Pergamon

Comput. & Graphics, Vol. 19, No. 4, pp. 551–556, 1995
Copyright © 1995 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0097–8493/95 $9.50 + 0.00

0097-8493(95)00033-X

Technical Section

A NEW CONTOUR FILL ALGORITHM FOR OUTLINED CHARACTER IMAGE GENERATION

SHAO LEJUN and ZHOU HAO

School of EEE, Nanyang Technological University, Nanyang Avenue, Singapore 2263

Abstract—This paper presents a new algorithm to rasterize outlined characters for desk-top publishing applications. The font shape is described by its contours, cubic Bezier curves and/or straight lines. The algorithm is based on chain-code flags. It is fast, easy to implement and can guarantee the correct filling under any scale factors.

INTRODUCTION

Up to now, there are basically three methods to represent fonts in computer display or desk-top publishing applications. They are (1) bit-mapped representation; (2) vector representation; and (3) curved representation.

In bit-mapped representation, a font is defined as a matrix of dots, such as 15×16 dots. See Fig. 1 for an example of such a Chinese character. If we need to display or print a font of different size, say 24×24, the corresponding dot matrix for that font size is required. When bit-mapped fonts are used in Chinese or other Oriental computing systems, large storage space is needed to keep the dot matrix description of fonts for all styles in different sizes. For example, a typical Chinese font contains more than 7000 Chinese characters and a typical Chinese computing system contains at least four different styles of Chinese fonts. Many methods have been proposed to store the bit-mapped font in compressed form, but it is still impractical or impossible to store the bit-mapped representation of Chinese font of all sizes.

In vector representation, the font outline is represented by its polygon approximation. The data points indicating the vertices of the polygon are stored. This scheme is, however, not efficient. To approximate a piece of curve in high accuracy, many small vectors are needed.

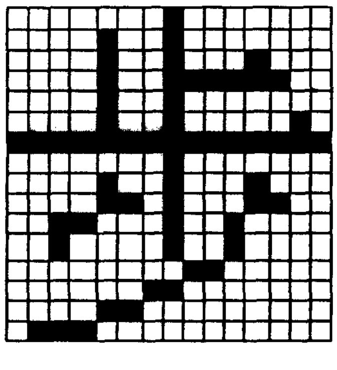


Fig. 1. A 15×16 bit-mapped Chinese character.

In curved representation, the font outline is described by a set of smoothly connected cubic curves. When the character is defined in this way, it is possible to generate different sizes and shapes of characters from the same curve descriptions. All we need is to specify scaling factors on x and y direction. The process of scaling a character is to carry out a coordinate mapping on the curve representations and to reconstruct a bitmap by using an area filling algorithm. Because of this advantage, the use of outlined font has become the trend in desk-top computing systems [2].

If text is formatted using outlined font, an efficient and robust algorithm has to be developed to convert the font described by its outlines into the bit-mapped form and fill it in the proper position of the frame buffer for later display or printing.

By robust we mean that the algorithm should rasterize font at any size with minimum distortion. In outlined font, one description is used to generate different sizes of font. Problems will appear when the algorithm needs to generate a small size font. The problems include broken strokes, over-filling, and distortion. When the font is scaled small enough, many strokes will become only one pixel wide. If the filling algorithm is not well designed, it may either over-fill or under-fill these strokes. Under-fill will result in broken stroke, over-fill will result in filling extra blank spaces in the frame buffer. The distortion may be the result of uneven stroke width for one stroke, merged strokes, *etc.*, caused by quantization.

In this paper, we will present such an algorithm which can avoid these problems. Before describing our algorithm, we will first give an overview of existing algorithms and examine their suitability for rasterizing font. A new font rasterizing algorithm, denoted "chain-code flags," will be proposed. This algorithm is fast, easy to implement and, most importantly, can rasterize the font correctly in any scaling factor (font size). Our algorithm can be divided into two parts. The first part is a new filling algorithm which is designed to avoid the over-fill and under-fill problems. The second part is hinting which is used to solve distortion problems. This paper will concentrate on the filling algorithm. The hinting algorithm will be described in a separate paper.

The algorithm has been successfully implemented in software and the hardware implementation is underway. Examples will be given to show the robustness of the algorithm. A discussion will be given to show how this algorithm can be combined with hinting information to produce high quality font in any font size.

