ELECTRONIC PUBLISHING, VOL. 6(3), 171–181 (SEPTEMBER 1993)

Rasterizing the outlines of fonts

FIAZ HUSSAIN
School of Computing and Mathematical Sciences
De Montfort University, Kents Hill Campus,
Hammerwood Gate, Milton Keynes, MK7 6HP
email: fhussain@dmu.ac.uk

MICHAEL L. V. PITTEWAY
Department of Computer Science,
Brunel University,
Uxbridge, Middlesex, UB8 3PH, England
email: Mike.Pitteway@brunel.ac.uk

SUMMARY

Two mathematical descriptions of outlines which have found acceptability and widespread usage are the Bézier cubic and the general conic forms (which include the distinctive parabolic format). Though there are good reasons for employing just the general conic, PostScript characterises fonts in terms of splines based on four-point Bézier cubics.

In order to improve the efficiency with which these PostScript fonts can be rendered, the equation of the Bézier cubic is here reduced to the non-parametric form required to exploit an efficient cubic tracking algorithm first presented in 1968. Although successful in most cases, the occasional breakdowns are both spectacular and disastrous. The cause of the problem is analysed, and possible solutions suggested.

KEY WORDS Algorithm Bézier Conic Fonts Rasterize Spline

1 INTRODUCTION

Capturing the changing nature of character outlines can either be done as contours or as bitmaps. *Contour* refers to a mathematical description for the profile of a character, given in terms of equations of curves and lines. The attraction of this representation is that it provides a precise, size-independent description of a character, which can be easily scaled and rotated. Most output devices such as display screens and hard-copy printers, however, are raster output devices, which require fonts to be bitmapped.

In 1965, Jack Bresenham presented an all-integer arithmetic algorithm which defines a 'best-fit' approximation to a straight line drawn between two integer mesh points in an arbitrary direction [1]. Although the algorithm was primarily developed for computer control of a digital plotter, it can equally be applied to bitmapping outlines on modern displays and laser printers. Since the arithmetic is exact, the algorithm guarantees that the string of incremental moves generated will always pass through the end point precisely. The inner loop of the algorithm is sketched in Figure 1. (The 1965 algorithm was written for a plotter with diagonal moves also available, and is thus applicable to generating an

repeat until finish
if $d < 0$ then $\{x = x_{step}, d = d + b\}$
else $\{y = y_{step}, d = d - a\}$

Figure 1. Bresenham's line-drawing algorithm

CCC 0894–3982/93/030171–11

Received 1 September 1993
Revised 3 December 1993

repeat until finish
if $d < 0$ then { if $b < -0.5 \times K_2$ then { change sign: d,b,K_1,K_3,y_{step},
$b = b - K_2, d = d - a$ }
else { $x = x + x_{step}$,
$a = a + K_2, b = b - K_1, d = d + b$} }
else { if $a < 0.5 \times K_2$ then { change sign: d,a,K_1,K_3, x_{step}
$a = a + K_2, d = d + b$ }
else { $y = y + y_{step}$,
$a = a + K_3, b = b - K_2, d = d - a$ } }

Figure 2. Pitteway's conic drawing algorithm (square move only)

8-way-connected straight-line path. For simplicity, the three algorithms outlined here are all given for 4-way-connected paths).

For a line drawn in the 'first quadrant', and involving a total movement of u increments in the positive x direction and v increments in the positive y direction, the initial conditions require b to set equal to $2v$, a to $2u$, and d initialised to $v - u$. The algorithm passes through the branch point test on the sign of d a total of $u + v$ times, drawing a line with u x-steps and v y-steps spread as evenly as possible to represent a straight line. The other quadrants are dealt with in a similar manner.

The exact integer control offered by Bresenham's algorithm is an important feature when it is used to trace the outline of a polygon representing a letter, numeral or some other 'icon' for display, as it guarantees that the polygon can be closed precisely, with the start and end point coincident. If this were not the case, it would be necessary to bring in extra code to prevent a subsequent fill of the outline from bleeding through the gap and spoiling the display.

