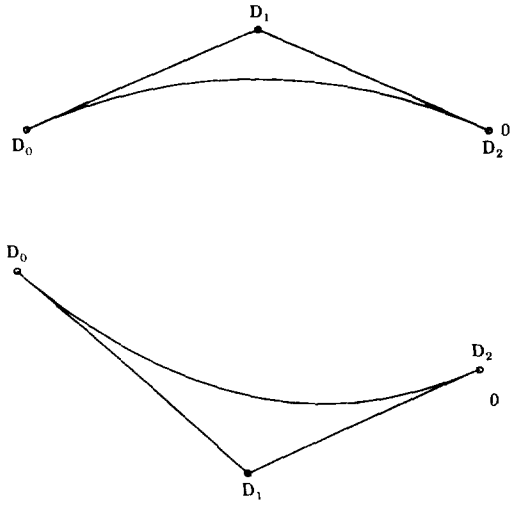


Fig. 7. Two nonmonotonic derivative curves, corresponding to monotonic curves.

The minimum value of $\dot{s}$ should be zero. Substituting (6) into (4) with $t = t_m$ from (5),

$$\frac{D_0^2 + D_0 D_2 - 12 D_0 h - 12 D_2 h + 36 h^2 + D_2^2}{D_0 + D_2 - 4h} = 0,$$

which is satisfied by points on the ellipse

$$\alpha^2 + \alpha\beta - 12\alpha - 12\beta + 36 + \beta^2 = 0,$$

with $\alpha = D_0/h$, $\beta = D_2/h$. α takes on its maximum value of 8 for $\beta = 2$, so $D_0 \leq 8h$ for this case. The curve of $\alpha = 8$, $\beta = 2$ has points $[8h, -4h, 2h]$, as required.

(ii) *One relative extremum.* The sign of $\dot{s}$ changes once. By symmetry we can assume that $D_0 \leq 0$ and $D_1 \geq D_2 \geq 0$. Figure 8 shows a curve of this class. $\dot{s}$ has its largest magnitude at either $t = 0$ or at some point in the interior with $t \geq \frac{1}{2}$. For it to be largest in the interior, $v_2 - v_1$ must be as large as possible, and thus v_2 should be as high as v_3. Raising v_2 above v_3 makes D_2 negative, which falls under case (iii), below. To make the separation between v_2 and v_1 as large as possible, v_1 must be as low as possible without s going off the bottom of the screen. v_1 may go lower if v_0 is raised as far as possible, which is to h. The control polygon is then as shown in Figure 9. s comes tangent to the $y = -h$ line at $t = 1/3$ with $v_1 = -7h/2$. By inspection of the hodograph, $\dot{s}$ has its largest magnitude at $t = 0$ where it is $-27h/2$; the relative maximum occurs at $2/3$ where it is only $9h/2$.

By construction this curve can not have a larger derivative somewhere in the interior; it remains to be shown that it can have no larger derivative at $t = 0$. Referring back to the curve in Figure 8, the way to maximize the

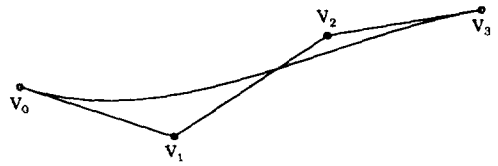
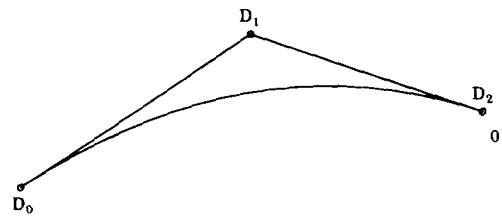
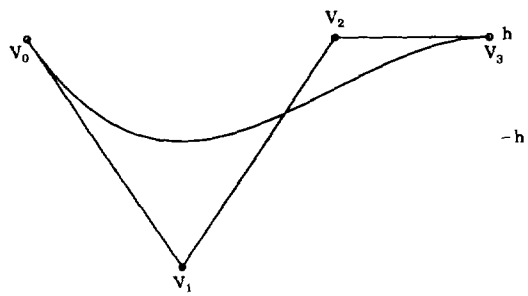


Fig. 8. A curve with one relative extremum (in this case a relative minimum).





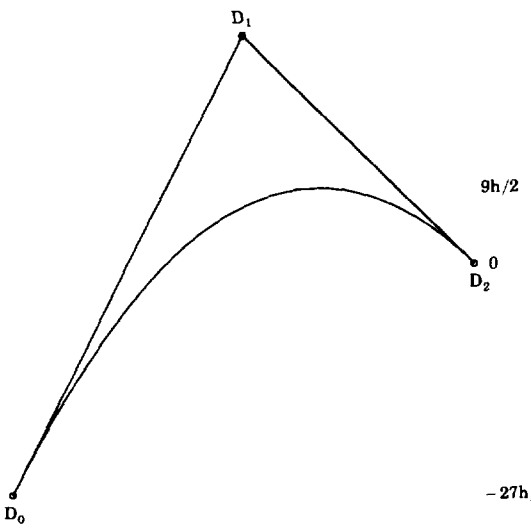


Fig. 9. The curve with one relative extremum having maximum derivative (magnitude). The derivative at $t = 0$ is $-128/9h$.

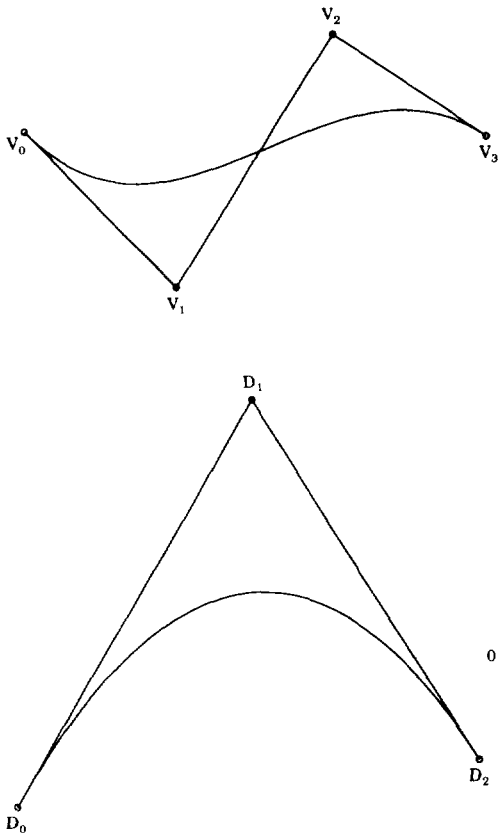


Fig. 10. A curve having two relative extrema.

magnitude of $\dot{s}(0)$ is to lower v_1 as far as possible, as was done for the curve of Figure 9.

