ALGORITHMS FOR BRUSH MOVEMENT IN PAINT SYSTEMS

Kenneth P. Fishkin

Brian A. Barsky

Berkeley Computer Graphics Laboratory
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720
U.S.A.

ABSTRACT

Paint systems are an increasingly popular application of modern frame buffer technology. In such a program, an artist creates a brush that is moved across a frame buffer, providing a simplified simulation of a physical brush moving across an actual canvas. Movement of the brush often requires modification of a large number of pixels in a small amount of time. Existing algorithms for brush movement are discussed, and two new algorithms are presented that reduce the amount of i/o needed to move a brush, but at the expense of increased computational complexity.

KEYWORDS: Paint systems, Algorithms.

1. Introduction

The fundamental feature of an interactive "paint" system is the display of the brush on the frame buffer. Such a program typically works by detecting lines of motion from the user, often via a tablet and stylus. The brush then follows this line, "painting" over the existing pixels in the frame buffer.

The pixel i/o caused by this brush motion is often the crucial bottleneck of the paint program.[5,7,8] For example, consider a request for 100 points/second, with a brush of radius 10 pixels; even in this mild case, the program must "paint" over nearly 1 pixel per 30 nanoseconds. This computation can easily "choke" the paint program with pixel i/o requests, especially in the case of *non-opaque* brushes, where each pixel is read *and* written.

This problem can be approached in hardware by improving the speed of pixel i/o and host communication. This paper addresses the problem from a software standpoint, attempting to minimize the number of pixels altered, with no loss in image quality. Of course, the time taken to alter each pixel should also be minimized.

First, the formalism and notation for describing the brush and its motion is introduced. Next, three different algorithms are presented to address the problem. The first algorithm, a benchmark, performs no special action at all. The second algorithm, created and implemented as part of a paint program in 1982 at the University of Wisconsin by Steve Biedermann,[2] is presented for the first time. The third algorithm has been created by the authors.

This work was supported in part by the Semiconductor Research Corporation under grant number 82-11-008 and the National Science Foundation under grant number ECS-8204381.

1.1. Terminology

Many different data structures and formats for describing a brush and its effects exist; see Smith[7] for an excellent tutorial. In this presentation, one of the most common[2,5,7] and mathematically tractable structures is used:

A brush is described by a two-dimensional array, B, where B[i][j] determines the opacity of the brush in the i'th row and j'th column, $0 \le i < m$, $0 \le j < n$. The *opacity* values range between 0 and 1; 0 indicates perfect transparency and 1 indicates perfect opacity. This array is a function in three-space, mapping from the row-column plane into a height in the opacity dimension. The surface formed by this mapping will be termed the *opacity surface* of the brush.

To demonstrate the notation, consider the following algorithm (adapted from Smith[7]) to put the brush down with upper left corner at point (x,y):

```
for row := 0 to m - 1 do
    for col := 0 to n - 1 do
        Old := pixel( y + row, x + column );
            { for a monitor with (0,0) in upper left }
        new := brush_colour*B[row][column]
            + Old*(1 - B[row][column]);
    od;
od;
```

The brush shape is not constrained to be an m by n rectangle; this rectangular shape simply contains the bounding box of the brush shape. Two different brush shapes are discussed; one bounded by an m by n rectangle, and one bounded by a circle of radius r pixels.

We are interested in the motion of these brushes across the frame buffer, the shape "stamped" into the image by the motion of the brush across some line. For example, the shapes left by the circular and rectangular brushes are shown in Figure 1.1.

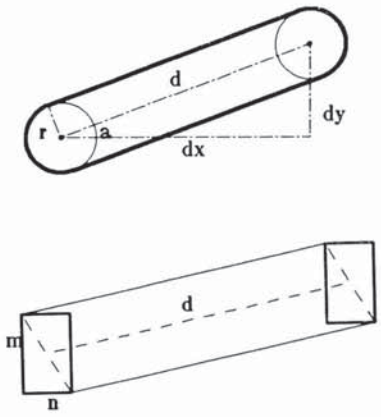

Figure 1.1.
Moving circular and rectangular brushes.