REVIEW OF EXISTING FILLING ALGORITHMS

There are many filling algorithms in the literature. They can be divided into two broad categories: (1) seed fill; and (2) edge fill [9].

In seed fill algorithms [3, 5, 7, 10], a connected contour of the shape to be filled is first drawn onto the drawing buffer. A seed point, known to be in the interior of the shape, is then selected. The algorithm will then fill this pixel and all pixels that can be accessed from the seed pixel. The filling will continue until it reaches the contour points. The advantage of seed fill is that during the filling process it will not visit any pixel outside the contour. But seed fill presents many problems for our application. One drawback of seed fill is the need to supply or find an initial seed point for each contour. The seed point can be supplied as a part of the input to the algorithm together with all the contour data. But when the font size becomes small, one contour may split into two or more separate contours. The seed point supplied will fall in one of these contours and only that contour can be filled, thus resulting in under-fill. If all the seed points need to be found by the algorithm itself before filling takes place, the algorithm will become quite complicated.

Edge fill is also called scan conversion fill. In edge fill algorithms [1, 6, 9], the first step is also to draw the font's contour onto a drawing buffer. The next step is to scan the whole drawing buffer line by line and fill the interior of the contour. Within each horizontal scan line, the algorithm will fill segments between the first and second contour pixels, third and fourth contour pixels, *etc*. This is based on the fact that a straight line intersects any closed contour an even number of times. If the first point of the line lies outside the contour, then we can traverse it and decide which segments are in the interior by counting the number of intersections. All edge fill algorithms have to cope with special cases such as corner point, overlapped contours in order for the algorithms to work correctly all the time. Some hardware implementation based on edge fill has been reported [4]. Usually, edge fill algorithms have to scan the whole drawing buffer in order to fill correctly. In some cases, another buffer the size of which is the same as the drawing buffer is needed to help to avoid the broken strokes. This makes the algorithm less efficient.

In the following, we will present our new filling algorithm. It is based on chain-code flags. In addition to drawing buffer, an additional buffer is needed to store chain-code representation of the contours. The additional buffer will occupy much less memory than the drawing buffer, as can be seen in the following discussion.

During filling, the algorithm will scan and fill the drawing buffer line by line, but it will never scan outside the contour. It achieves this without the need to have a seed point to help the filling. The chain-code flags are used to avoid broken stroke and over-fill problems. That is, this algorithm combines the advantages of edge fill and seed fill and at the same time avoids the disadvantages of two algorithms, making it a very efficient and robust algorithm.

DESCRIPTION OF PROPOSED ALGORITHM

Bezier curves and outline font representation

There are several outline font description schemes. Our algorithm used cubic Bezier curve [8] to describe the font because this curve description scheme is used by PostScript, the most popular page description language adopted by many desk-top publishing and printing systems.

A parametric cubic curve segment is one for which the curve points are represented as a third-order polynomial of some parameter t. Because we deal with finite segments of a curve, we can limit the range of the parameter, without loss of generality, to $0 \le t \le 1$. For cubic Bezier curve segment, four points are used, as shown in Fig. 2. The curve equation expressed as a vector form is as follows:

$$P(t) = \sum_{i=0}^{3} P_i t^i (1-t)^{3-i} \quad 0 \le t \le 1.$$

Bezier curve has many important properties. Some of them are useful in implementing our algorithm and are given below:

1. P_0 and P_3 are the two endpoints of the curve segment. This can be seen if we set $t=0$ and $t=1$ respectively in the above equation.

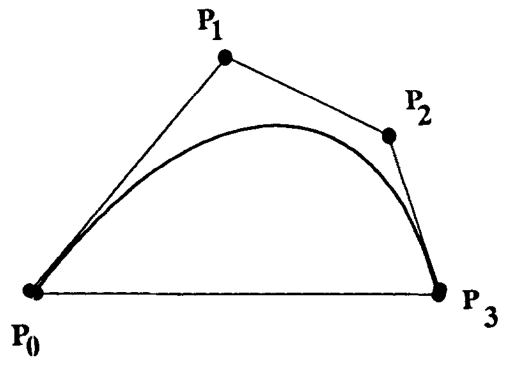


Fig. 2. A Bezier curve segment.

