Report

Algorithms for drawing thick lines and curves on raster devices

Author(s):
Stamm, Beat

Publication Date:
1989

Permanent Link:

https://doi.org/10.3929/ethz-a-000505295 →

ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Beat Stamm

Algorithms for Drawing Thick Lines and Curves on Raster Devices

May 1989

107

Author's Address

Institut für Computersysteme
ETH–Zentrum
CH–8092 Zürich / Switzerland

Abstract

The present technical report describes algorithms for drawing lines of non trivial width on raster displays. Two models to extend standard Bresenham type algorithms to such lines are illustrated, the *geometric approach* and the *brush trajectory approach*. Particular results are efficient methods for drawing straight lines, polygons with mitered line joints and, for drawing lines with an arbitrary convex brush, an algorithm that reduces computational complexity from $O(n^2)$ to $O(n)$. The algorithms were implemented in Modula–2 and assembly language on a Ceres workstation [Eber87] and were found to be well suited for interactive applications such as a desktop publisher [Vett89], especially on the more recent Ceres–2 model [Heeb88].

Contents

1. Elementary Line Drawing

1.0. Lines

A well known algorithm for drawing straight lines on an integer grid is due to Jack E. Bresenham [Bres65] (many readers may be familiar with it, but since its basic methodology will be exploited in detail, a brief recall may be welcome). For simplicity the following paragraph will assume that a line

$$y_r := m \cdot x + q$$

is to be drawn from coordinates (x_1,y_1) to (x_2,y_2) in the first octant, i.e. with a slope

$$0 \le \frac{dy}{dx} \le 1; \quad dx := x_2 - x_1 > 0; \quad dy := y_2 - y_1 \ge 0.$$

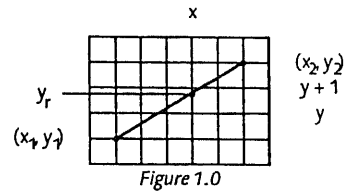

Figure 1.0

The key idea is to dispense with computing the line equation for each x and to work incrementally instead. Assume the line has been drawn up to some point (x,y) where y is the best possible approximation of the line equation with respect to the given grid (such a point always exists when using integral start/end points by putting $x := x_1$ and $y := y_1$). If x is now incremented by 1 then either y is still the best approximation or it also has to be incremented by one. The following program fragment illustrates the basic procedure.

```
PROCEDURE DrawLine(x1,y1,x2,y2: INTEGER);
  VAR x,y: INTEGER;
BEGIN
  Dot(x1,y1);
  x := x1; y := y1;
  WHILE x < x2 DO
    x := x + 1;
    IF yr doser to y + 1 than to y THEN y := y + 1 END;
    DisplayDot(x,y);
  END;
END DrawLine;
```

The above procedure assumes the existence of a subroutine *DisplayDot* for displaying individual dots on an integer grid.

In terms of Figure 1.0, the guard deciding whether or not to increment y can be formulated as

$$\text{"}y_r \text{ doser to } y+1 \text{ than to } y\text{"} \Leftrightarrow y + 1 - y_r < y_r - y \Leftrightarrow 2y + 1 < 2y_r.$$

Substituting the real line defined as

$$y_r := \frac{dy}{dx}(x - x_1) + y_1$$

into it, the guard becomes

$$\text{"}y_r \text{ doser to } y+1 \text{ than to } y\text{"} \Leftrightarrow 2y + 1 < 2\frac{dy}{dx}(x - x_1) + 2y_1.$$

By trivial algebra the quotient can be eliminated yielding

$$\text{"}y_r \text{ doser to } y+1 \text{ than to } y\text{"} \Leftrightarrow 2(x - x_1)\cdot dy - 2(y - y_1)\cdot dx - dx > 0.$$

Introducing d

$$d := 2(x - x_1)\cdot dy - 2(y - y_1)\cdot dx - dx$$

the guard is reduced to

$$\text{IF } d > 0 \text{ THEN } y := y + 1 \text{ END.}$$

Instead of evaluating d with each iteration, it is more efficiently computed incrementally. At the beginning, $x = x_1$, $y = y_1$ and $d := -dx$. With each iteration x increases by 1, hence

$$d := d + 2\cdot dy.$$

Similarly for y

$$d := d - 2\cdot dx.$$

In the procedure to follow one can see that the entire algorithm works with integral numbers and the operations required are add, subtract and multiply by 2 (a shift instruction) only.

```
PROCEDURE DrawLine(x1,y1,x2,y2: INTEGER);
  VAR x,y,d,dx,dy: INTEGER;
BEGIN
  DisplayDot(x1,y1);
  x := x1; y := y1; dx := x2 - x1; dy := y2 - y1; d := -dx { d ≤ 0 };
  WHILE x < y2 DO
    x := x + 1; d := d + 2*dy { d };
    IF d > 0 THEN y := y + 1; d - 2*dx { d ≤ 0 } END;
    DisplayDot(x,y);
  END;
END DrawLine;
```

To see that at the end of each iteration $d \leq 0$ consider the case where $d > 0$ after $x := x + 1$ (if $d \leq 0$ after $x := x + 1$ then there is nothing to prove). Since $dy \leq dx$ always and $d \leq 0$ at the start of each iteration, $d + 2{\cdot}dy - 2{\cdot}dx \leq d$ and thus $d \leq 0$.

The generalization for slopes > 1 is done by interchanging the roles of x and y. In case $dx < 0$ and/or $dy < 0$, x and/or y have to be decremented rather than incremented. The total of 8 cases can be reduced to 2 by replacing $x := x + 1$ and $y := y + 1$ by $x := x + inx$ and $y := y + iny$ respectively for variables inx and iny initialized appropriately.

1.1. Circles

It was again Bresenham [Bres77] who showed that also circles may be drawn using add, subtract and shift instructions only. This may appear quite surprising to the unbiased reader, since circular functions are usually defined in terms of square roots or trigonometric functions. But if only one octant of the circle

$$x_r := \sqrt{r^2 - y^2};\ y \geq 0$$

is considered (located at the origin), i.e. the 1st octant,

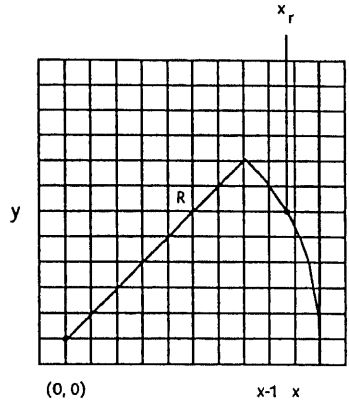

Figure 1.1

then

$$0 \leq \frac{dx}{dy} \leq 1$$

and the following procedure can be derived.

```
PROCEDURE DrawCircle(r: INTEGER);
  VAR x,y: INTEGER;
BEGIN
  DisplayDot(r,0);
  x := r; y := 0;
  WHILE y < x DO
    y := y + 1;
    IF x_r doser to x - 1 than to x THEN x := x - 1 END;
    DisplayDot(x,y);
  END;
END DrawCircle;
```

As can be seen in Figure 1.1, the italicized guard is

$$\text{"}x_r \textit{ doser to } x-1 \textit{ than to } x\text{"} \Leftrightarrow x_r - (x-1) \le x - x_r \Leftrightarrow 2\sqrt{r^2 - y^2} \le 2x - 1$$

Since $y \le r$ and r and y are integers, either

$$\sqrt{r^2 - y^2} = 0$$

or

$$\sqrt{r^2 - y^2} \ge 1,$$

thus by the monotonicity of the square root this is equivalent to

$$\text{"}x_r \textit{ doser to } x-1 \textit{ than to } x\text{"} \Leftrightarrow 4(r^2 - y^2) \le (2x-1)^2$$

It is again elementary algebra to transform it such that with

$$d := 1 - 4(r^2 - x^2 - y^2 + x)$$

the guard becomes

$$\text{IF } d \ge 0 \text{ THEN } x := x - 1 \text{ END.}$$

If d is evaluated incrementally, the squares need not be computed. At start, $x = r$, $y = 0$ and therefore

$$d := 1 - 4 \cdot r.$$

Then for each step

$$d := d + 8 \cdot y + 4; \quad y := y + 1$$

and eventually (guarded)

$$d := d - 8\cdot(x - 1);\quad x := x - 1$$

which leads to the following procedure.

```
PROCEDURE DrawCircle(r: INTEGER);
  VAR x,y,d: INTEGER;
BEGIN
  DisplayDot(r,0);
  x := r; y := 0; d := 1 - 4*r { d < 0 };
  WHILE y < x DO
    d := d + 8*y + 4; y := y + 1 { d };
    IF d ≥ 0 THEN x := x - 1; d := d - 8*x { (y < x) ∧ (d < 0) ⊕ (y ≥ x) } END;
    DisplayDot(x,y);
  END;
END DrawCircle;
```

To show that $d < 0$ at the end of each iteration is possible only if $y < x$ at that point (if $y \geq x$ then the guard for the iteration is false anyhow). Again the case to consider is where $d \geq 0$ after $y := y + 1$. Before such a case $d < 0$ and $y - 1 < x + 1$ or $y - 1 \leq x$ which means that $d + 8\cdot(y - 1) + 4 - 8\cdot(x + 1) \leq d + 8\cdot x + 4 - 8\cdot(x + 1) < d$ and therefore $d < 0$ again after the iteration.

The generalization to the other 7 octants can be done most easily by exploiting the eightfold symmetry of the circle, drawing eight dots at a time (interchanging x against y will do the 2nd octant, swapping the sign of x octants 3 and 4, and finally flipping the sign of y the remaining octants 5 through 8). Circles that are not located at the origin require each dot to be offset by the center coordinates of the circle.

Some care should be exercised at the junctions of the 8 octants when XORing the dots into the frame buffer. At these points exploiting the symmetry results in some of the dots being drawn twice, thus undrawing them again (this is always true for the angles $\alpha = 0$, $^1/_2\cdot\pi$, π and $^3/_2\cdot\pi$ and sometimes for the angles $\alpha = {}^1/_4\cdot\pi$, $^3/_4\cdot\pi$, $^5/_4\cdot\pi$ and $^7/_4\cdot\pi$, where the post condition of the iteration is either $y = x$ or $y > x$).