The brush is moved between the points (0,0) and (dx,dy), assuming dx $\ge$ dy. Any line can easily be transformed into this form by trivial scaling and reflections. The length of the line, $\sqrt{dx^2 + dy^2}$, is referred to as d. The angle of the line is denoted as α. Since $dx \ge dy$, $0 \le \alpha \le 45$ degrees.

1.2. Evaluation criteria

Three different criteria will be used to evaluate the brush algorithms:

- *Pixel i/o.* The *pixel area* is the number of pixels whose colour is changed by the motion of the brush. When the colour of a pixel is changed, we say that it has been *visited*. The total number of visits within the pixel area is the sum of the number of visits for each pixel in the pixel area. For example, if one pixel is written eight times, then the pixel area is one, but there have been a total of eight visits. For a given pixel area, the total number of visits should be minimized.
- *Computational expense.*
- *Image degradation.* In order to achieve gains in the first two criteria, algorithm may make approximations to the intensities of pixels, or even cover fewer pixels than are in the true area.

It is difficult to rank the relative importance of these criteria. The third, image degradation, may be the most important visually, but the question of "acceptable" degradation is a subjective one.

The total execution time of the algorithm is the product of the number of visits with the work done at each visit, but the two terms are not necessarily equal in importance. Pixel i/o is often the limiting bottleneck; greater computational cost may be acceptable in exchange for relatively minor decreases in the number of pixel visits. Furthermore, the time required for a pixel i/o operation varies greatly from system to system. The slower the pixel i/o is, compared to the computational speed, the more attractive any decrease becomes.

2. The Naive algorithm

For benchmark purposes, the first algorithm we present performs absolutely no special processing. This algorithm will be referred to as the *Naive* algorithm.

The Naive algorithm uses a Bresenham[3] -like algorithm to find the pixels on the *line of motion*, the line from (0,0) to (dx,dy). The brush is then put down centred at each such pixel, as shown in Figure 2.1.

```
Algorithm symmetric_Naive:
for each pixel on the line do
        put the brush down, centred at that pixel
od
```

The brush is put down $dx + 1$ times if the Bresenham algorithm is used, and each time πr^2 pixels are altered for a circular brush of radius r. Thus, the total number of visits is

$$\pi r^2 (dx + 1)$$

Since $dx \ge dy$, hence $d \le \sqrt{2}\,dx$, and therefore the number

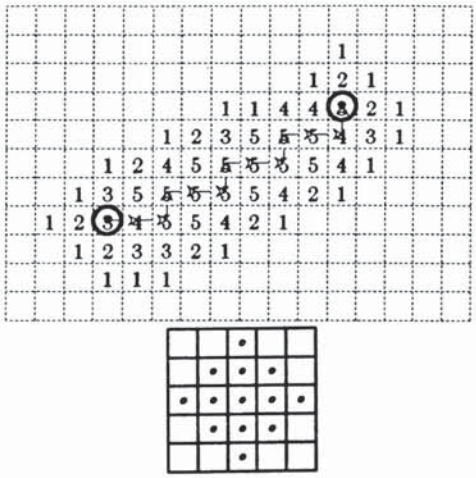

Figure 2.1.
Moving the brush with the Naive algorithm; the numbers indicate the pixel visits in each pixel.

of visits is $O(dr^2)$, that is, quadratic in r and linear in d.

The path of brush motion is determined by use of a Bresenham-like algorithm which is $O(d)$ in computational expense.

The Naive algorithm has two advantages: it works on all brushes, and uses an existing, very fast algorithm to determine which pixels to visit. Its singular disadvantage lies in the large number of wasteful pixel visits performed; the number of visits increases *quadratically* with r, while the pixel area (when $2dr >> \pi r^2$) increases *linearly* with r.