Since most fonts found in printing and for displaying readable text involve curved outlines, the employment of Bresenham's algorithm leads to a large number of line segments being used to make the curves appear reasonably smooth. Alternatively, Bresenham's algorithm itself can be adapted by arranging for the constants b and a to also vary, so that the gradient changes within the inner loop to produce a curved arc directly. The mathematical description of each curved arc is more complicated than for a straight line, and requires the storing of additional parameters, but hopefully far fewer arcs are required, so there is a net gain in the storage necessary to describe a complete font (26 lowercase letters, 26 uppercase letters, 10 numerals and various punctuation marks and the odd symbols required by French, German and the like). Unlike the straight-line description, the curved segments can be piecewise connected so that at the joining points (often called *knots*) there is gradient continuity, and in some cases, curvature continuity is also gained.

One scheme which rasterizes curved outlines directly is shown in Figure 2. It was first published in 1967 [2], and can be used to track any one of the curve sections belonging to the conic family (circles, ellipses, hyperbolas and parabolas). The controlling parameters b and a are made to vary by introducing the curvature constants K_1, K_2 and K_3. It is necessary now to monitor the signs of b and a to take remedial action, including changing the sign of x-step and y-step, if either of them threaten to become negative, for otherwise the loop will 'lock' onto one branch, with a continual sequence of x-step or y-step commands. With

repeat until finish
if $d < 0$ then { if $b < -0.5 \times K_2$ then { change sign: d,b,K_1,L_1,L_3,y_{step},
$K_1 = K_1 + L_2, K_3 = -K_3 - L_4$,
$b = b - K_2, d = d - a$ }
else { $x = x + x_{step}$,
$K_1 = K_1 + L_1, K_2 = K_2 + L_2, K_3 = K_3 + L_3$
$a = a + K_2, b = b - K_1, d = d + b$} }
else { if $a < 0.5 \times K_2$ then { change sign: d,a,K_3,L_2,L_4,x_{step}
$K_1 = -K_1 - L_1, K_3 = K_3 + L_3$,
$a = a + K_2, d = d + b$ }
else { $y = y + y_{step}$,
$K_1 = K_1 + L_2, K_2 = K_2 + L_3, K_3 = K_3 + L_4$,
$a = a + K_3, b = b - K_2, d = d - a$ } }

Figure 3. Cubic drawing algorithm (square move only)

appropriate remedial action, a change of overall direction into the neighbouring quadrant can be allowed for, enabling the drawing of a complete circle or ellipse (though there are problems in tracking sharply turning curves, in which the gradient changes appreciably within the distance of one pixel or increment [3,4]). Note that the constant K_2 is used in both branches of the algorithm, an important feature of the conic tracking analysis.

It was not until 1985, however, that Vaughan Pratt [5] finally solved the problem of adapting the conic drawing algorithm of Figure 2 to integer working, so that the exact control required for rendering font outlines can be achieved. His techniques are summarised in Section 2 of this paper. He also offers compelling reasons for preferring conic (quadratic) arcs to cubics in describing shape contours, a position in which he is joined by Pavlidis [6] and Hussain [7]. Since outlines of many characters involve the use of inflection points, the choice of cubic curve segments was 'natural' as no single conic section can accommodate a point of inflection. It turns out, however, that quadratic sections can be used to approximate points of inflections very effectively if they are joined together appropriately [5].

Much work has been done by John Hobby on analysing the process of rasterization for curves described in terms of a non-parametric form [8,9]. A thorough introduction to the topic can be found in his earlier work, where he also extends the concepts looked at originally by Pratt [5] for the conic case to cater for curves of any degree [8]. A detailed discussion is presented by Hobby on how to develop numerically stable techniques for converting rational parametric cubic forms to an implicit format [9]. The work focuses on an implicitization method which uses floating-point arithmetic both to generate and then to assess the overall conversion.

These developments are all recent, however, and PostScript, in which fonts are defined by four-point Bézier cubic arcs, has become something of a *de facto* industry standard. As it has a popular commercial audience, we address here the possibility of a more efficient cubic rasterizing algorithm, specifically a cubic extension of the conic drawing algorithm first published in 1968 [10]. The algorithm, which is shown in Figure 3, requires only six add