2. Straight line segments can be represented by Bezier curves. If a straight line is formed by connecting two points P_0 and P_3, any two points on the straight line P_0P_3 can be used as the two remaining control points P_1 and P_2.
3. The two inner control points P_1 and P_2 are located on the tangent vectors of the curve segment at the two corresponding endpoints. That is,

$$P_1 = P_0 + \Delta P_0 \cdot k_0$$

$$P_2 = P_3 + \Delta P_3 \cdot k_3$$

where k_0 and k_3 are scalars.
4. When a Bezier curve is divided into two segments at the middle point $t = 0.5$, each of them is again a Bezier curve and the new control points can be calculated by the following formulas (see Fig. 3):

$$\begin{cases} Q_0 = P_0 \\ Q_1 = \dfrac{P_0 + P_1}{2} \\ Q_2 = \dfrac{P_0 + 2P_1 + P_2}{4} \\ Q_3 = \dfrac{P_0 + 3P_1 + 3P_2 + P_3}{8} \end{cases}$$

$$\begin{cases} R_0 = Q_3 \\ R_1 = \dfrac{P_1 + 2P_2 + P_3}{4} \\ R_2 = \dfrac{P_2 + P_3}{2} \\ R_3 = P_3. \end{cases}$$

The property 1 tells us that if a curve is formed by joining several curve segments and the curve is closed, each curve segment can be expressed by using three data points. This is because the end-point of one curve segment is the starting-point of the following curve segment.

Property 2 can be used to check whether a piece of curve is a straight line. If the two control points are located on or very close to the line segment P_0P_3, this curve can be considered as a straight line and can be represented by two points instead of four points. Property 4 tells us a way of drawing Bezier curve on raster devices.

Unlike English characters, most Chinese characters consist of several disconnected components. The outline of each component may contain one or more closed contours. The character "chan," for example, consists of eight disconnected components (see Fig. 4). Among the eight components, five components (components 1, 2, 5, 7, 8) are represented by single closed contours, other three (components 3, 4, 6) are each represented by two closed contours. Each closed contour is described by several Bezier curve segments.

For Chinese or other Oriental characters, our algorithm will rasterize the font component by component. That is, the algorithm will read the complete curve description of one component, rasterize it before read and rasterize the next component. Component rasterizing within a font will be based on the top-down and left-to-right orders. In the example of character "chan," the rasterizing process will follow the order of component 1 to 8.

For each contour, the data points describing the Bezier curve segments will be inputted in the algorithm in a constant direction. The direction can be arbitrarily determined but must be consistent. For example, if the inner side of the closed contour is viewed in the left-hand side with respect to the direction (clockwise), the outer side will be in the right-hand side (counterclockwise).

To simplify our discussion, we will describe the rasterizing process of our algorithm for one component. It is assumed that a drawing buffer (frame buffer memory) will be available to store the rasterized bitmap font image.

The algorithm consists of three steps: (1) generate font outline on drawing buffer, generate chain-code description of the outline and chain-code flags; (2)

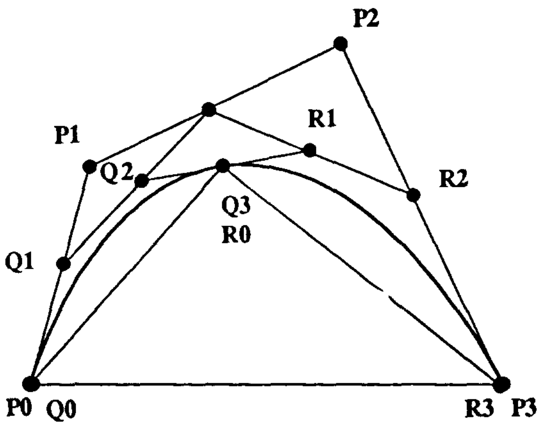


Fig. 3. Bezier curve decomposition.

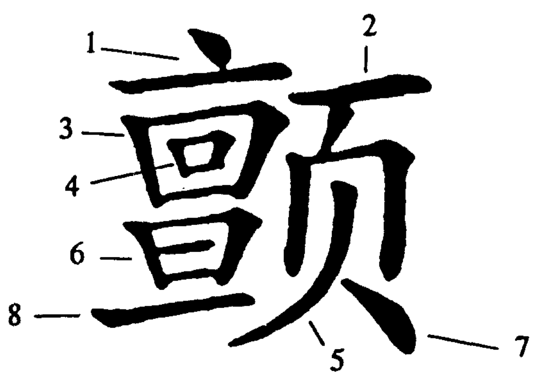


Fig. 4. Chinese character "chan" consists of 8 components and 11 contours.

add missing chain-code flags; and (3) fill inside the contour(s).