Sometimes it is desirable to draw a circle sequentially, rather than displaying dots in all the 8 octants at a time, e.g. when drawing circular arcs or when using the brush trajectory approach (to be discussed in chapter 4). For these situations 8 iterations have to be programmed, each being governed by an invariant of its own. They are in turn (where $\Delta := r^2 - x^2 - y^2$):

	Invariant	Action	Re-establish invariant
1.octant	$d := 1 - 4(\Delta + x) < 0$	$d := d + 8y + 4;\ y := y + 1$	$d := d - 8x + 8;\ x := x - 1$
2.octant	$d := 1 - 4(\Delta - y) \geq 0$	$d := d - 8x + 4;\ x := x - 1$	$d := d + 8y + 8;\ y := y + 1$
3.octant	$d := 1 - 4(\Delta + y) < 0$	$d := d - 8x + 4;\ x := x - 1$	$d := d - 8y + 8;\ y := y - 1$
4.octant	$d := 1 - 4(\Delta + x) \geq 0$	$d := d - 8y + 4;\ y := y - 1$	$d := d - 8x + 8;\ x := x - 1$
5.octant	$d := 1 - 4(\Delta - x) < 0$	$d := d - 8y + 4;\ y := y - 1$	$d := d + 8x + 8;\ x := x + 1$
6.octant	$d := 1 - 4(\Delta + y) \geq 0$	$d := d + 8x + 4;\ x := x + 1$	$d := d - 8y + 8;\ y := y - 1$
7.octant	$d := 1 - 4(\Delta - y) < 0$	$d := d + 8x + 4;\ x := x + 1$	$d := d + 8y + 8;\ y := y + 1$
8.octant	$d := 1 - 4(\Delta - x) \geq 0$	$d := d + 8y + 4;\ y := y + 1$	$d := d + 8x + 8;\ x := x + 1$

The sequential circle algorithm first establishes d_1 for x = r and y = 0. It then iterates while y < x under invariance of d_1. At that point it has done the 1st octant and now starts to do the 2nd octant. To do so it has to compute d_2, done easiest by

$$d_2 := d_1 + 4(x + y)$$

and proceeds iterating while x > 0 under invariance of $d_2 \geq 0$. Switching to the 3rd octant is done by

$$d_3 := d_2 - 8r \text{ or } d_3 := 1 - 4r, \text{ as for } d_1$$

since at that point x = 0 and y = r, and so forth for the remaining octants. This method of formulating d_{2i} in terms of d_{2i-1} has the advantage of never having to compute terms like r^2, x^2 or y^2, whereby the range of integer numbers may be exceeded.

For a circular arc for a given start point (x_s,y_s) the respective $d_i(x_s,y_s)$ has to be evaluated, however. The start point can be obtained out of the given start angle using (real–valued) trigonometric functions, provided rounding is performed such that for the corresponding d_i function the associated invariant is established. With the given end point (x_e,y_e) an analogous rounding choice has to be made. Inbetween iterations of the kind just illustrated are done, possibly comprising some integral octants. The arcs produced thereby consist of an exact subset of the points that a corresponding circle is made of.

In order not to duplicate program code, the drawing of circles and arcs can be implemented using procedure variables for the individual octants. For arcs the only differences are the initial values of x,y and d and the termination condition, to be passed as a parameter to these procedures.

1.2. Ellipses

In elementary analytical geometry ellipses are defined by

$$\frac{x^2}{r_1^2} + \frac{y^2}{r_2^2} = 1$$

which describes an ellipse located at the origin with halfaxes r_1 and r_2 parallel to the coordinate axes. Solving for x gives

$$x_r = r_1 \cdot \sqrt{1 - \frac{y^2}{r_2^2}}$$

Again in the 1st octant

$$0 \leq \frac{dx}{dy} \leq 1$$

and for an incremental algorithm

$$\text{"}x_r \text{ doser to } x\text{-}1 \text{ than to } x\text{"} \Leftrightarrow 2r_1 \cdot \sqrt{1 - \frac{y^2}{r_2^2}} \le 2x - 1$$

By algebra this can be transformed such that with

$$d := 1 - 4\left(r_1^2 - \frac{r_1^2}{r_2^2} \cdot y^2 - x^2 + x\right)$$

the guard becomes again

IF $d \ge 0$ THEN $x := x - 1$ END;

Notice that by setting $r_1 = r$ and $r_2 = r_1$ this reduces to the analogous case for circles. Since this decision is qualitative but not quantitative, either side of the relation can be multiplied by r_2^2 and the division vanishes.

$$\text{"}x_r \text{ doser to } x\text{-}1 \text{ than to } x\text{"} \Leftrightarrow 2r_1 \cdot \sqrt{r_2^2 - y^2} \le r_2 \cdot (2x - 1)$$

and

$$d := r_2^2 - 4\left(r_1^2 \cdot r_2^2 - r_1^2 \cdot y^2 - r_2^2 \cdot x^2 + r_2^2 \cdot x\right)$$

For the incremental evaluation of d at start $x = r_1, y = 0$ and $d = r_2^2 \cdot (1 - 4 \cdot r_1)$ and for each step

$$d := d + r_1^2 \cdot (8 \cdot y + 4); \quad y := y + 1$$

and eventually (guarded)

$$d := d - r_2^2 \cdot 8 \cdot (x - 1); \quad x := x - 1$$

The implementation from that point on is straightforward and is therefore not elaborated. Notice that with laser printers x, y, r_1 and r_2 may assume values larger than 2^{10}, however, thus e.g. $r_1^2 \cdot (8 \cdot y + 4)$ may exceed 2^{32}. This either demands 64–bit integer arithmetic or, to cure the symptoms, the use of long reals (IEEE 64–bit reals use 52 bits for their mantissa, they can be used instead of integers unless the magnitudes of the numbers to be represented exceed 2^{52}).

Being obliged to use real arithmetic anyhow suggests to immediately move to general ellipses, i.e. ellipses whose halfaxes are not aligned with the coordinate axes. Such ellipses require an extra parameter, the angle φ by which the larger halfaxis is rotated counter–clockwise with respect to the x–axis. They can be obtained out of

$$\frac{x_1^2}{r_1^2} + \frac{x_2^2}{r_2^2} = 1$$

if the ellipse is formulated using a quadratic form. With

$$x := \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}; \quad Q := \begin{pmatrix} \frac{1}{r_1^2} & 0 \\ 0 & \frac{1}{r_2^2} \end{pmatrix}$$

this becomes

$$x^T \cdot Q \cdot x = 1$$

where x^T denotes the transpose of x. If the coordinates x are now rotated by φ, i.e.

$$x' := R \cdot x$$

where

$$R := \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}$$

then the equation of the ellipse becomes

$$x'^T \cdot Q' \cdot x' = 1$$

where

$$Q' := R \cdot Q \cdot R^T$$

since

$$\forall R \in SO(2): R^T \cdot R = 1$$

Using the original coordinates x

$$x^T \cdot Q' \cdot x = 1$$

describes the desired ellipse. Doing the required algebra

$$Q' = \begin{pmatrix} \sigma & \delta \\ \delta & \bar{\sigma} \end{pmatrix}$$

where

$$\sigma := \frac{\cos^2\varphi}{r_1^2} + \frac{\sin^2\varphi}{r_2^2};\quad \overline{\sigma} := \frac{\sin^2\varphi}{r_1^2} + \frac{\cos^2\varphi}{r_2^2};\quad \delta := \sin\varphi\cdot\cos\varphi\cdot\left(\frac{1}{r_1^2} - \frac{1}{r_2^2}\right).$$

Thus the ellipse finally becomes

$$\sigma\cdot x_1^2 + 2\delta\cdot x_1 x_2 + \overline{\sigma}\cdot x_2^2 = 1.$$

The next problem to solve is to find the points on the ellipse where

$$\frac{dy}{dx} = \pm\infty, \pm 1, 0$$

since between two such (consecutive) points either $|dx| \le 1$ or $|dy| \le 1$ and an incremental algorithm analogous to the one for circles in the 1st octant can be formulated. Defining

$$F(x,y) := \sigma\cdot x^2 + 2\delta\cdot xy + \overline{\sigma}\cdot y^2$$

the total differential is

$$dF = \frac{\partial F}{\partial x}\cdot dx + \frac{\partial F}{\partial y}\cdot dy = (2\sigma\cdot x + 2\delta\cdot y)\cdot dx + (2\delta\cdot x + 2\overline{\sigma}\cdot y)\cdot dy = 0.$$

Thus for $dx = 0,\ dy \neq 0$ (or $\frac{dy}{dx} = \pm\infty$)

$$dF = 0 \Leftrightarrow (2\delta\cdot x + 2\overline{\sigma}\cdot y) = 0 \Leftrightarrow y = -\frac{\delta}{\overline{\sigma}}\cdot x$$

$$F\left(x, -\frac{\delta}{\overline{\sigma}}\cdot x\right) = x^2\cdot\left(\sigma - \frac{\delta^2}{\overline{\sigma}}\right) = 1$$

$$x = \pm\sqrt{\frac{\overline{\sigma}}{\sigma\cdot\overline{\sigma} - \delta^2}}.$$

For $dx \neq 0,\ dy = 0$ (or $\frac{dy}{dx} = 0$)

$$dF = 0 \Leftrightarrow (2\sigma\cdot x + 2\delta\cdot y) = 0 \Leftrightarrow x = -\frac{\delta}{\sigma}\cdot y$$

$$F\left(-\frac{\delta}{\sigma}\cdot y, y\right) = y^2\cdot\left(\overline{\sigma} - \frac{\delta^2}{\sigma}\right) = 1$$

$$y = \pm\sqrt{\frac{\sigma}{\sigma\cdot\bar{\sigma} - \delta^2}}.$$

And finally for $dx = \pm dy$ (or $\frac{dy}{dx} = \pm 1$)

$$dF = 0 \Leftrightarrow (2\sigma\cdot x + 2\delta\cdot y) = \mp(2\delta\cdot x + 2\bar{\sigma}\cdot y) \Leftrightarrow y = \mp\frac{\sigma \pm \delta}{\bar{\sigma} \pm \delta}\cdot x =: \mp\kappa_\pm\cdot x$$

$$F(x, \mp\kappa_\pm\cdot x) = x^2\cdot(\sigma \mp 2\delta\cdot\kappa_\pm + \bar{\sigma}\cdot\kappa_\pm^2) = 1$$

$$x = \pm\sqrt{\frac{1}{\sigma \mp 2\delta\cdot\kappa_\pm + \bar{\sigma}\cdot\kappa_\pm^2}}.$$