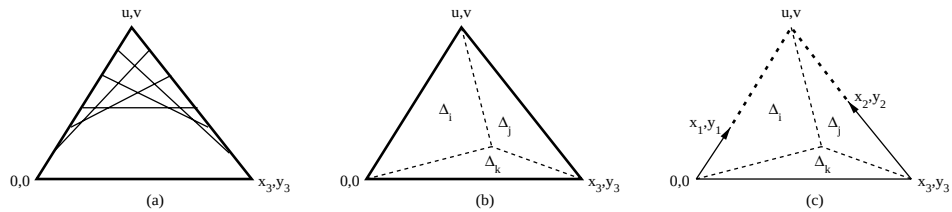

Figure 4. (a) Manual construction of a parabolic arc; (b) triangular set-up; and (c) Bézier formulation

operations (plus a branch and sector change tests) in the inner loop, so it is potentially very fast. The initial conditions for this algorithm, however, require that the cubic equation be given in a non-parametric form, with the ten necessary terms: $x^3, y^3, x^2y, xy^2, x^2, y^2, xy, x, y$ and a constant [11]. As the Bézier cubic spline is frequently defined in terms of a normalised parameter t (which ranges between 0 and 1, see Equation 5), it is necessary to perform an algebraic transformation to gain a non-parametric mathematical description; such a transformation is presented in Section 3. The behaviour and performance of this cubic algorithm is outlined in Section 4, where the spectacular breakdown of the algorithm is also analysed.

2 GENERAL CONIC SPLINES

As manifest in Figure 4a, a parabolic arc can be constructed between two end knots by using an additional control point gained with respect to the starting and ending knot gradients. The guiding triangle formed by such an arrangement is, therefore, enough for the complete description of a parabolic curve. For the sake of algebraic simplicity, the starting knot is positioned at the origin, the end knot at (x_3,y_3), and the control point at (u,v). The construction, which is well known to carpenters and joiners, involves dividing the line from the origin to the control point into a number of equally-spaced intervals, and the line from the control point to the end knot into the same number of equally-spaced intervals, and then joining them in sequence as shown in Figure 4a; the envelope of the straight lines forms a parabola (see Liming [12,13] for a complete analysis). Since the control point defines the starting and ending gradient, piecewise formulation of an outline can easily be gained with at least gradient continuity at the knots, the only constraint being that each knot lies on the straight line connecting adjacent control points. Mathematically, this parabola can be described in a parametric form by:

$$\begin{aligned} x_i &= 2t(1-t)u + t^2x_3, \\ y_i &= 2t(1-t)v + t^2y_3 \end{aligned} \tag{1}$$

where

x_i and y_i are parabolic curve points,
u and v denote the control position,
x_3 and y_3 the end knot position, and
t ranges between 0 (origin) and 1 (end knot).

This concept can be extended to yield any of the curves belonging to the conic family by introducing an additional parameter referred to as either ‘sharpness’ or ‘stiffness’, and

denoted by S. Using the workings of Vaughan Pratt [5], the parametric form for the general conic is given by:

$$\begin{aligned} x_i &= \frac{2t(1-t)Su + t^2x_3}{1+2t(1-t)(S-1)}, \\ y_i &= \frac{2t(1-t)Sv + t^2y_3}{1+2t(1-t)(S-1)}, \end{aligned} \tag{2}$$

where S can range from 0 to infinity.

For a given 'parabolic' triangle, any one of the conic arcs can be generated by varying the sharpness parameter S. When S equals zero the arc degenerates into the line connecting the two knot points, between zero and one it returns elliptic segments; at one a parabola results (Equation 2 reduces to Equation 1), and for values of S between one and infinity, hyperbolic arcs are realised. For very large values of S, the arc returns the two line segments associated with the control point. The curvature at the origin is given by

$$\frac{\Delta}{S^2c^3} \tag{3}$$

where

$$\begin{aligned} &\Delta = 0.5(vx_3 - uy_3), && \text{the area of the parabolic triangle and} \\ &c = \sqrt{u^2+v^2}, && \text{the line length from the origin to } (u,v). \end{aligned}$$

The curvature at the end knot is obtained by simply replacing c with a, where a is the length of the line from the control point (u,v) to this (end) knot [5]. Curvature-continuous piecewise conic representation can be achieved by choosing successive values of S so that the curvature at the end of one conic curve is matched to the curvature at the start of the next [5].