(iii) *Two relative extrema.* Figure 10 shows this case. As for case (ii), $\dot{s}$ has its largest magnitude at either $t = 0$ or at the relative extremum of the hodograph. By symmetry, we can assume that $D_2 \geq D_0$, as in the figure, in which case the relative extremum is at $t \geq \frac{1}{2}$. The derivative's magnitude is maximized at $t = 0$ if v_0 and v_1 are as far apart as possible. Thus $v_0 = -h$ and v_1 is as high as possible. v_1 may be raised further without s going off the screen if v_2 is as low as possible. Maximizing the magnitude of $\dot{s}$ in the interior requires having v_2 and v_1 as far apart as possible. Thus as for case (ii), the magnitude of $\dot{s}$ in the interior is maximized when the magnitude of $\dot{s}(0)$ is also maximized. So far, v_0 is at $-h$, v_1 is as high as possible, and v_2 is as low as possible. Setting $v_3 = h$ makes it possible to lower v_2 further than for any lower value of v_3, and v_3 can be no higher. With $v_1 = 5h$ and $v_2 = -5h$, s is tangent to the lines $y = \pm h$ at $t = \frac{3}{4}$ and $t = \frac{1}{4}$. This curve is shown in Figure 11. The derivative at $t = 0$ is $18h$, and at $t = \frac{1}{2}$ it is $-6h$.

The cases discussed above exhaust the space of possible curves fitting the assumptions given in Section 3. For curves having a monotonic derivative,

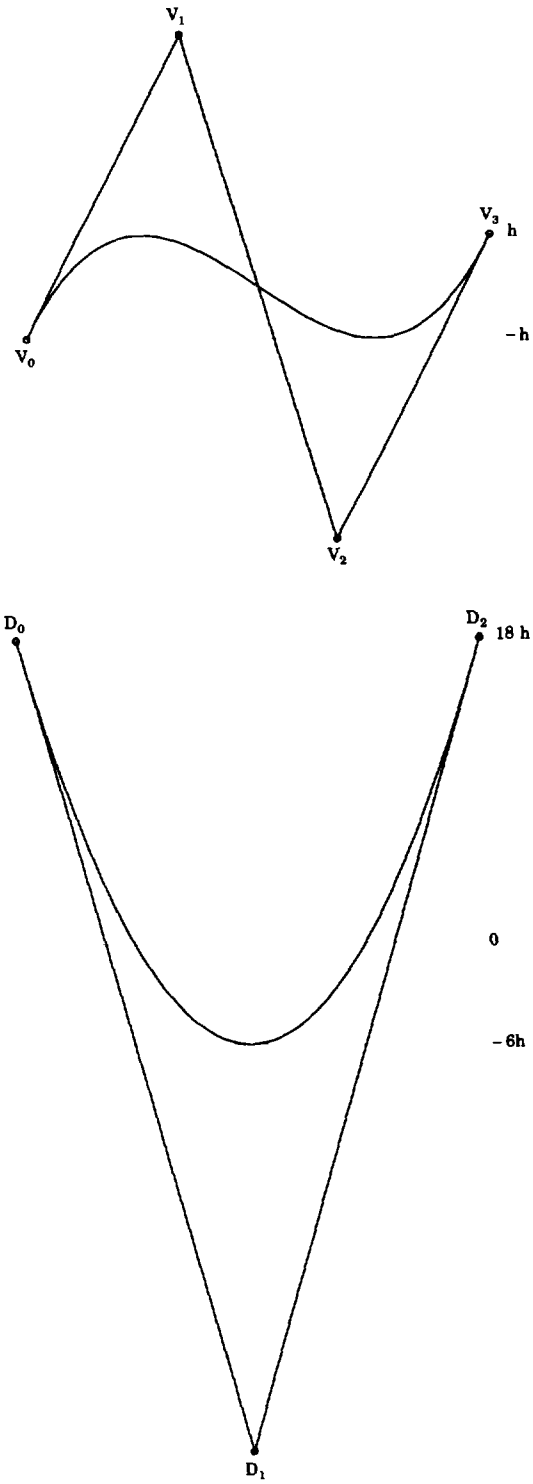


Fig. 11. The curve with derivatives of greatest magnitude. The derivative reaches $\pm 18h$ at the ends. In the text this curve is referred to as the S curve.

maximum values of the derivative are $6h$ and $8h$, for monotonic and nonmonotonic curves, respectively. For curves with nonmonotonic derivatives, the maximum values of the derivative are $8h$ for monotonic curves, $27h/2$ for curves having a single relative extremum in the interval, and $18h$ for curves with two relative extrema in the interval. The one that has the largest derivative has control vertices $[-h, 5h, -5h, h]$, and is shown in Figure 11. By the mean value theorem, we can then say that $\delta_1(t, \Delta) \leq 18h\Delta$. For convenience in the discussion that follows, this curve is called the S curve because of its S-like shape.

The other differences are easily shown to be maximized by the same construction. First, δ_3 is maximized, since the construction places v_1 and v_2 as far apart as possible, and v_3 and v_0 as far apart as possible, with $v_1 > v_2$ and $v_3 > v_0$. From (2), c_3 is clearly maximized, which means that by (3) δ_3 is.

By (3), $\delta_2 = \Delta^2 d^2 s/dt^2 + \delta_3$, so maximizing the second derivative maximizes the second difference. The second derivative of a cubic Bézier on $[v_0, v_1, v_2, v_3]$ is linear and proportional to the first order Bézier curve $[(v_2 - v_1) - (v_1 - v_0), (v_3 - v_2) - (v_2 - v_1)]$. This is maximized at one end point or the other; by reversing the order of the control vertices the end point of maximum value can be exchanged with the one of minimum value. With the arbitrary choice of $t = 1$, it is maximized by maximizing $v_3 - v_2$ and $v_1 - v_2$, yielding the S curve.