The Naive algorithm is trivially extended for asymmetric brushes. Since the algorithm makes no reference to the shape of the brush with which it draws, the change is a vacuous one:

Algorithm asymmetric_Naive:
for each pixel on the line **do**
put the brush down (mn pixels in size),
centred at that pixel
od

Using the performance evaluation for the Bresenham algorithm found in Field,[4] the Naive algorithm (as written in Foley & Van Dam[6]) is evaluated in Table 2.1, for a line from (0,0) to (dx,dy); this shows the visits to be $O(mnd)$, and the computational expense to be $O(d)$.

3. Biedermann's algorithm

The first work in minimizing pixel i/o known to the authors was done by Steve Biedermann at the University of Wisconsin in 1982.[2] His algorithm makes two strong assumptions about the brush being drawn, and with the power gained by those assumptions makes striking reductions in pixel i/o. The assumptions are:

Table 2.1: cost of the Naive Algorithm
for the asymmetric [symmetric] case
for a brush with m rows by n columns [radius r]

Operation	Times executed
:=	5 + dx
increment	dx + dy
shift left	2
+	dx
-	2
visits	$mn(dx + 1)\ [\ \pi r^2 (dx + 1)\ ]$

1) The brush must be opaque. The algorithm was implemented on a colour-mapped paint system, whose brushes generally possess this constraint[2, 1, 9] since interpolation and creation of new colour is very difficult for a colour-mapped picture.

2) The set of opaque pixels in the brush must form a solid, convex shape. The rationale for this assumption is explained below.

3.1. The symmetric case

In the easier case of a radially symmetric brush, the Biedermann algorithm "slices" the centre row (or column) of the brush, and paints with that one pixel-wide brush. This "centre slice" is, essentially, a bit vector describing the opacity of the path of the brush. When the brush is convex, solid, and symmetric, this slice describes the motion of the brush as a whole.

Algorithm symmetric Biedermann_Put
Put B at (0,0)
Put B at (dx,dy)
Let Center be the brush formed by the centre slice of B
Perform the Naive Algorithm between (0,0) and (dx,dy)
with Center as B

This algorithm is easy to implement. The existing Naive algorithm is still called, with a different brush, as shown in Figure 3.1.

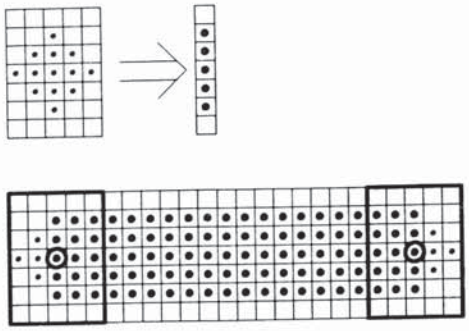

Figure 3.1.
An example of the symmetric Biedermann algorithm.

The need for the convexity requirement is demonstrated by

the brush in Figure 3.2. While the two slices quite properly mimic the behaviour of the brush *over a distance* for a non-convex brush, if the brush is only moved a short distance, an incorrect path is laid (Figure 3.2).

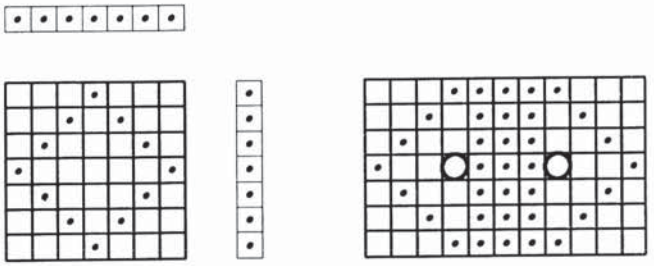

Figure 3.2.
A situation where Biedermann fails.

3.2. Visits by the symmetric Biedermann algorithm

The symmetric Biedermann algorithm makes an order of magnitude improvement over the Naive algorithm in pixel visits. The brush is laid down the same number of times (ignoring the endpoints), yet the brush size is now proportional to the *radius*, rather than the *area* of the brush. Since the endpoints are (for ease of implementation) used as the endpoints for the Center brush, slight overlap occurs, $2\pi r^2 + 2\,dx\,r$ pixels being visited. While the number of visits asymptotically approaches the number performed by the Naive algorithm as dx approaches 0, when $dx > \pi r$ it becomes $O(dr)$.