Notice that for circles ($r_1 = r_2 = r$, $\varphi = 0$, $\sigma = \bar{\sigma} = \frac{1}{r^2}$, $\delta = 0$ and $\kappa_\pm = 1$) these points are in turn

$$x = \pm r;\ y = \pm r \quad \text{and} \quad x = \pm\frac{r}{\sqrt{2}}$$

as expected. What is left to do is to work out appropriate functions d such that conditions similiar to

```
IF d ≥ 0 THEN x := x – 1 END;
```

can be used and to work out their rates of change in case x or y changes by ±1. Starting at the point that corresponds to $x = r, y = 0$ for circles and proceeding counter-clockwise, these are in turn (where $\Delta := 1 - \sigma x^2 - \bar{\sigma} y^2$)

	Invariant	Action	Re-establish invariant
1.octant	$d := \sigma - 4(\Delta + \sigma x - \delta y(2x-1)) < 0$	$d := d + \bar{\sigma}(8y+4) + \delta(8x-4)$	$d := d - 8(\sigma(x-1) + \delta y)$
2.octant	$d := \bar{\sigma} - 4(\Delta - \sigma y - \delta x(2y+1)) \geq 0$	$d := d - \sigma(8x-4) - \delta(8y+4)$	$d := d + 8(\bar{\sigma}(y+1) + \delta x)$
3.octant	$d := \bar{\sigma} - 4(\Delta + \sigma y - \delta x(2y-1)) < 0$	$d := d - \sigma(8x-4) - \delta(8y-4)$	$d := d - 8(\bar{\sigma}(y-1) + \delta x)$
4.octant	$d := \sigma - 4(\Delta + \sigma x - \delta y(2x-1)) \geq 0$	$d := d - \bar{\sigma}(8y-4) - \delta(8x-4)$	$d := d - 8(\sigma(x-1) + \delta y)$
5.octant	$d := \sigma - 4(\Delta - \sigma x - \delta y(2x+1)) < 0$	$d := d - \bar{\sigma}(8y-4) - \delta(8x+4)$	$d := d + 8(\sigma(x+1) + \delta y)$
6.octant	$d := \bar{\sigma} - 4(\Delta + \sigma y - \delta x(2y-1)) \geq 0$	$d := d + \sigma(8x+4) + \delta(8y-4)$	$d := d - 8(\bar{\sigma}(y-1) + \delta x)$
7.octant	$d := \bar{\sigma} - 4(\Delta - \sigma y - \delta x(2y+1)) < 0$	$d := d + \sigma(8x+4) + \delta(8y+4)$	$d := d + 8(\bar{\sigma}(y+1) + \delta x)$
8.octant	$d := \sigma - 4(\Delta - \sigma x - \delta y(2x+1)) \geq 0$	$d := d + \bar{\sigma}(8y+4) + \delta(8x+4)$	$d := d + 8(\sigma(x+1) + \delta y)$

The increments and decrements for both x and y have been left out, since they are the same as in the analogous case for circles.

The implementation should take care of a numerical problem. On laser printers coordinates and radii may assume values of order 2^{10}, hence x^2 or y^2 may be of order 2^{20}, together with σ, $\bar{\sigma}$ and δ of order 2^{-20}. Since the decisions are qualitative, appropriate scaling by a factor s can reduce this kind of problem.

$$\sigma' := s\cdot\sigma \quad \overline{\sigma}' := s\cdot\overline{\sigma} \quad \delta' := s\cdot\delta \quad \Delta' := s - \sigma' x^2 - \overline{\sigma} y^2$$

As with circles also elliptical arcs are best implemented using procedure variables for each of the eight sections.

The present section has discussed how to extend an incremental line drawing algorithm to ellipses. One problem has been carefully omitted though. The algorithm always decides between either two vertical or two horizontal neighbours on an integer grid (a so-called two point algorithm). But that decides for the point with minimal vertical or horizontal distance, and not for the one with minimal orthogonal distance. For straight lines the equivalence of both decision rules is obvious.

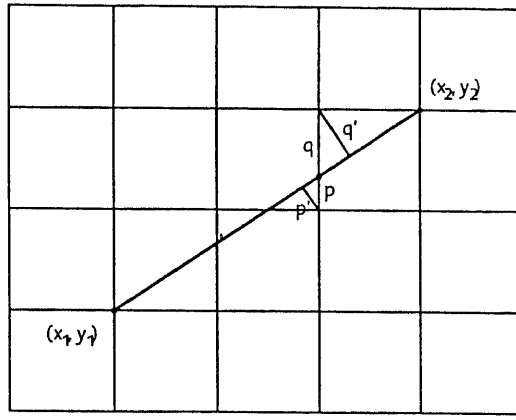

Figure 1.2

In terms of Figure 1.2, the decision whether or not to increment y is based upon $q < p$. The triangles formed by q, q' and the real line and by p, p' and the real line respectively are similar. Therefore

$$\frac{p}{p'} = \frac{q}{q'}$$

and deciding upon $q < p$ is equivalent to deciding upon $q' < p'$, which minimizes the orthogonal distance.

For circles the argument is not as simple as that. Although also for circles the quantities p, q, p' and q' can be defined accordingly, but this time

$$\frac{p}{p'} \neq \frac{q}{q'}$$

except for special cases where the real circle passes through a grid point. The proof has been done by Bresenham [Bres77] and is therefore not repeated here.

For ellipses an analogous proof was dispensed with for the following reasons:

1. The error must be smaller than one pixel (otherwise the two–point method would have decided for a different pixel in the first place). This can eventually be perceived for ellipses whose major axis is much longer than the minor axis.

2. In view of lines having to be drawn considerably wider than one pixel only (c.f. the following chapters), point 1 becomes irrelevant.

3. Since the algorithm has to be programmed with real arithmetic (coefficients like σ, $\bar{\sigma}$ and δ are obtained through trigonometric functions) a proof would have to take the underlying (machine dependent) implementation of real numbers into account, e.g. for tests like $d = 0$ or similar, which is not acceptable.

1.3. Splines

Given a set of n points in the Euclidian plane, then n–1 third order polynomials can be fitted to them such that

1. the resulting curve is continuous
2. the first derivative is continuous
3. and the average of the second derivative is minimal.

Such curves are called splines. The third requirement then corresponds to minimal deformation energy to be invested in the fitting process. It is in general a topic of numerical mathematics to show, how the polynomials are obtained out of the control points and this section will therefore not come back to this. Rather, it will discuss how to draw third order polynomials themselves.

Such a polynomial can be written as

$$\vec{\gamma}(s) = \vec{a}\cdot s^3 + \vec{b}\cdot s^2 + \vec{c}\cdot s + \vec{d}$$

in parametric form. They are best evaluated using forward differences.

$$\Delta^{(0)}(s) := \vec{\gamma}(s) = \vec{a}\cdot s^3 + \vec{b}\cdot s^2 + \vec{c}\cdot s + \vec{d}$$

$$\Delta^{(1)}(s) := \Delta^{(0)}(s+1) - \Delta^{(0)}(s) = 3\vec{a}\cdot s^2 + (3\vec{a} + 2\vec{b})\cdot s + \vec{a} + \vec{b} + \vec{c}$$

$$\Delta^{(2)}(s) := \Delta^{(1)}(s+1) - \Delta^{(1)}(s) = 6\vec{a}\cdot s + 6\vec{a} + 2\vec{b}$$

$$\Delta^{(3)}(s) := \Delta^{(2)}(s+1) - \Delta^{(2)}(s) = 6\vec{a}$$

As expected, the third order forward differences are independent of s. The control points themselves are used to start with. At such a control point $s = 0$ and the above formulae are simple to evaluate. Then for each step the lower ordered difference is obtained out of the higher ordered by simple addition.

$$\Delta^{(i-1)}(s+1) := \Delta^{(i-1)}(s) + \Delta^{(i)}(s)$$

For third order forward differences and two coordinates this yields 6 additions per iteration. The drawing algorithm is sketched out in the following program fragment.

```
PROCEDURE DrawSpline(px,py: Polynomial);
  VAR x,dx1,dx2,dx3,y,dy1,dy2,dy3: REAL;
BEGIN
  dx3 := 6*px,a; dx2 := 6*px,a + 2*px,b; dx1 := px,a + px,b + px,c; x := px,d;
  dy3 := 6*py,a; dy2 := 6*py,a + 2*py,b; dy1 := py,a + py,b + py,c; y := py,d;
  DisplayDot(x,y);
  WHILE not completed the polynomial DO
    dx2 := dx2 + dx3; dx1 := dx1 + dx2; x := x + dx1;
    dy2 := dy2 + dy3; dy1 := dy1 + dy2; y := y + dy1;
    DisplayDot(x,y);
  END;
END DrawSpline;
```

Some points of the above informal sketch require a closer look. First, for the resulting lines to become continuous with respect to the given integer grid, both dx_1 and dy_1 have to be less or equal to 1 in magnitude. The continuity properties of the splines imply that this can be achieved by selecting an appropriate interval for which to form the forward difference,

$$\exists\, ds > 0\ \vec{\gamma}: \mathbb{R} \supset [0,1] \to \mathbb{R},\ s :\to \vec{\gamma}(s), \vec{\gamma}(0) = P_s, \vec{\gamma}(1) = P_e \text{ and}$$
$$\forall s: \Delta_{x,y}^{(1,ds)}(s) := |\gamma_{x,y}(s + ds) - \gamma_{x,y}(s)| \leq 1$$

where P_s and P_e denote the start and end point of the polynomial. In this case the above formulae are subject to modification in order to cover for $ds \neq 1$.

Another approach is to scale the parameter interval such that the restriction on the magnitude of $\Delta^{(1)}$ holds for $ds = 1$ in the first place.

$$\exists\, S > 0\ \vec{\gamma}: \mathbb{R} \supset [0,S] \to \mathbb{R},\ s :\to \vec{\gamma}(s), \vec{\gamma}(0) = P_s, \vec{\gamma}(S) = P_e \text{ and}$$
$$\forall s: \Delta_{x,y}^{(1)}(s) := |\gamma_{x,y}(s + 1) - \gamma_{x,y}(s)| \leq 1$$

The scaling problem is again left to the numerical part obtaining the polynomials out of the control points in a first step and is therefore not pursued any further here.