Generate outline, chain-code, and flag

The first step of our algorithm is to draw the font outline on the drawing buffer and at the same time generate the chain-code description of the outline and chain-code flags.

Chain-code is an encoding scheme used to represent a digital curve. In this scheme, the direction vectors between successive curve pixels are encoded. A chain-code commonly employs eight directions, which can be coded by 3-bit code words (see Fig. 5). Typically, the chain-code contains a starting pixel address followed by a string of code words. In our algorithm, four bits are used to represent one node, three bits for direction and one bit for flag. Chain-code flags are the indications of contour collision and used by the filling algorithm to avoid under-filling and over-filling. Contour collision would happen when the contour is scaled down enough so that different portions of the contour are mapped onto the same set of pixels due to data quantization.

In this step, the algorithm will draw Bezier curves on the drawing buffer by using the Bezier curve decomposition property, generate corresponding chain-code and set chain-code flags for collided outline pixels:

Algorithm Step 1

```
Clear the stack, initialize variables
For each closed contour Do
 Begin
 If the segment is reduced to a single pixel
  Begin
  Generate the chain-code for that pixel;
  If the associated pixel on drawing buffer is 0
   Begin
   Set flag for this chain-code to 0;
   Draw that pixel on drawing buffer;
   End
  Else Set the flag for this chain-code to 1;
  If the stack is empty
   Stop;
  Else Pop a stacked segment;
  End
 Else
  Begin
  Divide a segment into two;
  Push the second segment into the stack;
  End
 End
```

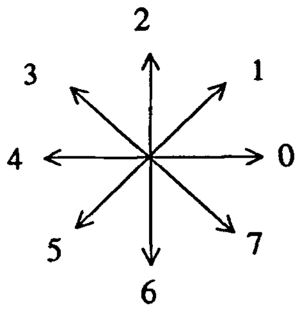


Fig. 5. Chain-code representation.

To hold chain-code values, an additional buffer is needed. Because the algorithm rasterizes the font component by component, the buffer size is determined by the maximum total length (total number of pixels) of contours describing one component among all components in the font. Suppose there are N components in a Chinese character, l_i represents the total number of contour pixels for component i and S is the required buffer size, we have $S=f(\max(l_i, i=1, \ldots N))$.

Before the implementation of the algorithm, the drawing buffer should be cleared. For each new generated outline pixel, a corresponding chain-code will be created. Whether the flag for that chain-code will be set to one is determined by its associated pixel value in the drawing buffer. If that pixel was already drawn on the drawing buffer, the flag will be set to one indicating a collision. Otherwise, the flag will be set to zero and the associated pixel in the drawing buffer will be drawn.

When this step is finished, the outlines of that font component are drawn on the drawing buffer, and the chain-codes together with its flags are also generated.

Add missing china-code flags

As we said in the previous section that contour collision is the result of different portions of contour mapped onto the same set of pixels due to quantization error. Each of such pixels will have two associated chain-codes. During the first step only the second chain-code flag was set, an additional step is needed to check backward and set the missed chain-code flags. This is to guarantee the correct filling. The pseudo code for this step is as follows:

Algorithm Step 2

```
For each closed contour Do
Begin
 Get the coordinates of the end point;
 Do
  Begin
  Compute the coordinates of the current pixel from chain-code;
  If the flag for this chain-code is 1
   Clear outline pixel (x, y) on drawing buffer;
  Else
  If the outline pixel (x, y) is 0
   Begin
   Set the flag of the associated chain-code to 1;
```

```
13      Set outline pixel (x, y) on drawing
          buffer;
14      End
15    Else
16    If the outline is going upward along
        vertical direction AND
17      the adjacent pixel (x+1, y) is drawing
          buffer is 1;
18      Set the flag of that associated chain-
          code to 1;
19    End
20  Until the current pixel reached the last
      one in the loop;
21 End
```

Since all contours are closed, the coordinates of the starting pixel will be the same as that of the end pixel. For each contour, choose that pixel as the starting point and traverse the chain-code backward to check the missing chain-code flags. For each chain-code, if its flag is already set, it indicates a contour collision. An additional chain-code exists, which is also mapped onto the associated collided pixel. We need to find out that chain-code and set its chain-code flag to one. To do that, the algorithm will reset that collided pixel to zero in the drawing buffer and continue the traversing process. Later on, whenever an outline pixel in the drawing buffer is found cleared, the corresponding chain-code flag will be set.