Although the parametric form for the conic is useful in some applications, the tracking algorithm works with a non-parametric (implicit) mathematical expression. This can be readily obtained by using the triangular breakdown of the parabolic triangle as depicted in Figure 4b, and by utilising Equation 2 to equate to zero the residue d_i for each data point (x_i,y_i):

$$d_i \equiv 4S^2\Delta_i\Delta_j - \Delta_k^2, \tag{4}$$

where

$$\begin{aligned} \Delta_i &= 0.5(vx_i - uy_i), \\ \Delta_k &= 0.5(y_ix_3 - x_iy_3), \\ \Delta_j &= \Delta - \Delta_i - \Delta_k. \end{aligned}$$

The tracking algorithm of Figure 2 operates by evaluating Equation 4 in a stepwise manner at the points on the corners between the pixels as they are selected, the sign of d_i indicating whether the point in question is situated above or below (or, if $d_i = 0$, on) the curve. This information is used to make either an x step or a y step, depending on which octant the conic curve resides. The algorithm can be made to work with integer arithmetic if the origin, control and end knot are all chosen to be integer mesh points, and if the

sharpness parameter S is expressed in terms of a rational form, scaling up Equation 4 by the square of the denominator and any factors of 2 required to avoid halves.

The implicit form of Equation 4 is also useful for estimating the quality of fit of a conic arc, in a 'least squares' sense, to a given set of n data points d_i, where $1 \leq i \leq n$, represents a font outline. The d_i can be formed directly, and their squares summed to yield a 'goodness of fit' residue [7]. Had the parametric form of Equation 2 been used, it would have being necessary to develop a relationship between the parameter t defining the conic points to each of the given data points (x_i,y_i), a much slower estimation.

3 BÉZIER CUBIC SPLINES

The Bézier cubic formulation uses two control points to generate a desired arc. The two points (x_1,y_1) and (x_2,y_2) are shown in Figure 4c. The gradients at the two knot points are governed by the positioning of the control points: the starting gradient is set by the point (x_1,y_1), whilst the finishing gradient is provided by the line connecting point (x_2,y_2) to the end knot (x_3,y_3). The mathematical description for the four-point Bézier is usually expressed through its parametric form:

$$\begin{aligned} x_i &= 3t(1-t)^2x_1 + 3t^2(1-t)x_2 + t^3x_3, \\ y_i &= 3t(1-t)^2y_1 + 3t^2(1-t)y_2 + t^3y_3. \end{aligned} \tag{5}$$

In order to aid simplicity and add geometric sense, it is convenient to retain the point (u,v) for the Bézier case. This, as depicted in Figure 4c, can easily be achieved by extending the control point lines so that the intersecting point becomes point (u,v), and by introducing parameters r and s (the proportional distances of the control points along the sides of the triangle) thus:

$$\begin{aligned} x_1 &= ru, \\ y_1 &= rv, \\ x_2 &= x_3 - s(x_3 - u), \\ y_2 &= y_3 - s(y_3 - v). \end{aligned} \tag{6}$$

Using the notation of Equation 6 to substitute in Equation 5, it is not difficult to show that the curvature at the origin of the Bézier arc is given by

$$\frac{4(1-s)\Delta}{3r^2c^3}, \tag{7}$$

where c is the line length from the origin to (u,v), as before.

Similarly, the curvature at the end knot is gained through replacing r for s and a for c (where a is again the length of line from (u,v) to (x_3,y_3)) in Equation 7.

The curvature at the respective knots can be matched to the curvature of a conic arc by equating Equation 7 with Equation 3. This shows that r and s need to be chosen such that

$$r = s = 2S\frac{\sqrt{S^2+3} - S}{3}. \tag{8}$$

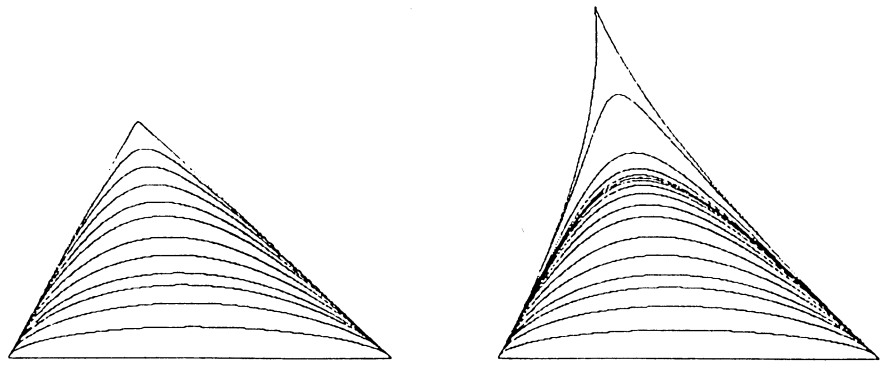

Figure 5. Effect on changing the sharpness S on: (a) the general conic; and (b) the Bézier cubic, with $r = s$ matching