The S curve has differences

$$\begin{aligned}
\delta_1 &= (96h\Delta)t^2 + (-96h\Delta + 96h\Delta^2)t + (18h\Delta + 32h\Delta^3 - 48h\Delta^2) \\
\delta_2 &= 192ht\Delta^2 - 96h\Delta^2 + 192h\Delta^3 \qquad (7) \\
\delta_3 &= 192h\Delta^3.
\end{aligned}$$

δ_2 grows linearly from $-96\Delta^2 h(1 - 2\Delta)$ at $t = 0$ to $96h\Delta^2(1 - 2\Delta)$ at $t = 1 - 2\Delta$, the last time its value affects the result. δ_1, which is initially positive ($2h\Delta(4\Delta - 3)^2$), begins by falling to $-2h\Delta(3 - 4\Delta^2)$ at $t = \frac{1}{2} - \Delta/2$, and then rises to $2h\Delta(4\Delta - 3)^2$ at $t = 1 - \Delta$. The worst-case curve has its largest differences at $t = 0$ and $t = 1 - \Delta$ (and they are equal at those two points). This means that if the initialization is accomplished without overflow for the worst-case curve, then stepping does not cause overflow either.

6. INITIALIZATION

The details of the initialization depend on the basis being used to represent the curve prior to forward differencing, but for any reasonable basis, it is easy to avoid overflow in the values of the control vertices. Changing basis costs no more than 4×4 matrix-vector multiplication. Changing to the power basis involves no intermediate values greater than six times the value of the maximum control vertex, for any of the commonly used bases (see [1, sec. 10.8]). If overflow can be avoided in the initialization using the power basis, any other basis may be used equally well.

Equations (3) with $t = 0$ give the initialization for the power basis.

$$\delta_0 = c_0$$
$$\delta_1 = c_1\Delta + c_2\Delta^2 + c_3\Delta^3$$
$$\delta_2 = 2c_2\Delta^2 + 6c_3\Delta^3$$
$$\delta_3 = 6c_3\Delta^3$$

In practice, the expressions for the coefficients are normally expanded and the expressions for δ_i are simplified by hand prior to coding. For example, for the Bézier curve defined by the four control points $v_{0..3}$, the initial values are

$$\delta_0 = v_0$$
$$\delta_1 = 3(v_1 - v_0)\Delta + 3(v_0 - 2v_1 + v_2)\Delta^2 + (3(v_1 - v_2) + v_3 - v_0)\Delta^3$$
$$\delta_2 = 6(v_0 - 2v_1 + v_2)\Delta^2 + 6(3(v_1 - v_2) + v_3 - v_0)\Delta^3$$
$$\delta_3 = 6(3(v_1 - v_2) + v_3 - v_0)\Delta^3.$$

Noting that $\Delta = 1/n$, the difference expressions may be rewritten as

$$\delta_1 = \frac{c_1 n^2 + c_2 n + c_3}{n^3}$$
$$\delta_2 = \frac{2c_2 n + 6c_3}{n^3} \tag{8}$$
$$\delta_3 = \frac{6c_3}{n^3}.$$

The S curve maximizes not only the magnitudes of the differences, but also the magnitudes of any intermediate values used in calculating their initial values. This is because the S curve maximizes the magnitudes of the power basis coefficients.

There are several ways of choosing the value of n. One option is to choose a sufficiently high value that a linear approximation to the curve passes close enough to the actual curve for the purposes of the application. Wang has given an algorithm for finding such a value of n [15]. Another option is to choose a step size small enough that line segments are no longer (in screen space) than some number of pixels. Since δ_1 (in (1)) gives the step size in screen space at a given value of t, if there is a maximum screen space step size imposed, the S curve requires the largest number of steps to meet that constraint. If the constraint is that the maximum step be of unit length, then

$$1 \geq \delta_1 = \frac{18hn^2 - 48hn + 32h}{n^3}.$$

For this constraint to be met, n must be large enough that the leading term dominates, so there must be enough bits available for $n = 18h$.

Note that the choice of basis does not affect the above analysis. The curve specified requires the largest number of steps for a given maximum screen space step size, regardless of the basis in which it is represented.

7. ALGORITHM A

Field's Algorithm A for linear interpolation is the simple DDA [10] expressed in fixed point with proper rounding:

$x \leftarrow ((((a \ll 1) + 1) \ll Z) - 1) \gg 1$
$dx \leftarrow (((b - a) \ll (Z + 1)) + n)$ **div**$(n \ll 1)$
for i **from** 0 **to** n **do**
  output$(i, x \gg Z)$
  $x \leftarrow x + dx$

Here $x \ll a$ and $x \gg b$ are used to denote 'x shifted left a bits' and 'x shifted right by b bits' (using arithmetic shifts), to emphasize the multiplication and division by powers of two. $n + 1$ output values are generated uniformly over the interval $[a, b]$. Z bits of fractional precision are retained during the computation. x is increased by $\frac{1}{2}$ in the initialization so that the results are properly rounded. The strategy is to keep enough fractional precision to avoid single-pixel errors, and discard the low order bits of the result before using it. As long as all of the accumulated error is contained in the low order (guard) bits, no error appears. The largest error that can occur in the result is $\frac{1}{2} + 2^{-Z-1}n$. Z is selected to minimize the maximum error without causing overflow in dx.

For cubics, the loop becomes

for i **from** 0 **to** n **do**
  output$(i, (d_0 \gg Z))$
  $d_0 \leftarrow d_0 + (d_1 \gg Z)$
  $d_1 \leftarrow d_1 + (d_2 \gg Z)$
  $d_2 \leftarrow d_2 + d_3$

Here d_i is the machine representation of δ_i. (This loop appears on page 406 of [1] but the initialization shown in the first printing is incorrect.) With $Z \geq \lceil \log_2 n \rceil$, the error in each of the d_i is contained within the guard bits for all n steps, and does not propagate from term to term. As long as the errors in any one term are contained within the guard bits, only a linear growth in error can be expected in each of the differences. To avoid single pixel errors due to rounding, the errors in d_0 should be contained within half of the first guard bit, hence one extra bit is needed. Thus d_0 is maintained shifted up by $Z + 1$ bits, d_1 is shifted an additional Z bits for a total of $2Z + 1$, and d_2 and d_3 are shifted $3Z + 1$ bits. This changes the second line to

output$(i, d_0 \gg (Z + 1))$.