In this step, additional chain-code flags will be set if its corresponding contour is in a vertical direction and its interior region is only two-pixel wide. The additional flags will be used in Step 3 to avoid overfilling. Lines 16–18 in the pseudo codes are for this purpose.

Fill inside of the contour

Now all the collision pixels in the contour, if it exists, have been detected and marked by the chain-code flags. Additional flags have also been set to avoid overfill. The filling can start.

For each contour, the algorithm will traverse its chain-code and check its corresponding flag. If the flag is not set, the algorithm will find the location of the corresponding contour pixel in the drawing buffer and fill its interior along horizontal directions, either from left to right or from right to left, which is judged by the constant direction rule described before. The filling will continue along that direction until it meets a pixel which has already been filled.

During filling, the algorithm has to deal with an overfill problem. Our algorithm begins with an outline drawing by using the mathematical contour description. In mathematical terms, the contour has zero width whereas in an outline drawing the outline has a fixed width of 1 pixel. Consider a rectangle with coordinates (0,0),(3,2). An outline representation of the rectangle consists of four vectors (0,0),(3,0); (3,0),(3,2); (3,2),(0,2); and (0,2),(0,0). If this outline is to fill in the drawing buffer, the result is a rectangle

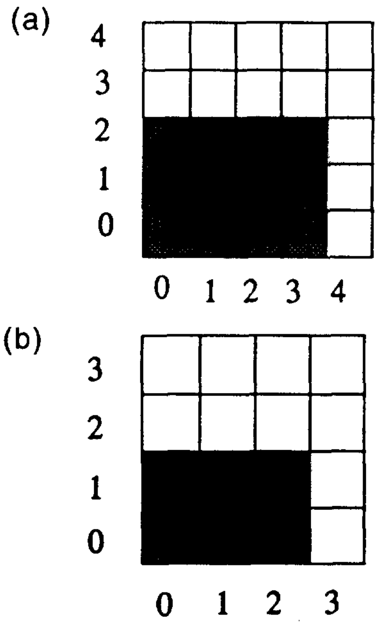


Fig. 6. (a) Conventional pixel representation. (b) Modified pixel representation.

with 12 square units (see Fig. 6a). In the mathematical terms, however, the rectangle has only 6 square units (Fig. 6b). This is the overfill problem, Ackland and Weste [1] gave a detailed discussion about this problem. Lines 11–13 in our following code are used to kill these overfilled pixels.

An outline of the filling algorithm is as follows:

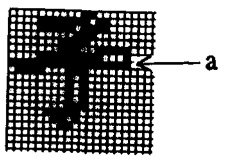
Algorithm Step 3

```
01 For each closed contour Do
02 Begin
03   Get the coordinates of the starting
       point;
04   Do
05     Begin
06     Locate the current contour pixel in
         drawing buffer from chain-code;
07     Decide the horizontal filling direc-
         tion;
08     If the flag for this chain-code is zero
09       Fill the drawing buffer from that
           contour pixel follow the
10       horizontal filling direction until a
           filled pixel is met
11     If (the contour is in vertical-down or
         in horizontal-left direction)
12       and its corresponding chain-code flag
           is zero
13       Reset that contour pixel to zero
14     End
15   Until the end point is reached.
16 End
```

Rasterizing a Chinese font

Now, let us use an example to illustrate the whole rasterizing process as illustrated in Fig. 7. The Chinese

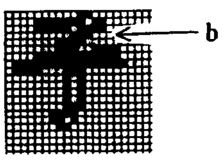
(a)



(b)

```
6,4,4,4,4,4,4,4,6,0,6,6,6,6,6,4,6,6,5,4,2,4,2,4,0,0*,0,2,2,2,2,2
2,2,4,4,4,4,6,4,4,4,2,4*,0,0*,0,0,2¹,0,0,0,0,2,4,2,0,6,2,0*
2,0,2,2,4,4,6,4,4,4,4,2,0,0,0,0,0*,0*,2*,0,7,0*,6,4,4,6,4,6*
4,6*,4,0,6*,0,0,0,2,0,0,0,0,6,0*
```