Under these circumstances, it is not too difficult to formulate an implicit equation for the Bézier cubic, necessary for the rasterizing process:

$$d_i \equiv 9s\Delta[3s^2\Delta_i\Delta_j - (1-s)\Delta_k^2] + (2-3s)^2\Delta_k^3. \tag{9}$$

In comparison to the non-parametric form for the conic, Equation 4, the implicit form for the Bézier shows that the left-hand side of Equation 9 is simply d_i, scaled by a factor $9s(1-s)\Delta$, plus an additional term involving $(2-3s)^2\Delta_k^3$. This term becomes zero if $s = 2/3$. From Equation 8, it is clear that this is true for the parabolic case, where the sharpness parameter S equals one. The implicit form for the Bézier cubic can also describe the degenerative instance of the straight line extending from the origin to the end knot exactly: in the limit $s \to 0$, sharpness $S \to 0$, it sets $\Delta_k = 0$. Δ_k is relatively small for the case of an elliptic arc, where s ranges between 0 and 2/3. As s (and hence S) are made to increase, the curvature moves from the two end knots towards the centre of the curve. As this happens, the term Δ_k becomes more significant in Equation 9, and the resultant Bézier cubic fails to yield satisfactory hyperbolic arcs, as shown in Figure 5.

This shortcoming, and the limiting nature of the non-parametric form of Equation 9 (not being able to represent cubic curves containing inflection points, for example), leads to a new mathematical expression allowing the parameters r and s to take on individual values. To gain such an expression it is necessary to convert the non-rational parametric form of Equation 5 to an implicit format. This is achieved through using a similar approach to that described by Sederberg [14]. Through use of an algebraic manipulation package, it is possible to realise the following non-parametric form for the Bézier cubic spline:

$$9\Delta_i(3r - 3r^2 - s)gc_1 + 9\Delta_j(3s - 3s^2 - r)gc_2 + \Delta_k gc_3 = 0 \tag{10}$$

where

$$\begin{aligned} gc_1 &= 3s^2\Delta_i\Delta_j - (1-r)\Delta_k^2, \\ gc_2 &= 3r^2\Delta_i\Delta_j - (1-s)\Delta_k^2, \\ gc_3 &= 27[(r+s)(2s-r)(2r-s) + 2(r-s)^2 - 3r^2s^2]\Delta_i\Delta_j \\ &\quad -[9(r-1)(s-1) - 1]\Delta_k^2 \end{aligned}$$

Although the non-parametric expression of Equation 10 is in a form which can be used directly by the cubic rendering algorithm of Figure 3, it is always zero when the parameters $r = s = 2/3$, which is a nuisance. In the special case when $r = s$ the problem disappears, as a factor $(2 - 3s)$ drops out to give Equation 9, but in general the pole at $r = s = 2/3$ cannot be factored out. This leads to problems in modelling Bézier cubic splines to a given set of data points, a topic beyond the scope of this paper.

It is worth noting that curves containing an inflection point can be accommodated by Equation 10. The values for r and s simply take on opposite signs. In the case where the control point (u,v) cannot be defined (parallel start and end vectors), however, the triangular format cannot be constructed, and Equations 9 and 10 cannot be resolved. Though in the limiting case where for given values of r and s, the control point is moved towards infinity, it is seen that triangular areas Δ_i and Δ_j will become larger, whilst Δ_k will remain unchanged. With r and s also becoming smaller, the net effect would be that the changes will nullify each other, resulting in finite quantities for the two equations 9 and 10.

4 EXPERIMENTAL ANALYSIS

After some effort, a code was generated which facilitated the use of the Bézier cubic format for font outlines, as characterised in Figure 4c. The conversion from parametric to non-parametric form was undertaken by employing Equation 10, which resulted in the initial values for the coefficients needed for the cubic rendering algorithm of Figure 3.