What is important to notice, however, is the fact that the polynomials' coefficients necessarily have to be real valued, for their computation requires divisions. Hence x and y have to be rounded to the nearest integer before being used for displaying a dot. It is then possible to define a reasonnable execution condition for the iteration.

```
PROCEDURE DrawSpline(px,py: Polynomial; Ps,Pe: Point);
  VAR x,dx1,dx2,dx3,y,dy1,dy2,dy3: REAL;
      xi,yi: INTEGER;
BEGIN
  dx3 := 6*px,a; dx2 := 6*px,a + 2*px,b; dx1 := px,a + px,b + px,c; x := px,d; xi := Ps,x;
  dy3 := 6*py,a; dy2 := 6*py,a + 2*py,b; dy1 := py,a + py,b + py,c; y := py,d; yi := Ps,y;
  DisplayDot(xi,yi);
  WHILE (xi # Pe,x) OR (yi # Pe,y) DO
    dx2 := dx2 + dx3; dx1 := dx1 + dx2; x := x + dx1; xi := Round(x);
```

```
    dy2 := dy2 + dy3; dy1 := dy1 + dy2; y := y + dy1; yi := Round(y);
    DisplayDot(xi,yi);
  END;
END DrawSpline;
```

It remains to mention that the preceeding procedure actually has to iterate over all polynomials that the spline is made of.

At that point the question may be risen why not to use forward differences for lines, circles and ellipses, too. Recall the line algorithm. There for each step

$$d := d + 2*dy$$

and eventually (guarded)

$$d := d - 2*dx.$$

Since during the iteration both dy and dx are constant and not used for anything else than re–establishing the invariant, they can be replaced.

$$dx' := 2*dx; \quad dy' := 2*dy$$

The resulting algorithm operating with dx' and dy' then uses first order forward differences. For circles second order forward differences are expected to be constant. To illustrate the point, the circle algorithm for the first octant is given next.

```
PROCEDURE DrawCircle(r: INTEGER);
  VAR x,y,dx,dy,d: INTEGER;
BEGIN
  DisplayDot(r,0);
  x := r; y := 0; dx := 8*(x - 1); dy := 8*y + 4; d := 1 - 4*r;
  WHILE y < x DO
    d := d + dy; dy := dy + 8; y := y + 1;
    IF d ≥ 0 THEN d := d - dx; dx := dx - 8; x := x - 1 END;
    DisplayDot(x,y);
  END;
END DrawCircle;
```

During the iteration the circle algorithm too requires nothing but additions and subtractions. With little imagination the generalization to ellipses should be feasible. But unless special hardware permitting the direct addressing of individual pixels is used, the DisplayDot operation may be a lot more expensive than any gain obtained thereof. This is especially true for lines that are considerably wider than one pixel only, which will be discussed in the following chapters.

2. Advanced Line Drawing Models

Modern typesetting programs are often equipped with sophisticated graphical illustration capabilities. Even the simplest stock exchange index diagram for example requires the drawing of straight lines, which are not necessarily horizontal or vertical, as is sketched out in Figure 2.0. If the diagram is to be printed on a medium resolution (300 pixels/inch) laser printer and the lines are only 1 typographical point ($^1/_{72}$ inch) wide, then the associated raster operation effectively has to draw a 4 pixels wide line.

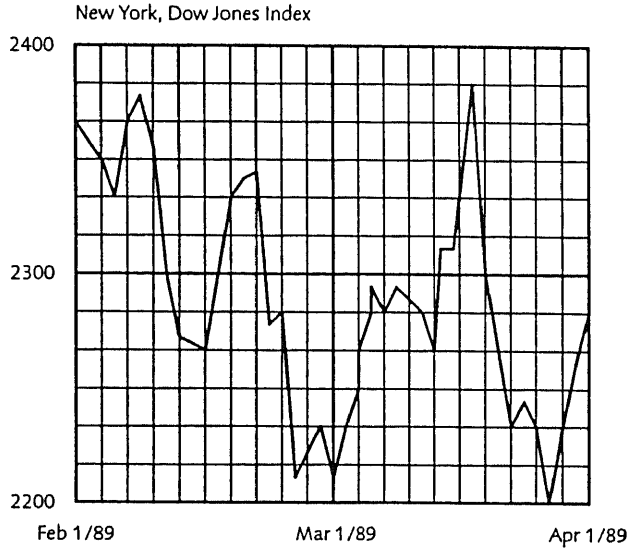

Figure 2.0

The present chapter will introduce two approaches for modeling wide lines. Algorithms for their efficient implementation follow in the subsequent chapters 3 and 4.

The first model to describe wide lines is to take the narrow lines from chapter 1 and simply make them "wider", i.e. symmetrically add "paint" on either side of it. This may be thought of as a transition from 1-dimensional to 2-dimensional objects, as illustrated in Figure 2.1 (Strictly speaking also chapter 1 discusses 2-dimensional objects, which are all composed of picture elements of small but not neglectable dimensions. Careless programmers disregarding this "academical" distinction are frequently lead into what is referred to as the "±1-paranoia". For the present informal introduction it is unimportant, however).

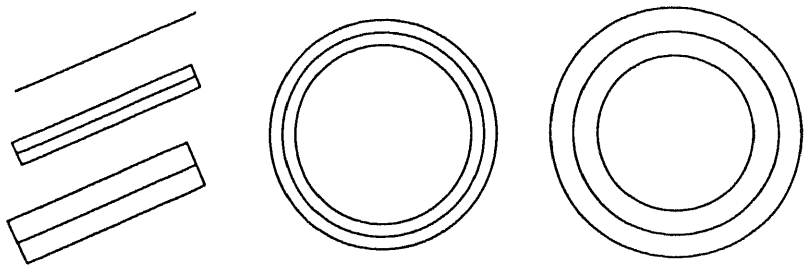

Figure 2.1

Widening narrow lines has the effect of turning e.g. a straight line into a thin rectangle or a circle into an annulus. These objects are fully described by their geometry, such as the width, the height and the orientation of the rectangle or the inner and the outer radius of the annulus. The underlying model is therefore called the *Geometric Approach.*

Moving into the second dimension adds a degree in complexity, despite the promising simplicity of this model. Consider the case where two or more lines are linked together, in order to draw the polygonial curve of Figure 2.0. The area which makes up for the junction of two such lines does not consist of a single pixel anymore, since these lines are now wide. Therefore the way in which they join has to be precisely specified. The following Figure 2.2 gives the outlines of a variety of such line junctions.

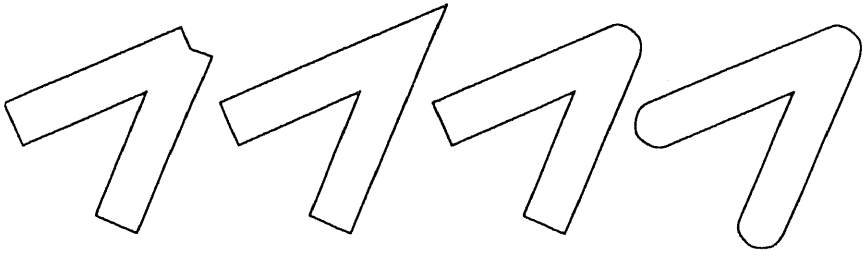

Figure 2.2

The leftmost example shows a junction of two narrow lines that have been widened *after* joining them. This leaves an undesirable notch inbetween them. A common way to overcome this kind of aesthetical disadvantage is to conceptually extend the outer edges upto the point where they intersect and form what is called a *mitered* line joint. It is much like the woodwork done by a carpenter upon joining two planks of a piece of furniture or so, as shown in the second example from the left. Acute angles yield vertices lying far away from the original point of intersection, compared to the width of the line. Commercially available software therefore allows to specify a limit beyond which mitering should be replaced by a *beveled* line joint, which is simply a mitered joint with its peak cut off at some point. Less sharply cornered results are obtained through *round* line joints, as in the third example.

The rightmost example finally also takes the line ends into account, here by appending a circular line cap.

A creative reader might think of further joints and caps, but the fact that so many things have to be added to the model sooner or later gives raise to doubt in its usefulness, let alone any implementation thereof. The second model for drawing wide lines therefore puts the emphasis on *drawing* the line rather than obtaining a *wide* line. Drawing is an incremental process whereby e.g. a pen, a pencil or a painter's brush is applied to paper or canvas and subsequently steered along some trajectory. Depending on the size and the shape of the drawing instrument a trail of color is left behind. The underlying model is called the *Brush Trajectory Approach.*

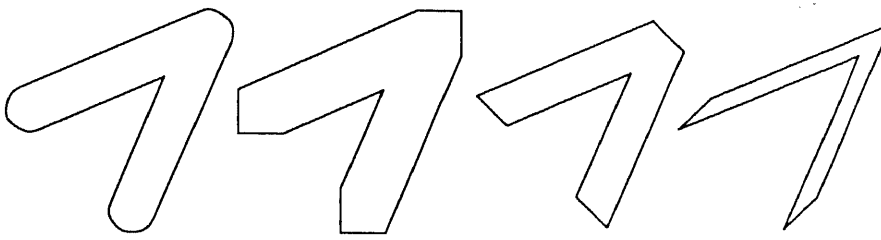

Figure 2.3

The leftmost example in Figure 2.3 uses a round brush whose diameter is the same as the width of the lines in the preceeding Figure 2.2. The resulting images are exactly the same. A square shaped brush yields the second example of Figure 2.3, while a thin pen held at different angles the remaining two.

The amount by which the brush trajectory approach appears more flexible than the geometric approach suggests to attempt to formulate the latter in terms of the former. One could for instance compute the shape and orientation of a thin pen held orthogonally with respect to the trajectory in order to obtain the straight line of the geometrical approach. Circles could then be done with a round brush of the right diameter. But already for a polygon with round line joints and straight line ends matters would become more troublesome. Unless the raster to which to apply the brush is actually a film exposed to light or an ordinary video memory to which the pixels to be drawn are inverted, one could in such a case switch between different brush shapes during the drawing process itself (for the two exceptions mentioned it is important not to draw a single pixel twice, however). The same thing holds true for mitered line joints. At the points where the switching actually takes place furthermore continuity of the resulting contours would have to be ensured, for quantization effects could result in e.g. a broad nibbed pen held at some angle having a width differing from its round brush companion by one pixel or so.