The initialization analogous to Field's Algorithm A is as follows:

$l \leftarrow n^3$
$d_0 \leftarrow (((c_0 \ll 1 + 1) \ll (Z + 1)) - 1) \gg 1$
$d_1 \leftarrow (((c_1 n^2 + c_2 n + c_3) \ll (2Z + 2)) + l)$ **div**$(l \ll 1)$
$d_2 \leftarrow (((2c_2 n + 6c_3) \ll (3Z + 2)) + l)$ **div**$(l \ll 1)$
$d_3 \leftarrow (((6c_3) \ll (3Z + 2)) + l)$ **div**$(l \ll 1)$.

The idiom $(x \ll (k + 1) + y)$ **div**$(y \ll 1)$ achieves the division of $x \ll k$ by y with proper rounding.

For any reasonable values of n and Z there is overflow at some point in the above computation. The largest intermediate value in the computation of d_1

is

$$((c_1 n^2 + c_2 n + c_3) \ll (2Z + 2)) + l. \quad (9)$$

For the S curve, if the step size is never greater than one pixel, ($n = 18h$), (9) becomes:

$$\left(((18h)(18h)^2 + (-48h)(18h) + (32h)) \ll (2Z + 1)\right) + (18h)^3. \quad (10)$$

Prior to the left shift, this has a term of $18^3 h^3$. For $h = 256$, $18^3 h^3$ is greater than 2^{36}. Even if a considerably smaller number of steps is taken (for approximating the curve with line segments), the left shift is likely to cause overflow. Performing the divisions as early as mathematically possible would avoid overflow, but would also lead to serious accuracy problems.

It is possible to permute some operations without loss of accuracy, and in this way reduce the size of the largest intermediate value. Beginning with the simplest case, the largest intermediate value in the computation of d_3 is $(((6c_3) \ll (3Z + 2)) + l)$. The algorithm for computing d_3 implied by this expression appears below, along with the largest intermediate value of each subexpression.

Statement	*Maximum Intermediate Value*
$shift \leftarrow 3Z + 2$	$3Z + 2$
$divisor \leftarrow l \ll 1$	$2n^3$
$temp < 6c_3$	$192h$
$d_3 \leftarrow ((temp \ll (shift)) + n^3)$ **div** $divisor$	$192h2^{3Z+2} + n^3$

The naive algorithm for computing δ_3.

To avoid single pixel errors, $Z = \lceil \log_2 n \rceil$. Suppose $2^Z = 2n - 2$. Shifting left by $3Z + 1$ is equivalent to multiplying by $16n^3 - O(n^2)$. The last intermediate value in the computation is $3072h(n - 1)^3 + n^3$. If h is 256 (suitable for a 512 × 512 display), then n must not be greater than 14 if this value is to be stored in a signed 32-bit integer. Since this formulation allows so few steps without possible overflow, the method's setup is too costly to be practical.

A left shift of Z is equivalent to a multiplication by a value less than $2n$. In the computation of d_3, a shift of $3Z + 1$ bits is applied before a division by $2n^3$. The same effect, with rounding, is achieved by pairing each shift by Z with a division by n as follows:

$divisor \leftarrow 2n$
$d_3 \leftarrow 6c_3$
$d_3 \leftarrow ((d_3 \ll (Z + 2)) + n)$ **div** $divisor$
$d_3 \leftarrow ((d_3 \ll (Z + 1)) + n)$ **div** $divisor$
$d_3 \leftarrow ((d_3 \ll (Z + 1)) + n)$ **div** $divisor$

Now the largest intermediate value, $768nh + n$, occurs just before the final division. With $h = 256$, n can be as large as 10922 using twos complement 32-bit integers without overflow.

A similar transformation may be applied to the computation of d_2. Divide through top and bottom of (8) by n,

$$\delta_2 = \frac{2c_2 + 6c_3/n}{n^2}.$$

In fixed point:

$d_2 \leftarrow ((6c_3 \ll (Z+2)) + n)$ **div** *divisor*
$d_2 \leftarrow (((d_2 + (2c_2 \ll (Z+1))) \ll (Z+1)) + n)$ **div** *divisor*
$d_2 \leftarrow ((d_2 \ll (Z+1)) + n)$ **div** *divisor*

After the first assignment, d_2 is shifted left $Z+1$. $2c_2$ is correspondingly scaled before being added in. Prior to each of the following divisions, another shift of Z bits is applied. The largest intermediate value in d_2 now occurs just prior to the last division, and it is $(\delta_2 2n + n)2^{3Z+1}$, which for the S curve is $(((192h/n - 96h)/n^2)2n + n)2^{3Z+1}$. Again using the example of $h = 256$, this is $((49152/n - 24576)/n^2)2n2^{3Z+1}$. With $n = 128$, $Z = 7$ and this is $-1585446912 > -2^{31}$. With $n = 129$, $Z = 8$, so the largest intermediate value becomes -12586801579, which is just greater than -2^{34}.

It is still possible to increase the maximum value of n by performing the last shift/division pair as a single operation. On a machine permitting it (such as a member of the VAX family), the operation may be coded with extended precision integer instructions, using a register pair for the intermediate values, at a cost only slightly higher (25% on a VAX 11/780) than that of performing the shift and divide with regular integer precision. Otherwise, it requires a small loop, similar to the one used for division on machines lacking a divide instruction (see Appendix). On a VAX 11/780, the cost of such a loop is roughly ten times the cost of a single shift and divide. For lack of previous notation, let

$$A \div \begin{matrix} c \\ b \end{matrix}$$

denote such a combination of shifting A left by b while dividing by c.

If n is large enough to justify the use of extended division, then as long as overflow does not occur in computing the numerators of (8), the best way of computing d_1 is the obvious way:

$$d_1 = ((c_1 n + c_2)n + c_3) \div \begin{matrix} 2^{2Z+1} \\ n^3 \end{matrix}$$

With $h = 256$, n may be as large as 963 without overflow occurring in the initialization. For larger n, overflow occurs in the second multiplication by n in the S curve. To avoid this, one division by n may be carried out early:

$d_1 \leftarrow c_1 n + c_2 + ((2c_3 + n)$ **div** *divisor*$)$
$d_1 \leftarrow d_1 \div \begin{matrix} 2^{2Z+1} \\ n^2 \end{matrix}$

Information lost in the early division would be lost eventually, since d_1 only has $2Z+1$ bits of fractional precision.