3.3. Image degradation of the Biedermann algorithm

The least desirable feature of the Biedermann algorithm is the slight but perceptible image degradation produced. The centre slice that is used to connect the two endpoints has, by construction, length equal to the diameter of the circular brush. Due to the tilt of the line of motion, the vertical cross-section of the brush path is actually slighter greater, as shown in Figure 3.3.

The percentage of degradation can be computed by finding the value of the variable x in Figure 3.3. Simple trigonometry shows that $x = r\sec\alpha$. The centre slice has a length of $2r$, as opposed to the correct length of $2x = 2r\sec\alpha$, causing image degradation.

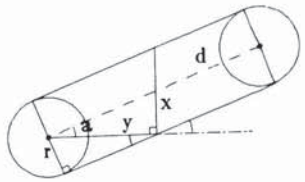

Figure 3.3.
Image degradation caused by Biedermann brushing.

The centre slice cannot be extended in length up to the correct length of $2r\sec\alpha$ for two reasons. First, the correct length is always non-integral (unless $\alpha = 0$), and the Naive algorithm (and brush algorithms in general) are defined only for brushes of integer dimension. Second, the stub must be computed for each α, and therefore on every line drawn.

Ignoring the endpoints, degradation is independent of the size of the brush. Degradation is a function solely of the tilt of the line, which varies from 0 to 45 degrees. The percentage of degradation at angle α is

$$1 - \frac{2r}{2r\sec\alpha} = 1 - \cos\alpha$$

In the best case, a horizontal line, $\cos\alpha = 1$, and no degradation is observed. In the worst case, a line at a 45 degree angle, $\cos\alpha = 1 / \sqrt{2}$, the degradation in the centre of the drawn line is .2929, over 29 percent. Assuming uniform distribution of α, the average degradation can be easily computed as

$$\frac{4}{\pi}\int_{\alpha=0}^{\frac{\pi}{4}} (1-\cos\alpha)\,d\alpha = .09968$$

In conclusion, the symmetric Biedermann method causes slight but perceptible image degradation. This degradation ranges from 0% to 29% across the centre of the brush line, and averages 10%.

3.4. The Biedermann algorithm for asymmetric brushes

For asymmetric brushes, two "slice arrays" are used instead of one. The first, the *horizontal slice*, represents the "trail" left by the brush when it moves horizontally. Each of the m entries represents the Boolean **or** of the opacity entries in that row. Similarly, the *vertical slice* represents the "trail" left by the brush when it moves vertically, each of the n entries representing the Boolean **or** of the opacity entries in that column.

The Bresenham line-drawing algorithm (used to determine the centre of the brush as it moves) makes a decision at every pixel; it either moves horizontally, or vertically. When the algorithm decides to move horizontally, the horizontal slice is put down. When the algorithm decides to move vertically, the vertical slice is put down (see Figure 3.4). The algorithm therefore makes $2\pi r^2 + mdx + ndy$ visits.

In conclusion, the Biedermann algorithms, both asymmetric and symmetric, are not perfect. Both contain cases where the brush shape displayed is incorrect. Both also make assumptions

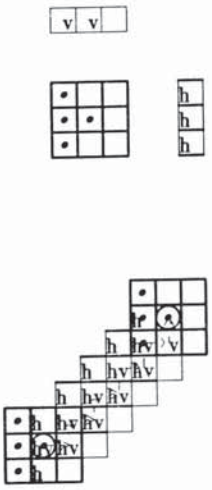

Figure 3.4.
An example of the asymmetric Biedermann algorithm.

about the convexity of the brush shape, and are only defined for opaque brushes. However, both reduce the pixel visits by an order of magnitude (compared to the Naive algorithm), with negligible increase in computational effort.