For most cases (apart from those highlighted in Section 3), the computations proceeded efficiently, and it was not surprising to see that the cubic algorithm generated outlines of similar quality to that observed whilst rendering a quadratic arc using Pitteway's method (of Figure 2) or that seen through digitising a line segment through applying Bresenham's approach (of Figure 1). When the cubic algorithm was extended to cater for a complete (Latin-script) font, however, an occasional spectacular breakdown occurred; an example of this is depicted in Figure 6 for a seemingly innocent-looking arc. The given curve is shown in Figure 6a (where the curve points have been generated through using rounded values returned by Equation 5). Instead of tracking the arc as intended from the start to the end knot, the algorithm loses its way at point C, as shown in Figure 6b. If the start and end knots are reversed, the algorithm again gets 'confused' at point C and loops back as in Figure 6c. A closer examination of the cubic curve (Figure 6d) shows that this contains a self-intersecting point (point C, called a *crunode* in the field of mathematics). The rendering algorithm fails to track the given Bézier curve properly, and performs an unwanted quadrant change at point C, resulting in the two cases shown in Figures 6b and 6c.

The reason for this failure can be easily understood. It is well known that a general cubic equation in an unknown in x has either one or three real roots. It follows, therefore, that the cubic curve in x and y described by Equation 10 will cross any given straight line either once or three times. The relevant straight line here extends from the start knot and finishes at the end knot. The cubic is known to cross this line at both knot points, so there has to be a third crossing. This, as shown in Figure 6d, occurs at a relative distance t_c measured along the baseline, where

$$t_c = \frac{r^2(r + 3s^2 - 3s)}{(r - s)^2}. \tag{11}$$

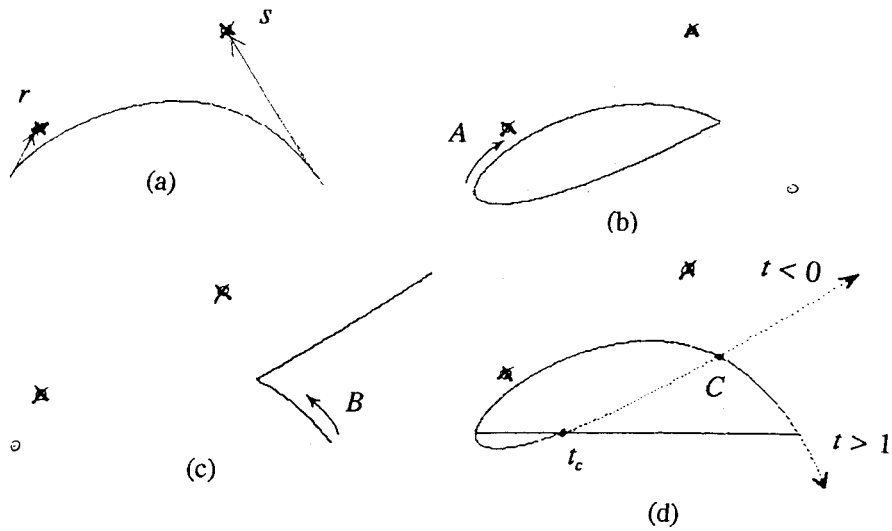

Figure 6. An example of how the tracking algorithm gets confused and subsequently loses control at point C: (a) the given arc; (b) following direction of A; (c) following direction of B; and (d) shows that at point C, the cubic self-intersects

In the case shown in Figure 6d, $r = 0.19444, s = 0.58333$, so $t_c = 0.34374$. Whenever we have t_c between zero and one, the third crossing of the baseline will lead to the confusion state of Figure 6. For a fix, we suggest that, before a PostScript (Type 1) font is released to a package using an implicit tracking algorithm like Figure 3, a scan is made of all the arcs evaluating t_c from Equation 11. If the value of t_c is less than zero or greater than one, the cubic intersecting point occurs outside the desired range and the respective arc can be rasterized using the tracking algorithm without any problem. If, on the other hand, the value of t_c indicates a state of confusion for the cubic algorithm, then the following remedies are available:

1. The first consists of making small adjustments to r and s until the third crossing point occurs outside the harmful range for t_c from one to zero. This approach, however, has two shortcomings; the first being that it is difficult to quantify 'small adjustments'; any changes to r and s will also change the shape of the resulting arc. The second difficulty is that this approach leads to an iterative (and thereby time-consuming) process for finding suitable values for the control parameters.
2. The second solution involves the employment of Bresenham's line drawing algorithm. This has the advantage that it is immune from the problem illustrated in Figure 6. The main disadvantage however is deciding on the number of line segments needed to represent an arc, and the fact that such a representation gives zero continuity between joining line segments, resulting in rasterized arcs which are not aesthetically acceptable.
3. The third, and probably the best, approach consists of subdividing the given Bézier curve at its self-intersecting point. By taking this point as a (new) knot point, the resulting two arcs will not contain any intersection points within the desired span, removing the confusion.

5 CONCLUSIONS

In this paper, we address the concern of digitising the outlines of fonts. It has highlighted the fact that it is easier to render such an outline through using modelling equations which are in a non-parametric form. As the Bézier cubic is usually expressed in terms of a parametric form, two expressions are presented for an implicit solution. The first constrains the Bézier control parameters r and s to be equal, which appears to be satisfactory for arcs which can be classified as either elliptic or parabolic. For the case of a hyperbolic curve, however, the implicit form for the Bézier cubic fails to match the curvature centred away from the start and end knots. As depicted in Figure 5b, for extreme hyperbolas the Bézier cubic first develops a cusp and then transforms into a curve with a loop, which is not helpful in matching an intended font outline.

The alternative implicit form is developed for the general case where $r \neq s$. The resulting form, unfortunately, has an annoying instability when $r = s = 2/3$; the expression degenerates to zero. This, as it happens, corresponds to the three-point Bézier curve, a parabola. This leads to problems in using this form for modelling outlines of shapes.

The non-parametric form required for efficient rendering cannot be used to distinguish between the intended cubic arc and the 'tail' of the curve, which can cause the tracking algorithm to get confused, as manifested in Figure 6. The tail results in a self-intersecting point for the cubic arc, the effect of which is analysed through Equation 11. Possible solutions are presented in Section 4, of which the prominent one is to locate and subdivide at the point of intersection, enabling the direct cubic tracking algorithm of Figure 3 to be utilised.

REFERENCES

1. Bresenham J. E., 'Algorithm for computer control of a digital plotter', *IBM Systems Journal*, **4**, 260–268, (1965).
2. Pitteway M. L. V., 'Algorithm for drawing ellipses or hyperbolae with a digital plotter', *Computer Journal*, **10**, 282–289, (1967).
3. Banissi E. K., *A Conic Drawing Algorithm with Grey Scale*, Ph.D. dissertation, Brunel University, UK, 1990.
4. Pitteway M. L. V., 'Algorithms of conic generation', *Fundamental Algorithms for Computer Graphics*, **17**, 219–238, (1985).
5. Pratt V., 'Techniques for conic splines', *Computer Graphics (ACM SIGGRAPH)*, **19**, 151–159, (1985).
6. Pavlidis T., 'Curve fitting with conic splines', *ACM Trans. on Graphics*, **2**, 1–31, (1983).
7. Hussain F., 'Conic rescue of Bézier founts', in *New Advances in Computer Graphics*, 97–120, Springer-Verlag, (1989).
8. Hobby J. D., 'Rasterization of nonparametric curves', *ACM Trans on Graphics*, **9**, 262–277, (1990).
9. Hobby J. D., 'Numerically stable implicitization of cubic curves', *ACM Trans on Graphics*, **10**, 255–296, (1991).
10. Botting R. J. and Pitteway M. L. V., 'Cubic extension of conic algorithm', *Computer Journal*, **11**, 120, (1968).
11. Hussain F. and Pitteway M. L. V., 'Rendering a cubic with square moves only', *Computer Journal*, **34**, 405, (1991).
12. Liming R. A., *Practical Analytical Geometry with Applications to Aircraft*, MacMillan Company, USA, 1944.
13. Liming R. A., *Mathematics for Computer Graphics*, Aero Publishers, USA, 1979.

14. Sederberg T. W., Anderson D. C., and Goldman R. N., 'Implicitization, inversion, and intersection of planar rational cubic curves', *Computer Vision, Graphics and Image Processing*, **31**, 89–102, (1985).