When n^3 is sufficiently large that it does not fit in a machine word ($n > 1024$ for a 32-bit word), all divisions by n^3 and shifts of $3Z$ must be split into a division by n and shift of $2Z$ followed by a division by n^2 and shift of Z.

With the shift and divide combined, n may be considerably higher without overflow. In this case, overflow in intermediate values of the differences is not a problem. The constraining factor becomes overflow in the initial values of the numerators, particularly of d_2. For the S curve, $\delta_2 = (-96hn + 192h)/n^3$. For large values of n, this is essentially $-96h/n^2$. If n is a power of two, then $n = 2^Z$, and $n^2 = 2^{2Z}$. This means that $d_2 = -96h2^{Z+1} =$

$-184hn$, so $|d_2| < 184hn$. (Actually the maximum screen size is slightly larger because of the second order term in δ_2.) If d_2 is to be stored in a 32-bit word, the product of h and n (or the next power of two) must be less than 11671106. Two possibilities are ($n \leq 1024$, $h \leq 11397$) (11.4 inches at 1000 dpi), and ($n \leq 16384$, $h \leq 712$) (1424 line high-resolution monitor). Note that in the latter case $n > 18h$, so that this is more than enough to touch every pixel on the S curve. With all of the above changes, the large n initialization of Algorithm A from Bézier control points is as follows:

```
cieln ← 1;
Z ← 0;
  Compute bits of Z - ceil(log2(n))
while(cieln < n)
    cieln ← cieln + cieln;
    Z ← Z + 1;
  d0 maintained shifted Z + 1 left
d0 ← v0 ≪ (Z + 1) + (1 ≪ Z);
  Computation of d1 - maintained shifted 2*Z + 1 left
d1 ← ((v1 - v0) × n + v0 - v1 - v1 + v2) × 3;
d1 ← d1 + (2 × (3 × (v1 - v2) + v3 - v0 + n)) div (n + n);
d1 ← Divide(d1, n × n, Z + Z + 1);
  Computation of d2 - maintained shifted 3*Z + 1
d2 ← 6 × (3 × (v1 - v2) + v3 - v0);
d2 ← d2 + 6 × (v0 - v1 - v1 + v2) × n;
d2 ← Divide(d2, n, Z + 1);
d2 ← Divide(d2, n × n, 2Z);
  Now d3
d3 ← 6 × (3 × (v1 - v2) + v3 - v0);
d2 ← Divide(d3, n, 2Z + 1);
d2 ← Divide(d3, n × n, Z);
```

In the error analysis, it was assumed that the initial errors in the three differences were bounded by the same constant, e. If the difference terms are computed as suggested above, the ordering of the divisions and shifts guarantees that the final error in each of the terms is bounded by $\frac{1}{2}$ of the least significant bit.

8. ALGORITHM B

For Algorithm B, Field generalizes Bresenham's algorithm to arbitrary numbers of steps, and makes it reversible, so it does not matter whether a line is drawn from a to b or b to a. The basic ideas are as follows. The variables x and dx are each split into integer and fractional parts. Whenever the fractional part of x would leave the $[0, 1)$ range, x is incremented by a constant ± 1 which depends on the direction in which the line is being drawn. To eliminate the asymmetry caused by the half-open interval, a small value is added to x, causing it to round the same way for either direction of travel. This is essentially the method of Boothroyd and Hamilton [2]. Floating point is eliminated by scaling the fractional parts by $2n$.

One of the properties of lines that the algorithm requires is that the sign of dx is constant. This is only the case for monotonic spline curves, and even then, the sign of the second difference may change. Because this property is used early in the development of Field's algorithm, it is necessary to develop the cubic version from scratch, using many of the same ideas.

To reduce the bulkiness of the presentation, the position and differences are represented as elements in an array, and a single step, in which the position and each of the differences is updated, is shown as a loop. In practice, the loop should be unrolled for efficiency. The initial algorithm is as follows.

$d_0 \leftarrow c_0 + 0.5$
$d_1 \leftarrow (c_3 + (c_2 + c_1\Delta)\Delta)\Delta$
$d_2 \leftarrow (2c_2 \times \Delta + 6c_3)\Delta \times \Delta$
$d_3 \leftarrow 6c_3 \times \Delta \times \Delta \times \Delta$
for $i \leftarrow 0$ **to** n
  output$(i, \textbf{trunc}(d_0))$
  for $j \leftarrow 0$ **to** 2
    $d_j \leftarrow d_j + d_{j+1}$

Note that the power basis coefficients may be computed from Bézier control points using only exact integer operations. As in Field's development of Algorithm B, the first modification separates the differences into integer and fractional parts.

for $j \leftarrow 0$ **to** 3
  $ip_j \leftarrow \textbf{floor}(d_j)$
  $fp_j \leftarrow d_j - ip_j$
for $i \leftarrow 0$ **to** n
  output(i, ip_i)
  for $j \leftarrow 0$ **to** 2
    if $fp_j + fp_{j+1} \geq 1$
      $ip_j \leftarrow ip_j + ip_{j+1} + 1$
      $fp_j \leftarrow fp_j + fp_{j+1} - 1$
    else
      $ip_j \leftarrow ip_j + ip_{j+1}$
      $fp_j \leftarrow fp_j + fp_{j+1}$

Using the floor function rather than truncating toward zero guarantees that the fractions are always positive.

The next step is to make everything integral by multiplying the fractional part by $2n^3$.

$ip_0 \leftarrow c_0$
$fp_0 \leftarrow n^3$
$d_1 \leftarrow 2((c_3 n + c_2)n + c_1)$
$d_2 \leftarrow 4c_2 + 12c_3 n$
$d_3 \leftarrow 12c_3$
for $j \leftarrow 1$ **to** 3
  if $d_j \geq 0$
    $ip_j \leftarrow d_j$ **div** $(2n^3)$
  else
    $ip_j \leftarrow d_j$ **div** $(2n^3) - 1$
  $fp_j \leftarrow d_j - 2n^3 ip_j$
for $i \leftarrow 0$ **to** n
  output(i, ip_i)
  for $j \leftarrow 0$ **to** 2
    if $fp_j + fp_{j+1} \geq 2n^3$
      $ip_j \leftarrow ip_j + ip_{j+1} + 1$
      $fp_j \leftarrow fp_j + fp_{j+1} - 2n^3$
    else
      $ip_j \leftarrow ip_j + ip_{j+1}$
      $fp_j \leftarrow fp_j + fp_{j+1}$

Note that c_0 is an integer since the control points are integer-valued. The position is stored in ip_0 and fp_0, with $0 \le fp_0 < 2n^3$ as before. The higher order differences still have the property that $ip_j + fp_j/(2n^3) = d_j$, but now $0 \le fp_j \le 2n^3$. fp_j reaches the high limit when $d_j \equiv 0 \bmod (2n^3)$. The case of $fp_j = 2n^3$ is correctly handled in the first branch of the **if** in the main loop.