4. The Sweep algorithm

This section presents a new algorithm to render brush motion termed the *Sweep* algorithm. This algorithm maintains the order of magnitude reduction in pixel visits over the Naive algorithm, removes the degradation of the Biedermann algorithm, and makes no assumptions about the opacity of the brush. However, the algorithm requires much more computation than either the Naive or Biedermann algorithms.

As in the preceding sections, we now discuss two cases of increasing complexity. In the first case, exactly one assumption is made about the shape of the brush:

Assumption: the opacity surface formed by the brush is radially symmetric.

Section (4.4) discusses implementation of the algorithm without this assumption.

The algorithm is based on the creation of two arrays, termed *SweepLeft* and *SweepRight*. For a brush of size m by n, these arrays, both m by n in size, store the "rolling opacity" of the brush as it moves across the frame buffer. The j'th column of SweepLeft represents the cumulative effect of being visited by columns j, then j-1, ... then 0 of the brush B, as it moves from left to right. Similarly, the j'th column of SweepRight represents the cumulative effect of being visited by columns n-1, n - 2 ... then j of B.

The arrays are defined formally and recursively as follows:

```
SweepLeft[j][0] := B[j][0];
SweepLeft[i][j] := 1 - (1 - SweepLeft[i][j-1])*(1 - B[i][j]);

SweepRight[j][n-1] := B[j][n-1];
SweepRight[i][j] := 1 - (1 - SweepRight[i][j+ 1])
                       * (1 - B[i][j]);
```

The Sweep arrays "sweep over" the brush perpendicular to the direction of motion (see Figure 4.1). Due to the assumption of radial symmetry, a horizontal direction can always be used for the present case.

Original Brush, rotated perpendicular to direction of motion

		.1		
	.2	.2	.1	
.1	.3	.5	.2	.1
	.6	.4	.2	
	.2	.1		

left sweep

		.1	.1	.1
	.2	.36	.42	.42
.1	.37	.69	.75	.77
	.6	.76	.81	.81
	.2	.28	.28	.28

right sweep

.1	.1	.1		
.42	.42	.28	.1	
.77	.75	.64	.28	.1
.81	.81	.52	.2	
.28	.28	.1		

Figure 4.1.
An example of the two Sweep arrays.

For a radially symmetric brush, SweepLeft[i][j] equals SweepRight[n-1-i][j], by construction. The two arrays will still be maintained for clarity, and to ease the transition to the asymmetric case; the m by n notation is also retained for both cases to describe the bounding box of the brush shape. In the present, symmetric case, this bounding box is also assumed to bound a circular brush of r pixels in radius, $m = n = 2r$.

For a radially symmetric brush, the sweep arrays are

independent of the angle of the line, and therefore can be computed exactly once when the brush is instantiated.

4.1. The rendering

The Sweep algorithm works on a different, more expensive rendering basis than the previous two algorithms. Both the Naive and Biedermann algorithms used a Bresenham-like algorithm to compute the pixels hit by the centre of the brush. The brush was laid down centred at each such pixel, *approximating* the *actual* geometric path. The actual geometric path is slightly different; each row of the brush is being moved from the start of the line to the end of the line, where each row is one pixel away from the other perpendicular to the direction of motion. The Sweep algorithm uses this approach for the rendering. A simple preliminary version of the algorithm can be written as

Algorithm Sweep_First_Try
for each row in the brush **do**
    move from the lower left to the upper right, visiting pixels
od

This pseudo-code introduces three questions: Where does each row start and end? Which pixels are visited during each row traversal? How is the opacity of a visited pixel determined?

These questions are all answered by considering the path traversed by each row as a rectangle of one pixel width, oriented at a given angle. Rendering the rectangle is simply a special case of the well-known polygon scan conversion process (Figure 4.2).

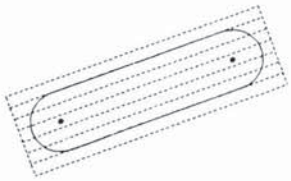

Figure 4.2.
Brush drawing by parallel rectangles.