The converse, namely to understand the brush trajectory approach as a subset of the geometric approach appeared quite hopeless at that point and was therefore immediately dispensed with. It looked as if one could not have one's cake and eat it, too. Or both approaches have to be pursued further independently. For this reason the following chapter 3 first gives some algorithms for the geometric approach for efficiently drawing lines, miters and circles. Hints eventually given in conjunction may give an idea which way to follow to implement all kinds of line joints and caps. Chapter 4 then supplies an algorithm for drawing lines with an arbitrary convex, simply connected brush and suggests extensions to multiply connected brush shapes with concavities.

3. The Geometric Approach

3.0. Lines

A straight line of uniform width may be thought of as a thin rectangle. Once its vertices are determined out of the start and end point and the width of the line, some kind of contour-tracing algorithm may be used to fill the rectangle. An elegant and most general filling algorithm can be found in [Kohe88]. If this is customized to the present simple situation, this leads to an efficient line drawing algorithm.

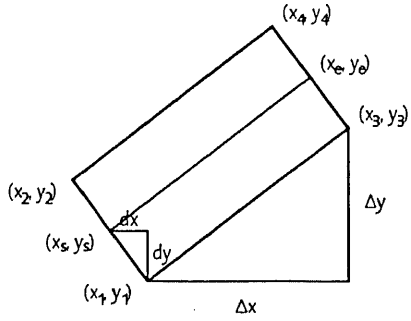

Figure 3.0

From the above Figure 3.0 the following formulae for computing the vertex (x_1, y_1) in terms of the start (x_s, y_s) and end point (x_e, y_e) and the width w of the line are straightforward.

$$x_1 := x_s + dx; \quad dx := \frac{w \cdot \Delta y}{2\sqrt{\Delta x^2 + \Delta y^2}}; \quad \Delta y := y_e - y_s$$

$$y_1 := y_s - dy; \quad dy := \frac{w \cdot \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}}; \quad \Delta x := x_e - x_s$$

The other vertices can be obtained in an analogous manner.

The main challenge here, however, is to find a method for their evaluation with integer arithmetic only. To start with, a version of Newton's iteration is used to compute the square root.

$$x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}; \quad f(x) := x^2 - a; \quad f'(x) := \frac{d}{dx} f(x) = 2x$$

With the initial value $x_0 := a$ this is known to converge to $x_N = \sqrt{a}$ after a few iterations only. If the integer operations are done in appropriate order, x_N is the largest integer less than or equal to $\sqrt{a}$.

Since this truncation is not always acceptable for selecting a point on an integer grid, rounding may be necessary. This can be done by applying the first binomial law to the result. If

$$x_N + 0.5 \leq \sqrt{a}$$

$$(x_N + 0.5)^2 = x_N^2 + x_N + 0.25 \leq a$$

$$4x_N \cdot (x_N + 1) + 1 \leq 4a$$

then x_N has to be increased by 1. Here is the algorithm:

```
PROCEDURE Sqrt(a: INTEGER) : INTEGER;
  VAR r,r2: INTEGER;
BEGIN
  r2 := a;
  REPEAT r := r2; r2 := r - (r - a DIV r) DIV 2 UNTIL r2 = r (* Newton *);
  IF 4*r*(r + 1) + 1 ≤ 4*a THEN r := r + 1 END (* first binomial law *);
  RETURN r;
END Sqrt;
```

For larger arguments 32–bit integers can be used equivalently. To avoid division by zero errors the argument n = 0 must be treated individually.

Remains to do the division. Here the simplest solution is to use fixed point binary arithmetic, e.g. shift the numerator four bits to the left beforehand and decide upon the remainder being ≥ 8 afterwards.

At that point the filling problem can be addressed. Since the rectangle is convex and the vertices lie on lattice points, each scan line between y_1 and y_4 passes through exactly two edges. The algorithm thus starts at y_1 and repeatedly determines the left hand and right hand intersection with the coutour of the rectangle until it has arrived at y_4, much like a horizontal line sweeping through the plane.

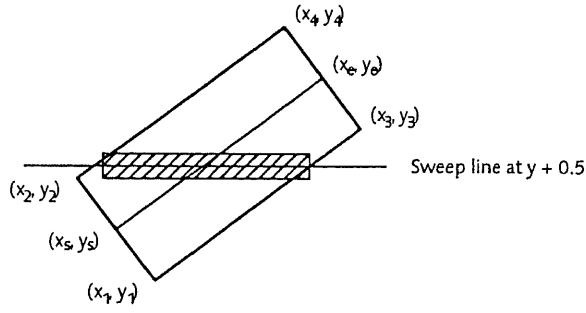

Figure 3.1

For a possible generalization to arbitrary polygons, it is important to use half integer coordinates for evaluating the intersections, i.e. to actually work with y + 0.5 rather than y only. Then there is always

an even number of intersections per scan line and the algorithm can draw small horizontal lines between two consecutive odd and even numbered intersections (parity fill).

In the present situation a simpler approach is possible. Since the sweep line always passes through exactly two edges, the algorithm can setup two ordinary line drawing procedures at (x_1, y_1), one directed towards (x_2, y_2), the other towards (x_3, y_3). Whenever these two have arrived at the same y–coordinate, their x–coordinates denote the intersection of the sweep line with the contour of the rectangle. The algorithm simultaneously walks along the contour in either direction until on both sides the next y–coordinate has been arrived at.

These Bresenham type steps can be sketched out by the following program fragment, assuming the required quantities are collected in a record (c.f. Chapter 1, Section 0):

```
PROCEDURE BresStep(VAR p: BresParms);
BEGIN
  IF it is a steep line THEN
    p.y := p.y + 1;
    IF real line is closer to x + 1 than to x THEN p.x := p.x + 1 END;
  ELSE
    REPEAT p.x := p.x + 1 UNTIL real line is closer to y + 1 than to y;
    p.y := p.y + 1;
  END;
END BresStep;
```

Eventually the walking arrives at (x_2, y_2). There the left hand collection of Bresenham parameters have to be setup for a line from (x_2, y_2) to (x_4, y_4). It then proceeds while it has not reached (x_3, y_3). This time the right hand parameters have to be re–initialized; to a line from (x_3, y_3) to (x_4, y_4). Finally both walks meet at (x_4, y_4) and the filling process is terminated.

Before rushing into the implementation an important detail has to be taken into account. These walking procedures actually have to return for each scan line the first and last pixel that is covered by the rectangle by at least 50%. The Bresenham type algorithm resulting thereof is slightly different from the one introduced in the first chapter and is therefore discussed next.

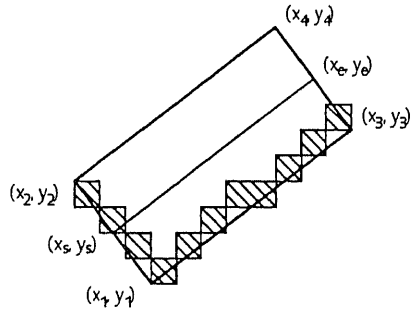

Figure 3.2

The pixel that covers the area $[x_1..x_1+1] \times [y_1..y_1+1]$ in the above Figure 3.2 *is* covered by the rectangle by at least 50%. In the present case for the line from (x_1, y_1) to (x_3, y_3) along which these last–pixels are determined,

$$\frac{dy}{dx} \leq 1$$

and therefore the next such pixel is either $[x_1+1..x_1+2] \times [y_1..y_1+1]$ or $[x_1+1..x_1+2] \times [y_1+1..y_1+2]$. In other words, x is increased always and y only if it is covered by less than 50%. This decision becomes quite simple, since the function for which to take the area is linear.

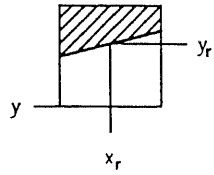

Figure 3.3

Thus whether or not to increase also y is equivalent to $y_r - y \geq 0.5$ (c.f. Figure 3.3). Substituting the equation of the real line

$$y_r := m \cdot x + q = \frac{dy}{dx} \cdot (x - x_1) + y_1$$

evaluated at x + 0.5, this leads to the invariant

$$d := (2x - 2x_1 + 1) \cdot dy - (2y - 2y_1 + 1) \cdot dx < 0$$

and

$$y := y + 1 \Leftrightarrow d \geq 0.$$

At start, $x = x_1$ and $y = y_1$, thus

$$d = dy - dx.$$

Then for each iteration

$$d := d + 2 \cdot dy; \; x := x + 1$$

and eventually (guarded)

$$d := d - 2 \cdot dx; \; y := y + 1.$$

In case

$$\frac{dy}{dx} > 1$$

the argument is analogous (through interchaning the roles of x and y) and is therefore not given. The only place in which steep lines differ from flat lines is the question, whether to take the x–coordinate to serve as an edge intersection *before* or *after* a Bresenham step. The following program fragment anwers this.

```
PROCEDURE Line(xs,ys,xe,ye,w: INTEGER);
  VAR left, right: BresParams;

  PROCEDURE BresStep(VAR p: BresParms);
  BEGIN
    p.drawY := p.y;
    IF p.dy > p.dx THEN
      p.drawX := p.x;
      p.y := p.y + 1; p.d := p.d – p.dx;
      IF p.d < 0 THEN p.x := p.x + 1; p.d := p.d + p.dy END;
    ELSE
      REPEAT p.x := p.x + 1;  p.d := p.d – p.dy UNTIL p.d < 0;
      p.y := p.y + 1; p.d := p.d + p.dx; END;
      p.drawX := p.x;
    END;
  END BresStep;

BEGIN (* Line *)
  ...
  WHILE ... DO
    BresStep(left); BresStep(right);
    Fill(left.drawX, left.drawY, right.drawX–left.drawX, 1);
  END;
  ...
END Line;
```

The program assumes the existence of a procedure Fill(x, y, w, h: INTEGER) that fills a rectangular area with lower left corner (x, y), width w and height h, aligned with the coordinate axes.