In order to simplify the expression in the **if** let

$$r_i = fp_i + fp_{i+1} - 2n^3, \qquad i = 1..3$$

Precomputing these values saves performing the additions in the test, and reduces the amount of computation in the **if** block. In order to keep r_i up to date, it must be adjusted every time either the corresponding fp_i or its successor is changed. The value of fp_{i+1} which should be added is the value after fp_{i+1} is updated, so that at the test next time through the loop, r_i correctly reflects the states of both fp_i and fp_{i+1}.

```
for i ← 0 to 2
  if r_i ≥ 0
    ip_i ← ip_i + ip_{i+1} + 1
    fp_i ← r_i
    r_i ← r_i − 2n^3
  else
    ip_i ← ip_i + ip_{i+1}
    fp_i ← fp_i + fp_{i+1}
  if i > 0
    r_{i−1} ← r_{i−1} + fp_i
r_2 ← r_2 + fp_3
```

(The last **if** disappears when the loop is unrolled). The final assignment need not be postponed, since fp_3 is a constant. When the loop is unrolled, the last assignment may be absorbed into the two branches of the **if**, and then since $r_3 = fp_3 - 2n^3$, the assignment to r_2 in the **if** branch may be replaced with the line $r_2 \leftarrow r_2 + r_3$, saving one addition when the **if** branch is executed. The initialization is still prone to overflow for large curves. The largest intermediate value is

$$d_1 = 2\big((c_3 n + c_2)n + c_1\big), \tag{11}$$

which for the S curve from (7) is

$$\begin{aligned} d_1 &= 2\big((32hn + -48h)n + 18h\big) \\ &= h(64n^2 - 96n + 36). \end{aligned}$$

If $h = 2^8$, n is limited to 64, which is not very good, considering values as high as $18h$ might be required. Note, however, that d_1 is eventually stored as ip_1 and fp_1, the integer and fractional parts. To reduce the size of the largest intermediate value, the individual terms in (11) may be computed as integer and fractional parts. The purpose of the leading 2 in (11) is to put d_1 on the same scale as fp_1; d_2 and d_3 are similarly scaled. With ip_j and fp_j

calculated from first principles, the d_i are never computed explicitly, and so this factor is not needed. For this reason the divisions are by n^3, and not $2n^3$.

```
if c1 ≥ 0
  ip1 ← c1 div n^3
else
  ip1 ← c1 div n^3 − 1
fp1 ← 2(c1 − ip1 n^3)
if c2 ≥ 0
  ipart ← c2 div n^2
else
  ipart ← c2 div n^2 − 1
fpart ← 2(c2 − ipart × n^2)
if fp1 + fpart > 2n^3
  fp1 ← fp1 − 2n^3 + fpart
  ip1 ← ip1 + ipart + 1
else
  fp1 ← fp1 + fpart
  ip1 ← ip1 + ipart
if c3 ≥ 0
  ipart ← c3 div n
else
  ipart ← c3 div n − 1
fpart ← c3 − ipart × n
fpart ← fpart + fpart
if fp1 + fpart > 2n^3
  fp1 ← fp1 − 2n^3 + fpart
  ip1 ← ip1 + ipart + 1
else
  fp1 ← fp1 + fpart
  ip1 ← ip1 + ipart
```

Similar manipulations may be performed with the calculation of ip_2 and fp_2. Computing all of the integer parts and fractional parts in this way corresponds to computing them using the expressions

$$\delta_1 = c_1 / n^3 + c_2 / n^2 + c_3 / n$$

$$\delta_2 = 2c_2 / n^3 + 6c_3 / n^2$$

$$\delta_3 = 6c_3 / n^3$$

and storing integer and fractional parts separately as each subexpression is computed. This results in an additional two divisions in the initialization, which are not necessary for small values of n, or curves in which it is known that the control vertices are close enough together to prevent overflow in (11).

For the worst-case curve, the largest intermediate value is now

$$6c_3 = 192h.$$

This is independent of the number of steps, and small enough that for all reasonable values of h, there can not be overflow in its representation.

The integer parts of each of the differences is bound to stay within range, if the curve remains on the screen and if the word size is at least two bits larger

Table I. Comparative Operation Counts for the Two Algorithms

Operation	Initialization Algorithm A	Algorithm B
+/-	17+3Z	40
*	12	12
div	1	5
extended div	5	0
shift	2Z−1	0
branch	Z	6
assign	28+2Z	33
Operation	**Inner loop Algorithm A**	**Algorithm B**
+/-	3	13
branch	0	3
shift	2Z	0
assign	3	10

than the number required to address the entire screen. If the position remains on screen, the first difference may never be greater in magnitude than one screen width. If this is the case, the second difference may never exceed two screen widths in magnitude, and the third difference may never exceed four screen widths in magnitude. (Any of the differences may go out of range in the last steps of a curve, when that difference is no longer used).

The fractional parts of each of the differences are stored exactly as a fraction of $2n^3$. Since they are always in the range $[0, 1)$, they must fit in $3\lceil \log_2 n \rceil + 1$ bits. The biased fractional values, r_i, take on values in the wider range $[-1, 1)$. These are also stored as fractions of $2n^3$, so they fit in $3\lceil \log_2 n \rceil + 2$ bits, including a sign bit. These taken together imply that for a 32-bit machine word, the number of steps is limited to 1024, independent of the screen resolution. Each additional three bits of machine word permit the maximum number of steps to be doubled.

9. COMPARISON

Algorithms A and B can be compared on several bases: initialization cost, incremental cost, and limitations.

The relative running times of the two algorithms depends on the processor. On a machine with a barrel shifter the costs of the two algorithms are similar. Table I gives the operation counts for the two algorithms, in which n is assumed to be large. For either algorithm, more efficient code may be implemented for small n, since much of the effort in the initialization is intended to prevent overflow. For the purposes of the comparison in Table 1, this code is not implemented, nor is the size of n tested to see whether the

more efficient code would suffice. (The merit of testing the size of n depends on the expected frequency of small values of n).