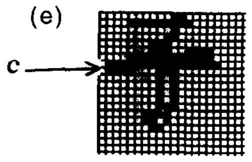
(c)



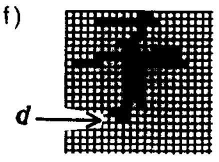
(d)

```
6,4,4,4,4,4,4,4,6,0,6,6,6,6,6,4,6,6,5,4,2*,4,2*,4*,0,0*,0,2*,2*,2,2,2
2,2*,4*,4,4,4,6*,4,4,4,2*,4*,0,0*,0,0,2¹,0,0,0,0,2*,4*,2*,0,6*,2*,0*
2*,0,2*,2*,4*,4*,6*,4,4,4,4,2*,0,0,0,0,0*,0*,2*,07,0,6,4,4,6,4,6*
4,6*,4*,0,6*,0,0,0,2,0,0,0,0,6,0*
```

(e)



(f)



(g)

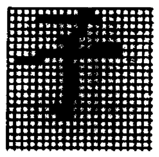


Fig. 7. Rasterizing of Chinese character "zi". (a) Outline drawn onto buffer. (b) Chain-code with flags marked. (c) Set additional chain-codeflags. (d) Filling up to position b. (e) Filling up to position c. (f) Filling up to position d. (g) Result.

character "zi" has an outlined description and is to be scaled-down and be rasterized into a 15 × 16 bitmap, the minimum font size allowed in Chinese font standard. Its outline uses a normalized 1000 × 1000 coordinate system and contains only one contour.

After the first step, the character's outline is drawn onto the drawing buffer (Fig. 7a) and the outline's chain-codes together with its flags are also generated (Fig. 7b). The chain-code starts from position *a* in clockwise direction as shown in Fig. 7a and the chain-codes with its flag set to one are marked by an *.

After the second step, additional chain-code flags are set as can be seen in Fig. 7c. The filling process is illustrated in Fig. 7d–g. The filling starts from position *a* in counterclockwise direction. Fig. 7d, e, f show the intermediate results of the filling process up to the points *b*, *c*, and *d* respectively. The final result is given in Fig. 7g.

The above mentioned Chinese font rasterizing algorithm has been implemented and tested on many Chinese characters with various scale factors. All give very good results. As mentioned earlier, to avoid distortion, hinting information has to be given. The result given in Fig. 7g does not use any hinting.

CONCLUSION

We have presented a new contour filling algorithm based on chain-code flags. This algorithm combined the advantages of seed fill and edge fill algorithms and at the same time avoided the disadvantages of these algorithms. It is very efficient and robust. This algorithm has been successfully implemented in software to rasterize Chinese and certain other Oriental fonts with arbitrary scale factors. No broken stroke and oversized stroke has ever been found during filling. Combined with our hinting algorithm, the system will be able to rasterize outlined Chinese or other Oriental font at any size with minimum distortion. Hardware implementation is underway.

REFERENCES

1. B. D. Ackland and N. H. Weste, The edge flag algorithm—A fill method for raster scan displays. *IEEE Trans. on Computers*, **30**, 41–48 (1981).
2. N. Asami and N. Inaba, Outline fonts to be standard OS function for PCs in Japan. *Nikkei Electronics ASIA*, 42–45 (1992).
3. K. P. Fishkin and B. A. Barsky, An analysis and algorithm for filling propagation. In *Proc. Graphics Interface '85*, Vancouver, Canada, 203–212 (1985).
4. N. Kai, T. Minagawa *et al.* A high speed outline font rasterizing LSI. In *Proc. of IEEE 1989 Custom Integrated Circuits Conference*, 24.6.1–24.6.4.
5. H. Lieberman, How to color in a coloring book. *Computer Graphics* **12**, 111–116 (1978).
6. W. D. Little and R. Heuft, An area shading graphics display system. *IEEE Trans. on Computers* **28**, 528–531 (1979).
7. P. S. Heckbert, *A Seed Fill Algorithm, Graphics Gems*, Academic Press, New York, 275–277 (1990).
8. W. M. Newman and R. F. Sprouli, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York (1979).
9. T. Pavlidis, *Algorithms for Graphics and Image Processing*, Springer-Verlag, New York (1982).
10. A. R. Smith, Tint Fill. In *SIGGRAPH'79 Proc.* 279–283, ACM, New York (1979).