The final algorithm will have to cover for both dy < 0 and dx < 0, as for simple lines. Therefore if dy < 0 then p.drawY := p.y – 1, in order to agree with the procedure Fill. Secondly lines may be considerably wider than long, so it should comprise the possibility to arrive at (x_3, y_3) *before* (x_2, y_2). And lastly lines of width 1 pixel vanish if drawn at an angle of $\pi/4$. For such lines

$$dx = dy = \frac{1}{4}\sqrt{2} \approx 0.35,$$

which if evaluated as described would be rounded down for both the first and the second vertex. The solution to this problem is to also evaluate 2·dx and 2·dy, instead of symmetrically rounding, at the expence of doing two square roots, rather than one only. However this still does not cover the problem of 2 pixel lines versus 1 pixel lines, again at an angle of $\pi/4$, to which there is no logical decision solution.

3.1. Polygons

This section will treat polygons with mitered line joints, since they are the only objects of interest that the brush trajectory approach cannot model reasonably (c.f. Chapter 2). Such polygons arise from individual lines by conceptually extending their outer edges upto the point where the latter intersect. If the vertices of the original lines are appropriately replaced by these intersection points, and if the algorithm to fill rectangles is generalized to arbitrary convex quadrangles, these polygons can then be drawn about as efficiently as separate lines.

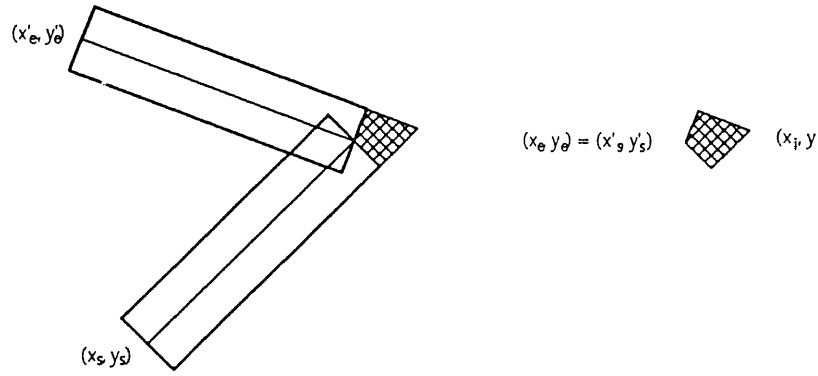

Figure 3.4

In Figure 3.4 the coordinates of the second line are primed, thus at the intersection $(x_e, y_e) = (x'_s, y'_s)$. The following Figure 3.5 repeats the rhomboid cross-hatched above. Its righthand vertex is the point of intersection under consideration.

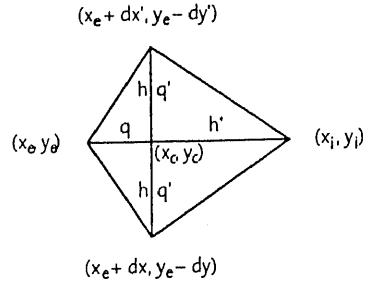

Figure 3.5

The point (x_e, y_e) is given, the quantities dx and dy have been introduced in Section 3.0, dx' and dy' are the respective ones for the second line and the computation of (x_c, y_c) is obvious. In terms of q and h'

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \frac{q + h'}{q} \cdot \begin{pmatrix} x_c - x_e \\ y_c - y_e \end{pmatrix}.$$

Due to similarity in the above rhomboid

$$\frac{h}{h'} = \frac{q}{q'}; \quad h = q'; \quad h' = \frac{h^2}{q}$$

and this becomes

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \frac{q^2 + h^2}{q^2} \cdot \begin{pmatrix} x_c - x_e \\ y_c - y_e \end{pmatrix}.$$

Formulating x_c, y_c, q and h in terms of dx, dy, dx' and dy'

$$\begin{pmatrix} x_c \\ y_c \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} x_e + dx' + x_e + dx \\ y_e - dy' + y_e - dy \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \frac{1}{2} \cdot \begin{pmatrix} dx' + dx \\ -(dy' + dy) \end{pmatrix}$$

$$\begin{pmatrix} x_c - x_e \\ y_c - y_e \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} dx' + dx \\ -(dy' + dy) \end{pmatrix}$$

$$q = \left| \frac{1}{2} \cdot \begin{pmatrix} dx' + dx \\ -dy' - dy \end{pmatrix} \right|; \quad q^2 = \frac{1}{4} \cdot ((dx + dx')^2 + (dy + dy')^2)$$

$$h = \left| \frac{1}{2} \cdot \begin{pmatrix} dx' - dx \\ -dy' + dy \end{pmatrix} \right|; \quad h^2 = \frac{1}{4} \cdot ((dx' - dx)^2 + (dy - dy')^2).$$

Collecting all the terms

$$\frac{q^2 + h^2}{q^2} = 2\cdot\left(\frac{dx^2 + dy^2 + dx'^2 + dy'^2}{dx^2 + dy^2 + dx'^2 + dy'^2 \;+\; 2\cdot(dx\cdot dx' + dy\cdot dy')}\right).$$

$$= 2\cdot\frac{w^2/2}{w^2/2 + 2\cdot(dx\cdot dx' + dy\cdot dy')}$$

$$= 2\cdot\frac{\rho\cdot\rho'}{\rho\cdot\rho' + \Delta x\cdot\Delta x' + \Delta y\cdot\Delta y'},$$

where in the last steps the various dx, dy, ... and dx^2, dy^2, ... terms have been substituted by their definitions and ρ and ρ' are defined as follows:

$$\rho := \sqrt{\Delta x^2 + \Delta y^2};\quad \rho' := \sqrt{\Delta x'^2 + \Delta y'^2}.$$

Thus the point of intersection now becomes

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \frac{\rho\cdot\rho'}{\rho\cdot\rho' + \Delta x\cdot\Delta x' + \Delta y\cdot\Delta y'}\cdot\begin{pmatrix} dx' + dx \\ -(dy' + dy) \end{pmatrix}.$$

After also replacing dx, dy, dx' and dy' the final result is

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \frac{w}{2}\cdot\frac{1}{\rho\cdot\rho' + \Delta x\cdot\Delta x' + \Delta y\cdot\Delta y'}\cdot\begin{pmatrix} \rho'\cdot\Delta y + \rho\cdot\Delta y' \\ -(\rho'\cdot\Delta x + \rho\cdot\Delta x') \end{pmatrix}.$$

The square roots are evaluated using the algorithm introduced in Section 3.0, while for the one final division again fixed point binary arithmetic is appropriate. Some care should be exercised though, since for widths of order 2^8 and Δx, ρ etc. of order 2^{10} this may be just about large enough to exceed the range of 32–bit integers.

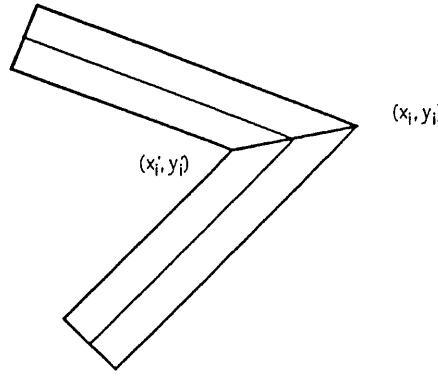

Figure 3.6

Upto that point the vertices of the trapezoids have been determined, as shown in Figure 3.6. The intersection point of the inner edges (x_i', y_i') can be obtained analogously to its outer companion by changing the sign of the $^w/_2$ term in the above formula.

For the subsequent filling process an important pecularity of Bresenham type algorithms has to be taken into account. In some place or other these algorithms decide whether or not to modify a coordinate based upon $d \geq 0$ or similar. These algorithms usually also treat cases where $dx < 0$ and/or $dy < 0$ (c.f. Section 1.0) symmetrically, in that instead of incrementing the coordinate they decrement it. Yet they use the same decision criterion as for $dx \geq 0$ and $dy \geq 0$. The advantage of doing so is a considerably shorter algorithm and exactly symmetric lines for e.g. the case of a line from (x_s, y_s) to $(x_e, y_s + dy)$ and from (x_s, y_s) to $(x_e, y_s - dy)$. The disadvantage, however, is that a line from (x_s, y_s) to (x_e, y_e) does not come out identical to that drawn from (x_e, y_e) to (x_s, y_s) for the points where $d = 0$ before the decision. This difference is generally neglected, but in case a Bresenham type algorithm is used to walk from (x_i, y_i) to (x_i', y_i') it becomes important. It thereby differently determines the x–coordinates for the edges at the junction of the two trapezoids. This results in some pixels being left out or some being drawn twice, which both are not acceptable, especially when XORing the pixels with the frame buffer.

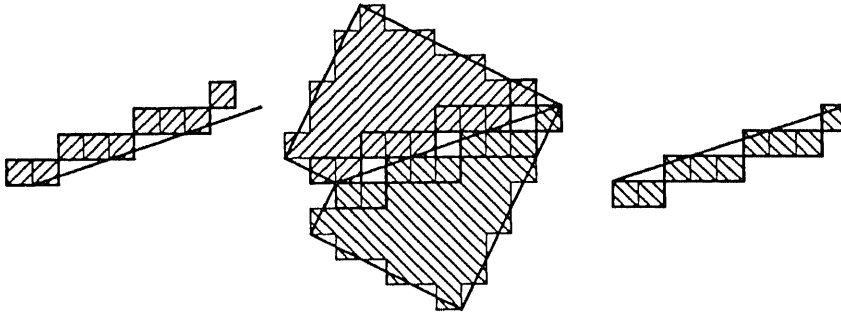

Figure 3.7

Figure 3.7 illustrates the point. A solution to that problem is to renumber the vertices such that filling is always done in the same direction, where the ordering relation is

$$(x_i, y_i) < (x_j, y_j) \Leftrightarrow (y_i < y_j) \vee (y_i = y_j) \wedge (x_i < x_j).$$

A simple bubble sort does this fast enough, the 4 elements are sorted in at most 6 exchange steps. The multitude of possible orientations is thereby reduced to 4 cases. They are shown in Figure 3.8.