Where two branches of an **if** block are available, it is assumed that they are taken equally often, which for this algorithm is not a bad assumption. (In a large ensemble of lines of random slopes, the two branches of the test in the inner loop of Bresenham's algorithm are taken equally often. This is analogous). The test associated with a branch is counted as an additive operation, unless the test involves an expression which is being compared with zero. In this case the condition codes would normally be set during evaluation of the expression.

The count of assignments is somewhat misleading. On a relatively high performance machine, rich in registers or having a fast operand cache, assignments are essentially free. On a machine without a barrel shifter the cost of stepping is greater in Algorithm A than B, since the two shifts by Z bits have to be implemented as loops of single bit shifts. On a VAX 8600, the costs of the two are comparable, since a shift is approximately 5.5 times as expensive as an add. If $Z > 4$, the initialization of Algorithm B is a clear winner, regardless of the processor. For processors with neither hardware extended division instructions nor multiple bit shifts, the difference is a factor of two or more.

The total number of steps possible for half-pixel accuracy is much greater for Algorithm A. For long curves, this means that the curves can be drawn in their entirety using Algorithm A, but must be split into multiple segments to be drawn using Algorithm B. Thus, the initialization is amortized over a larger number of steps, entirely compensating for the increased initialization time, if sufficiently long curves are drawn. A slightly conservative estimate of the number of bits of precision required for Algorithm A is given by $\lceil \log_2(184hn) \rceil$; for Algorithm B $\lceil 3 \log_2 n \rceil + 2$ bits are required, independent of h, as long as the control points fit in a machine word.

Either algorithm may be used in an antialiased line drawing program in which the positions are computed on a high-resolution grid, but only as many positions are computed as would be needed for a lower resolution. As the resolution of the high-resolution grid is increased, fewer points may be plotted using Algorithm A, but Algorithm B remains usable until the integer parts of the differences no longer fit in a machine word, since Algorithm B is practically immune to the size of the grid (as long as the grid is less than 2^{21} in its longest dimension).

In a recent paper, Chang et al. [4] present a method of integer adaptive forward differencing. This was independently developed during the same time period as the algorithms presented here. They claim to be able to step up to 2^{13} steps in a screen space of dimensions 2^{16} with a 32-bit machine word. It appears that the screen space refers to the space containing the control points, not the curve, and that the initialization is performed in floating point (double precision). The primary difference between their algorithm and the two presented here is that theirs requires a step size small enough that steps are on the order of one pixel, such a step size being guaranteed by adaptive forward differencing. For polyline approximations,

steps need not be so small, which can lead to differencing coefficients, that, in their representation, do not fit in an integer word if n is small. The algorithms presented here do not have that limitation.

10. A HYBRID ALGORITHM

By sacrificing some of the accuracy of Algorithm B, it is possible to increase the number of steps. Suppose the fractional values are stored multiplied by n^3 and rounded, rather than being multiplied by $2n^3$. The initial errors in the differences are bounded by $1/2n^3$, and they increase at each step. After n steps, e_0 is still less than $1/10$. If errors are to be kept less than $1/2$, even more steps may be taken. If the fractional values are stored multiplied by $n^3/2$, then the initial errors are bounded by $1/n^3$. In general, if the values are stored multiplied by $2n^3/k$ then the errors are bounded by $k/4n^3$. After n steps, e_0 is less than $k/20$. In order to keep the errors to less than $1/2$, k as high as 10 may be used. If $2n^3r_i$ can be stored in $3\log_2 n + 2$ bits, then $2n^3r_i/10$ can be stored in $\log_2 10$ fewer bits, so the maximum value of n can be increased by a factor of 2.154.

The advantage of such a compromise is an increase in the total number of steps available over Algorithm B, without the use of shifts required by Algorithm A. Even on a machine for which multiple bit shifts are expensive operations, this keeps the cost comparable to the cost of Algorithm B.

11. NONINTEGRAL ENDPOINTS

Algorithm A can be easily adjusted to allow nonintegral endpoints. In the initialization, points are prerounded by storing the fixed point value of $v_0 + \frac{1}{2}$ in d_0. There is no reason that v_0 must be an integer for this to work. Similarly, all of the other control vertices need not be integers. For small numbers of steps, the number of bits of fraction will be small. In order to avoid losing too much precision, a minimum number of fractional bits can be imposed.

Algorithm B can be similarly provided with fractional control vertices. The initialization must be adjusted to retain the fractional information, which is rounded to the nearest multiple of $1/n^3$. For small values of n, a small multiple of n may be substituted for n in the representation of 1, to provide more bits of fractional accuracy.

12. HARDWARE IMPLEMENTATION

The stepping parts of both algorithms are able to take advantage of pipelining and parallelism. In a hardware implementation, the shifts of Algorithm A can be done in wiring, and the three adds can be done in parallel. The total time required for one step is the time required to add two numbers and latch the result.

Algorithm B is somewhat more complicated. The three passes through the loop can be performed in parallel. For each pass, both branches of the **if** can be executed in parallel, reserving the assignments until the test has been completed. The expressions in each of the statements in each branch are independent, so they can be performed in parallel. The value which takes

longest to be ready is r_i, since it has to wait for fp_{i+1}, which is not ready until an addition has been performed. Assuming all possible branches of the **if**s are performed in parallel, and values are computed as their input data becomes ready, a single step requires the time of two additions and a demultiplex to select the correct answer from those computed. This is only one addition more than the time required to perform a step of Bresenham's (linear) algorithm.

13. SUMMARY

Two incremental cubic interpolation algorithms have been derived and analysed for speed and accuracy. Algorithm A is a generalization of the simple DDA employing fixed point arithmetic, and Algorithm B is a new algorithm using only integral arithmetic which is a generalization of Bresenham's line drawing algorithm. Algorithm B achieves perfect accuracy, and the speed tradeoff depends on the processor.

Since the algorithms are generalizations of the linear algorithms of Field (which originate in simple DDA and Bresenham's algorithm), when given the control points defining a straight line in parameter space, they generate the same points. It is an easy exercise to specialize these two algorithms to quadratics, since all of the parts of Field's algorithms which do not generalize to quadratics trivially have been handled in the cubic case. The bounds on the number of steps are significantly higher for quadratics. Generalization to higher order follows in a straightforward manner. While the maximum allowable number of steps falls as the order increases, the number that may be required to avoid missing pixels rises. As the order increases, the ability to split a curve in the optimum location (the location for which one piece requires the maximum allowable number of steps in order to hit every pixel) becomes more important. This is an issue which, to this author's knowledge, has yet to be addressed.