The pixel opacity is determined by reference to the distance traversed along the path, a quantity generally known to polygon rendering algorithms. The two Sweep arrays are indexed by the distance the brush has been moved. For example, at the beginning, the columns from the SweepLeft array will be referenced, and at the end, those from the SweepRight. In the middle, it is irrelevant whether the last row of the SweepLeft array or the first row of the SweepRight array is referenced.

Since each rectangle is one pixel in width perpendicular to the line of motion, its vertical width will be (in general) *non-integer*, as will the starting and ending coordinates. For this reason, the path cannot be determined by a Bresenham-like algorithm.

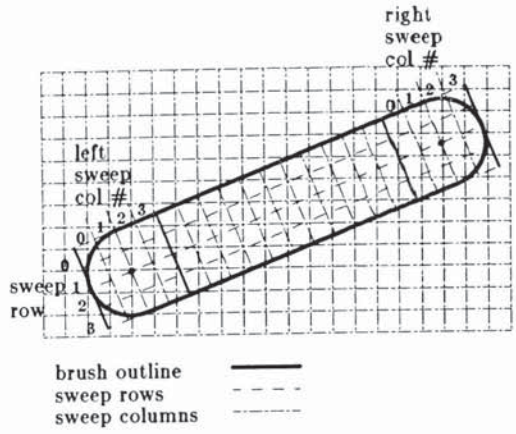

Figure 4.3.
Adding Sweep references to Figure 4.2.

4.2. Implementation

The rendering algorithm can be implemented in any number of ways. The pseudo-code for a very simple-minded implementation, designed to draw each row independently, is given at the end of this paper. This implementation incurs a significant speed disadvantage, making no use of rectangle coherence, but was chosen for readability and simple parallel implementation.

4.3. Properties of the symmetric Sweep algorithm

In theory, the Sweep algorithm visits every pixel needed exactly once. In practice, slight waste is incurred; the bounding box of the circle is visited for ease of implementation. Therefore, the number of pixel visits is

$$4r^2 + 2dr$$

Again ignoring endpoints (the $4r^2$ term), the number of visits is $O(dr)$. Due to the rendering of $2r$ rectangles, the computational expense is now $O(dr)$. This demonstrates the key tradeoff between the Naive and the Sweep algorithm; an order of magnitude reduction in visits for an order of magnitude increase in computation. The relative characteristics are summarized in Table 4.1.

Table 4.1: A summary of the algorithms' characteristics.

Algorithm	*Total Visits*	*Computation*	*Degradation?*
Naive	$O(dr^2)$	$O(d)$	*No*
Biedermann	$O(dr)$	$O(d)$	*Yes*
Sweep	$O(dr)$	$O(dr)$	*No*

4.4. The Sweep algorithm for asymmetric brushes

The Sweep algorithm presented to this point is only defined for radially symmetric brushes. While this is a significant assumption, it has been the experience of the authors and others[2,5,8,11] that this assumption generally holds. Extending the algorithm for the asymmetric case is straightforward, but

significant image degradation results. Note that the symmetric algorithmms are simply special cases of the asymmetric algorithms, for the case when $m = n = 2r$.

4.4.1. Rendering

Rendering is exactly the same for asymmetric brushes, once the Sweep arrays have been created. The implementation given in section 4 retained the m by n notation, and this notation can now be trivially applied to this case.

4.4.2. Computation of the Sweep arrays

Computation of the Sweep arrays for asymmetric brushes is difficult. The Sweeps are constructed by "sweeping" over the brush in a direction perpendicular to the line of motion. In the symmetric case, the arrays could be computed assuming the line was drawn horizontally; this allowed direct integer reference to the coordinates of the brush. In the asymmetric case, this assumption can no longer be made. The "sweep" now hits non-integer coordinates, making the computation of correct Sweep values very difficult (see Figure 4.4). This problem can be partially solved by rotating the brush and performing horizontal sweeping on it, as shown in Figure 4.5.