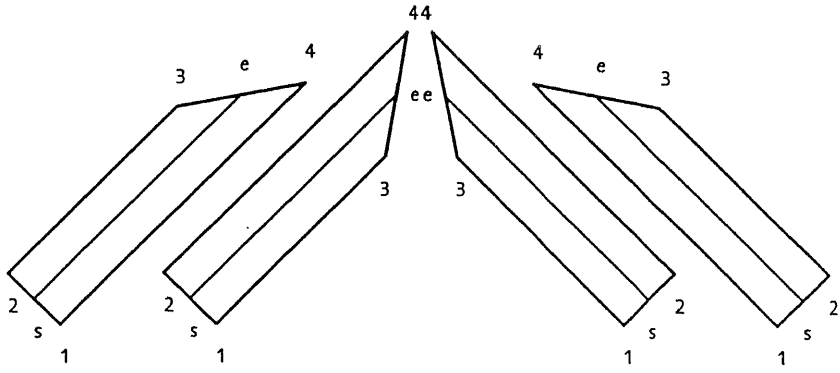

Figure 3.8

The general strategy to fill one of these trapezoids is to start at the point labeled 1, which is always the lowest, and to end at the point labeled 4, the highest of the four points. In order to setup the Bresenham parameters at point 1 correctly, the algorithm has to know on which side of the path from s to e the points 2 and 3 are. It does so by analyzing the orientation of the vector $\overrightarrow{sp}$ relative to $\overrightarrow{se}$ and the normal $\overrightarrow{sn}$, pointing out of the paper. If these three vectors form a right hand system, then p is on the right hand side of the path from s to e, as in the left part of Figure 3.9, and vice versa.

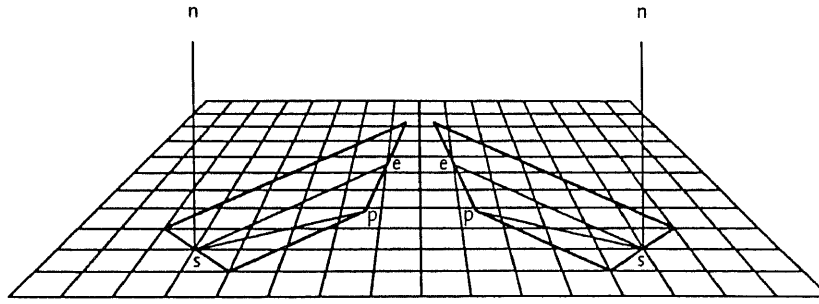

Figure 3.9

For $\overrightarrow{sp}$ and $\overrightarrow{se}$, which both lie in the xy–plane, being a right hand system is equivalent to

$$(\overrightarrow{sp} \wedge \overrightarrow{se})_z > 0.$$

Written in its components this expands to

$$(x_p - x_s)\cdot(y_e - y_s) > (y_p - y_s)\cdot(x_e - x_s).$$

To classify a trapezoid into one of the four types above, the decision has to be made for points 2 and 3 respectively. The algorithm then knows which points to use for setting up Bresenham parameters at point 1. Once arrived at point 2 it will also know which to use in order to proceed to point 3 and finally to point 4.

Compared to the lines discussed in section 3.0 drawing polygons with mitered line joints thus consist of no more than a different way to compute the four vertices of a quadrangle for given start and end points, apart from a slightly more general filling algorithm. For a closed polygon with n control points only two vertices have to be computed per control point. The most expensive operations therein are the two square roots. As discussed at the end of section 3.0 also straight lines require two square roots, hence the introductory claim to be able to draw mitered polygons about as efficiently as separate lines is informally confirmed.

3.2. *Circles*

To draw a circle of uniform width may be looked at as drawing an annulus with its inner and outer radius depending on the width chosen. The geometry of such figures doesn't pose any further problems, i.e. there are no vertices to compute or similar. If again some form of contour–tracing followed by immediate filling is used, an efficient algorithm may be anticipated.

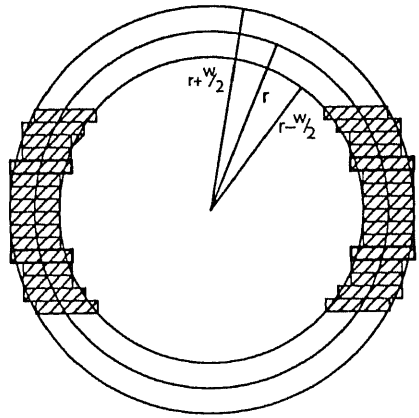

Figure 3.10

Figure 3.10 illustrates this. Two Bresenham type algorithms similar to those discussed in section 1.1 are set up at $(r-{}^{w}/_{2},0)$ and $(r+{}^{w}/_{2},0)$. They simultaneously walk counter–clockwise along the radii. Upon arrival at the same y–coordinate, due to symmetry, 4 small lines can be drawn with the same x–coordinates.

As with straight lines the algorithms to compute the x–coordinates should determine the first and last pixels on the respective scan line that are covered by at least 50%. Unlike with straight lines, however, this task is not that easy this time. To see this the following Figure 3.11 repeats one of these right edge pixels.

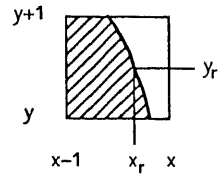

Figure 3.11

The shaded area above is

$$\int_{y}^{y+1} \sqrt{r^2 - y^2}\cdot dy - (x-1) = \frac{1}{2}\cdot\left(y\cdot\sqrt{r^2 - y^2} + r^2\cdot \mathrm{Sin}^{-1}\left(\frac{y}{r}\right)\right)\Big|_{y}^{y+1} - (x-1),$$

obtained through trigonometric substitution of the radicand. The result would have to be reduced to some expression that contains nothing but integer operations for efficient incremental evaluation.

An obvious first step is to try to reduce complexity through linear approximation, i.e. to do a taylor series expansion upto first order. If $y/r \ll 1$, then

$$\frac{1}{2}\cdot\left(yr\cdot\sqrt{1 - \left(\frac{y}{r}\right)^2} + r^2\cdot \mathrm{Sin}^{-1}\left(\frac{y}{r}\right)\right)$$

can be rewritten as

$$yr\cdot\left(1 - \frac{1}{4}\cdot\left(\frac{y}{r}\right)^2\right),$$

but in the worst case $y/r = 1/2\cdot\sqrt{2}$, which is sufficient for not being allowed to apply linearization.

The next attempt then is to geometrically approximate the value of the area, i.e. to linearize the integrand beforehand, as illustrated in Figure 3.12.

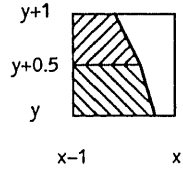

Figure 3.12

The shaded area here is easily seen to be

$$\frac{1}{4}\cdot\sqrt{r^2-(y+1)^2}+\frac{1}{2}\cdot\sqrt{r^2-(y+1/2)^2}+\frac{1}{4}\cdot\sqrt{r^2-y^2}-(x-1)$$

since this actually corresponds to the trapezoidal rule for numerical integration.

The filling strategy is to keep incrementing the y–coordinate while decrementing the x–coordinate only if the above area is smaller than $1/2$. To do so using integer arithmetic, the qualitative guard

$$\frac{1}{4}\cdot\sqrt{r^2-(y+1)^2}+\frac{1}{2}\cdot\sqrt{r^2-(y+1/2)^2}+\frac{1}{4}\cdot\sqrt{r^2-y^2}-(x-1)\le 0.5$$

has to be squared often enough to eliminate the roots. Since there are 3 roots this yields terms with upto the $2^{3rd} = 8^{th}$ power of x and y. But these terms do not cancel, thus for 32–bit arithmetic it reduces coordinates to values below $2^{32/8} = 2^4$, which evidently is not acceptable.

Thus what can efficiently be done at best is to approximate the area by the value of the circular function at $y + 1/2$

$$\sqrt{r^2-(y+1/2)}-(x-1)\le 0.5$$

which is equivalent to

$$2\cdot(r^2-x^2-y^2+x-y)-1\le 0.$$

The mechanism to set up the invariant for (x,y) = (r,0) and to proceed immediately follows from the preceeding text and is therefore dispensed with. However this will generally cover for a subset of the circles to be drawn only, for the width of the circle may be comparable in dimension to the inner radius. As a consequence, much like the filling of trapezoids, three cases have to be distinguished. The case where the width equals or supercedes the actual radius is trivial and therefore excluded from subsequent considerations.

Depending on the geometry of the annulus the filling process may arrive at the very y–coordinate where the magnitude of the slope of the outer perimeter drops below 1 *before* the walk along the inner perimeter has finished, or *after* or at the *same time*. Thus in terms of Figure 3.13 either

$$y_i < y_o < r_i < r_o \text{ or}$$

$$y_i < r_i < y_o < r_o \text{ or}$$

$$y_i < y_o = r_i < r_o.$$

The important quantities to look at are y_o and r_i. Since

$$y_o = \frac{1}{2}\cdot\sqrt{2}\cdot r_o$$

then due to monotonicity

$$2 \cdot y_o^2 = r_o^2 < 2 \cdot r_i^2 \text{ (or } r_o^2 = 2 \cdot r_i^2 \text{ or } r_o^2 > 2 \cdot r_i^2)$$

will give the case discriminator using 32–bit integer arithmetic.

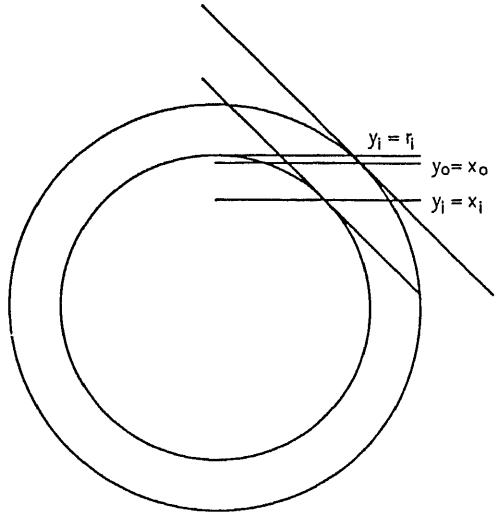

Figure 3.13

The circle drawing procedure resulting thereof is again very efficient, since its amount of computation favorably compares to the one for straight lines. Nevertheless it should not be kept as a secret that due to the simplification in constructing the invariant, small circles may appear a little like squares with round corners. Given a relatively coarse grid, scan–converting small objects are difficult to do in any case, for there quantization effects play a more and more important role, apart from properties of the display, printing and (human) perception devices.