As part of the analysis, it was found that the cubic curves with the largest difference values are those which start at one edge of the screen, cross the screen three times, and end at the opposite edge. (This was derived in one dimension only. Any curve with this set of control values in either dimension has large differences in that dimension). It is interesting to note that this family of curves is the worst-case curve for many purposes. These curves have Bézier control points furthest from each other, and require the largest number of uniform steps to traverse at a given minimum density. (The density is normally determined independently in x and y, and then the minimum of the two step sizes is used). Since the derivative vector of the Bézier curve $[v_0, v_1, v_2, v_3]$ is the lower order curve $[v_1 - v_0, v_2 - v_1, v_3 - v_2]$, and this curve has its control points as far apart as possible, the magnitude of the derivative vector is largest at the ends of the curve which crosses the screen three times in each dimension. This happens to be a straight diagonal line which is traversed three times as t ranges from 0 to 1.

A number of algorithms for drawing curves and surfaces split the curves and silhouettes into monotonic segments before drawing them [7, 12, 16]. It

would be interesting to see what the condition of monotonicity does to the maximum values of step count and the size of the difference terms.

The generalization of these algorithms to adaptive forward differencing [8, 14] is relatively straightforward. (See Klassen [7] and also Chang et al. [4]) At each halving or doubling of the step size, the value of n is effectively doubled or halved respectively. The amount of shifting in Algorithm A, and the implied denominator in the fractions in Algorithm B must be adjusted accordingly. If binary search is used to avoid replotting points and possibly missing some, then care must be taken in computing the new values after the search [7]. Algorithm B loses its perfect accuracy as a result of the change in denominator, while Algorithm A loses no more than the amount lost at each step.

APPENDIX. Algorithm for Combined Left Shift and Division.

```
Divide dividend × 2^shift by divisor yielding quotient
high ← 0   The dividend is shifted left into this word.
quotient ← 0   Answer bits are shifted in from the right as calculated.
sign ← false
  Note the signs of the operands, then make them positive.
if dividend < 0
  dividend ← −dividend,
  sign ← ¬ sign
if divisor < 0
  divisor ← −divisor
  sign ← ¬ sign
  Standard long division loop, with adjusted loop counter.
for count ← 1 to wordsize + shift
  quotient ← quotient ≪ 1
  if dividend < 0   The sign bit is set.
    high ← high + 1
  if divisor ≤ high
    high ← high − divisor
    quotient ← quotient + 1
  dividend ← dividend ≪ 1
  high ← high ≪ 1
If an additional iteration would set the next bit of the quotient, add one to round.
The dividend (low order word) is guaranteed zero.
if divisor ≤ high
  quotient ← quotient + 1
if sign
  quotient ← −quotient
else
  quotient ← quotient
```

ACKNOWLEDGMENTS

It was due to comments made by Richard Bartels on forward differencing in general that the author began analysing the numerical properties of integer forward differencing. Discussions with John Beatty (on his algorithm), Ron Goldman (on properties of Bézier curves), and Kelly Booth (on precision requirements) contributed to various parts of this paper. Most of the work

reported in this paper was performed while the author was a doctoral candidate in the Computer Graphics Lab at the University of Waterloo.

One of the anonymous referees pointed out that the extra guard bit required for half-pixel accuracy was not needed for the higher order differences. This added a factor of four to the number of steps possible.

REFERENCES

1. BARTELS, R. H., BEATTY, J. C., AND BARSKY, B. A. *An Introduction to Splines for Use in Computer Graphics and Geometric Modelling*. Morgan-Kaufmann, Palo Alto, Calif., 1987.
2. BOOTHROYD, J., AND HAMILTON, P. A. Exactly reversible plotter paths. *Australian Comput. J. 2*, 1 (1970), 20-21.
3. BRESENHAM, J. E. Algorithm for computer control of a digital plotter. *IBM Syst. J. 4*, 1 (1965), 25-30.
4. CHANG, S-L., SHANTZ, M., AND ROCCHETTI, R. Rendering cubic curves and surfaces with integer adaptive forward differencing. *Comput. Graph. 23*, 3 (Jul. 1989), 157-166.
5. FIELD, D. E. Incremental linear interpolation. *ACM Trans. Graph. 4*, (Jan. 1985), 1-11.
6. FOLEY, J., AND VANDAM, A. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass., 1983.
7. KLASSEN, R. V. Drawing antialiased cubic spline curves. *ACM Trans. Graph.* (Jan. 1991).
8. LIEN, S-L., SHANTZ, M., AND PRATT, V. Adaptive forward differencing for rendering curves and surfaces. *Comput. Graph. 21*, 4 (Jul. 1987), 111-118.
9. NEWMAN, W. M., AND SPROULL, R. F. *Principles of Interactive Computer Graphics*. 1st ed., McGraw-Hill, New York, 1973.
10. NEWMAN, W. M., AND SPROULL, R. F. *Principles of Interactive Computer Graphics*. 2nd ed., McGraw-Hill, New York, 1979.
11. ROGERS, D. F. *Procedural Elements for Computer Graphics*. McGraw-Hill, Toronto, 1985.
12. SCHWEITZER, D., AND COBB, E. Scanline rendering of parametric surfaces. *Comput. Graph. 16*, 3 (Jul. 1982), 265-271.
13. SEDERBERG, T. W., AND ZUNDEL, A. K. Scan line display of algebraic surfaces. *Comput. Graph., 23*, 3 (Jul. 1989), 147-156.
14. SHANTZ, M., AND CHANG, S-L. Rendering trimmed NURBS with adaptive forward differencing. *Comput. Graph. 22*, 4 (Aug. 1988), 189-198.
15. WANG, G. The subdivision method for finding the intersection between two Bézier curves or surfaces. *Zhejiang University Journal, Special Issue on Computational Geometry spline*, 1984, (in Chinese).
16. WHITTED, J. T. A scan line algorithm for computer display of curved surfaces. *Comput. Graph. 12*, 3 (Aug. 1978), p. 8.

Received August 1988; revised September 1989, March 1990; accepted May 1990

Editor: Bob Sproull