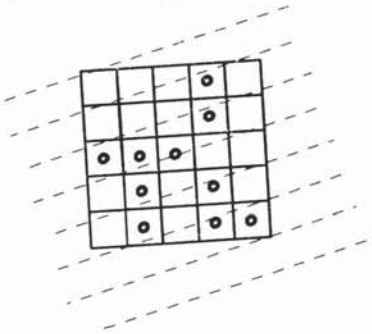

Figure 4.4.
Sweeping an asymmetric brush.

This solution has two disadvantages. First, the Sweep arrays must be recomputed *each time* a line is drawn. This expense alone could make the algorithm impractical. Second, the rotation procedure can introduce significant discretization errors for all but very special cases (rotations by multiples of 90 degrees).

5. Summary

Three algorithms have been presented for simulating the motion of a brush across a frame buffer. The first algorithm, the Naive algorithm, worked for all brushes, and performed $O(dr^2)$ (symmetric) and $O(dmn)$ (asymmetric) pixel visits, with $O(d)$ computational expense.

The second algorithm, the Biedermann algorithm, assumed an opaque, solid, convex brush. Within those limitations, it reduced the pixel visits to $O(dr)$ (symmetric) and $O(d(m+n))$ (asymmetric), and maintained the $O(d)$ computational expense,

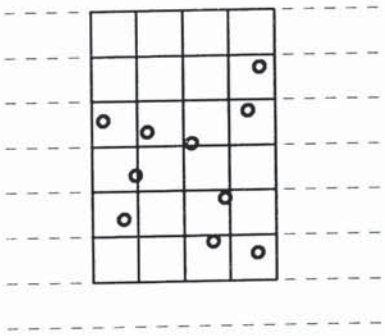

Figure 4.5.
Strobing a rotated copy; note the discretization problem.

However, some image degradation was incurred.

The third algorithm, the Sweep algorithm, made no assumptions about the shape of the brush, although the algorithm suffered severely in computational requirements and image quality if the opacity surface was not radially symmetric. The reduced number of pixel visits made by the Biedermann algorithm ($O(dr)$ (symmetric) and $O(m(d+n))$ (asymmetric)) was maintained, while the computational expense increased to $O(dr)$ (symmetric) and $O(dm)$ (asymmetric).

For each algorithm, pixel visit numbers are given in Table 5.1 for nine example cases. These cases represent the gamut of brush movements; a small, medium, and large brush is moved across a small, medium, and large distance.

Table 5.1: Sample total visits for the three algorithms

Algorithm			Naive	Biedermann	Sweep
r	dx	dy	$(dx+1)\pi r^2$	$2\pi r^2+2dxr$	$4r^2+2dr$
3	2	1	85	69	49
	12	3	368	129	110
	300	200	8,511	1,857	2,199
8	2	1	603	434	292
	12	3	2,614	594	314
	300	200	60,520	5,202	6,025
15	2	1	2,121	1,474	967
	12	3	9,189	1,774	1,271
	300	200	212,765	10,414	11,717

Table 5.1.

Several results are indicated by Table 5.1. The Naive algorithm is clearly and markedly inferior to the Biedermann algorithm when the assumptions of section 3 hold. The Sweep algorithm, while it makes roughly $2r(d - dx)$ more visits than the Biedermann, is still far better than the Naive algorithm.

As mentioned at the beginning of the paper, the number of pixel visits is not the sole criterion; the relative importance of visits, computational expense, and power of the algorithm is a subjective factor and makes a clear choice impossible.

6. Other algorithms

Two papers dealing with brush movement have been recently presented.

Tanner, Cowan, and Wein[8] have developed a brush-drawing algorithm termed *swath brushing*. This algorithm, developed independently of the *Sweep* algorithm mentioned above, contains the same basic principle: the brush is painted by the construction of a number of vectors parallel to the line. The swath algorithm implementation is incomplete,[10] making a more detailed comparison impossible at this time.

Whitted[11] has recently developed a related application of brush movement, a method for using brushes to draw multi-coloured anti-aliased lines. This method is tangential to the problem of brush sweeping, and so will only be referred to here in terms of the relevance to that problem:

1) As in the Sweep algorithm, endpoints are considered as a special case.
2) The problem of brush symmetry is also discussed briefly. The conclusion is reached that symmetric brushes are both common in practice and more tractable in computation.
3) The Naive algorithm is used to draw the line. The problem of re-drawing pixels is mentioned, with the conclusion that "for a software implementation this is wasteful, but it lends itself to fast execution in simple hardware".

Acknowledgements

The authors wish to thank Steve Biedermann for allowing them to present and evaluate his algorithms, and for his comments regarding the structure of the paper.

References

1. Aurora Systems, *The Aurora Paint System.* 1981.
2. Steve Biedermann, *The Superpaint Paint Program,* University of Wisconsin-Madison Graphics and Image Processing Laboratory, Madison, Wisconsin (1982).
3. J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal,* Vol. 4, No. 1, 1965, pp. 25-30.
4. Daniel E. Field, *Algorithms for Drawing Simple Geometric Objects on Raster Devices,* Ph.D. Thesis, Princeton University, Princeton, New Jersey (June, 1983).
5. Ken Fishkin, *Blot Users Manual,* University of California, Berkeley, California (1983). Berkeley Computer Graphics Laboratory internal document.
6. James D. Foley and Andries van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley Publishing Company (1982).
7. Alvy Ray Smith, *Painting Tutorial Notes,* Report No. 38, LucasFilm (1982).
8. Peter P. Tanner, William Cowan, and Marceli Wein, "Colour Selection, Swath Brushes and Memory Architectures for Paint Systems," pp. 171-180 in *Proceedings of Graphics Interface '83,* Edmonton(8-13 May 1983).
9. Via Video, *The Via Video Paint System.* 1982.
10. Marceli Wein, private communication. November 9, 1983.
11. J. Turner Whitted, "Anti-Aliased Line Drawing Using Brush Extrusion," pp. 151-156 in *Proceedings of SIGGRAPH '83,* (July, 1983).

Appendix: A pseudo-code implementation of the Sweep algorithm

```
/*
|     procedure  : SweepLine
|     description : draw a swept line from (0,0) to (dx,dy)
*/
proc SweepLine(in dx,dy : integer );
begin
   find (x,y) of lower left, upper left,
   lower right, and upper right edges of brush path;
   for each row of sweeps do
      DrawSweepLineFrag( LowerLeftPoint, UpperRightPoint,
            SweepLeft[row],SweepRight[row]);
      compute endpoints of next row
   od;
end procedure;
/*
|     procedure  : DrawSweepLineFrag
|     description : draw a fragment of a sweep line, from (x0,y0)
|                 : to (x1,y1), when both are real number pairs.
*/
proc DrawSweepLineFrag( in  x0, y0, x1, y1, : real;
      in LeftPtr,RightPtr : array of real );
begin
   walk over to first integer x coordinate;
   compute distances from left, right edges;
   for each x value between (x0) and (x1) do
      visit point ( x, floor(y) );
      { the opacities are determined by referencing the left and
         right distances, according to function FIND below }
      if ( floor( y - vertical width ) < floor(y) ) then
         visit point ( x, floor(y) - 1 );
      fi;
      update distances from left and right
   od;
end procedure;
function FIND takes a distance (fl) from the left edge and
a distance (fr) from the right edge returning the visiting opacity:
   if ( dr <= r ) then
      opacity = RightSweep[ max - fr ];
   else if ( fl <= r + r ) then
      opacity = LeftSweep [ fl ];
   else if ( fr <= r + r ) then
      /* can't combine with first clause, for short motions */
      opacity = RightSweep [ max - fr ];
   else
      opacity =  RightSweep [ 0 ];
   fi;
```