Combining algorithms for walking along straight lines and circles, a diligent programmer may also want to implement circular arcs of uniform width. The difficulty to master for such a case is that both the start and the end of such an arc may be in any of the four quadrants. In order not to get lost in a considerable number of case combinations the arcs are broken into pieces that individually fit into one quadrant. Their junctions are either exactly horizontal or vertical and should not produce any gaps or pixels drawn twice, as may have been the case with naive implementation of mitered line joints. Then the resulting partial figures pose no concavities in the direction of the y–coordinate. Since the sweep line moves in y–direction there will always be two intersections of the sweep line with the partial figure, or none at all, but not more than two. The little extra cost represents the trade–off between not implementing a most general but considerably slower region filling algorithm and formulating $4 \times 4 \times 3 = 48$ individual cases for allowing the 3 above cases to happen together with both start and end lying in any of the 4 quadrants.

4. The Brush Trajectory Approach

4.0. A brute force algorithm

If bleeding properties of the paint and the strength used to apply the brush are ignored, then the only information left about the drawing itself is the area where the brush is in direct touch with the ground. To a certain extent this is much like unicolored woodcut, lithography or copperplate engraving on parchment paper free of dispersion. Such a brush can be described as a bit map comprising all the pixels that a single imprint thereof would leave on the ground it is applied to. A continuous line can then be understood as a closely spaced series of such imprints. The resulting shape of the line directly reflects the shape of the brush used, see Figure 4.0.

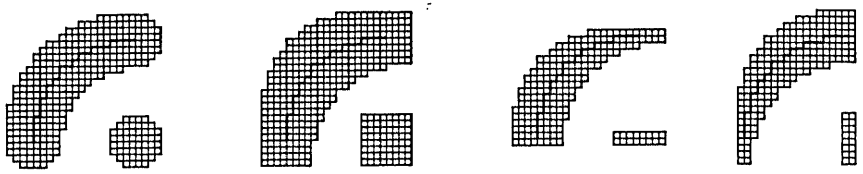

Figure 4.0

From a more technical point of view a single imprint of such a brush corresponds to one bit block transfer operation. The bit map defining the brush's shape is combined with a small rectangular area of the frame buffer using an OR operation. Drawing lines with such a brush can then readily be done using all the algorithms discussed in chapter 1 after "installing" an "alternate" dot procedure, one that does the very bit block transfer.

The performance of the resulting algorithm is rather poor, which is not too much of a surprise. Already with the second imprint of the brush most of the pixels that the latter is composed of are overprinted. For a line drawn with a round brush of n pixels diameter the majority of the pixels are consequently printed n times. The number of pixels drawn per unit length therefore comes close to n^2, hence the amount of processing time is $\sim O(n^2)$.

With modern computers efficiency is likely to cease somewhat to be the main concern of computer programming, maybe unfortunately. In any case the overprinting yields a far more drastic problem. If the involutory property of inverting the line over an existing image is required, the algorithm fails, as illustrated in Figure 4.1.

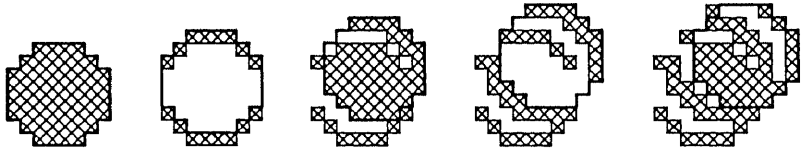

Figure 4.1

Two solutions to overcome this serious restriction can be thought of. One is to paint the curve into an auxiliary frame buffer and in a second step only invert an appropriate section thereof over the principal buffer. In view of its future use on a laser beam printer this solution would have to face the problem of a considerable requirement of storage resources. The background bit map's size for an 8 by 12 inch sheet on a 300 pixels/inch black and white printer could be anything upto 1 MByte. It was therefore rejected.

A more memory economic way to solve the invert problem is to work out the pixels added to the final result in each step. If only these pixels are inverted, then the result is correct with respect to inversion. To do so the output of the "alternate" dot procedure has to be buffered before sending it to the frame buffer. This way during the $n+1^{st}$ step the intermediate results of the n^{th} step are still known and the desired pixels can be obtained through set difference. Since the buffer has to extend to no more than the area right underneath the brush, this solution uses only twice the memory of a single brush, but its performance is evidently even worse.

It is usually a good piece of advice to do computational work in advance as much as possible, in order to evitate re-evaluating the same things repeated times. To address the performance problem one could for instance prepare short pieces of lines for all kinds of brushes and slopes once and forever and later on combine a (straight) line out of such pieces. Notice that this on the other hand agravates the memory consumption again, apart from being restricted to such brushes and slopes that acually *have* been prepared beforehand, let alone what to do for curves.

Efficiency problems with algorithms are quite frequently a direct consequence of an inadequately chosen data structure. The next section will therefore first of all investigate on an alternative representation of brushes.

4.1. A more sophisticated approach

To think of a better data structure let's look at a couple of possible brushes when specifying all th pixels they are made of.

Figure 4.2

In Figure 4.2 most brushes don't have any holes in them, nor do their outlines have any concavities. If they yet have these properties, consider their usefulness for drawing e.g. circles (apart from not being too realistic drawing instruments at all). Thus for the time being the brushes are assumed to be simply connected regions whose outlines are convex. This significantly simplifies subsequent considerations.

The larger the brushes are, the smaller gets the amount of information that they contain relative to their size in terms of storage locations. A round brush of e.g. diameter $^1/_3$ inch requires about 1 kByte of pixels on a 300 dpi printer, of which most bytes are all '1's. The same amount of information can also be represented if only the brush's outline is recorded. In view of its future application to a line oriented frame buffer the outline is best described as a series of line segments. A single line segment consists of a pair of (signed) integers denoting the number of pixels by which the brush extends to the left and to the right on that line. These line segments are in a way similar to the ones produced by the scan converting of a thin rectangle, c.f. section 3.0.

In related topics adjacency correlation is made use of in a similar way. When filling regions the knowledge of neighbours on the same scan line is used to outperform simple flood filling algorithms while for the storage and transmission of images e.g. run–length encoding is used (however it should be obvious by now that encoding is not the solution looked for in the present situation).

Displaying such a brush then means for each such line segment to replicate a bit pattern of all '1's into the frame buffer. At the first glimpse this may not look like too much of an advantage if the same strategy is used to display individual dots, since in that case the largest parts of the line segments are replicated repeatedly. At least the amount of memory used for the above $^1/_3$ inch brush has dropped to roughly $^1/_3$ kByte only, assuming each line segment is described by two integers. But there is more to this than reducing the trade–off between speed gain and memory loss.

With the new representation of brushes the memory usage, in terms of an average linear size n, has changed from $\sim O(n^2)$ to $\sim O(n)$, which is easy to see. This gives raise to some hope for a better algorithm processing it, for in the opposite case of an enlarged amount of data usually the amount of processing time is increased.

To find such an algorithm let's pick up the idea of tracking the incremental changes again and combine it with the concept of lazy evaluation. If as a result of one of the incremental algorithms of chapter 1 the brush is moved one step to the right, then all the line segments of the area under the brush possibly are extended by 1 pixel at their right end only. They are cross–hatched in the following Figure 4.3. But at that point *none has to be drawn* necessarily, since the number of line segments allocated to the brush still suffices to completely describe the intermediate figure (for a movement to the left, the extension takes place at the left end of the line segments by symmetry).

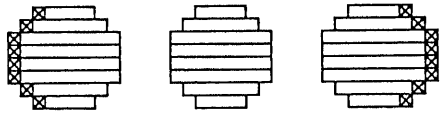

Figure 4.3

The actual replication of line segments into the frame buffer is delayed up to the point where the incremental movement has a vertical component. In such a case exactly 1 line segment leaves the buffered area underneath the brush. In Figure 4.4, this line is hatched vertically. For sufficiently continuous cases the latter is not expected to be subject to extension during the steps that immediately follow. Therefore, it *should be drawn.*

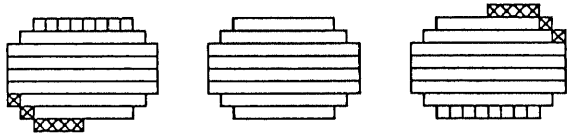

Figure 4.4

Now that 1 line segment has left, a new one is added at the vertically opposite side of the buffered area. Note that also vertical motion is symmetric with respect to the sign of the vertical component. If

the top most line is replicated, a new one is added at the bottom, and vice versa. The remaining line segments eventually have to be extended, but not drawn.

For motion in both horizontal and vertical direction the preceeding two methods are combined. The resulting algorithm is substantially faster than replicating the entire brush for each dot to be drawn. It is given in full detail in the next section.

4.2. Algorithm for drawing lines with a brush

To offer both flexibility and safety in its use the brushing algorithm is best encapsulated in a module of its own. Any operation manipulating a brush therefore requires the brush to be a parameter of the respective procedure. But before the data structure describing a brush is given, the following points should be taken into account.

1. Each brush consists of a series of records denoting the start and end point of a single line segment. For random access purposes (efficiency) this series is best formulated as an *array* structure. For memory effectiveness its upper limit however should be *variable.*
2. An application of the brush algorithm *has to be allowed* to setup these line segments. While the algorithm is under way each brush needs more information than its outline only (the current coordinates of the brush and a buffer for those line segments which are right underneath the brush while drawing). Yet to guarantee consistency this "internal" information *must not be accessible* to the application.
3. The application should be given a chance to decide on its own which is the position of the brush relative to the trajectory it is following. While for some users it may be desirable not to have to worry about that, others might not want e.g. a square brush to be pinned down at its upper left corner. Therefore a complete specification of a brush also comprises a base line, much like when denoting the descenders of characters. This serves as a y–offset (the x–offset can be included already in the line segments).

To cover for both dynamic arrays and information hiding an abstract data type was chosen. In Modula–2 these are opaque types, implemented as pointers. Since memory has to be allocated to them at run–time anyhow, a careful allocation thereof can provide for the "variable" size array. The base type of these pointers is specified in the implementation part only, thus covering for the information hiding requirement. Unfortunately there is no "public" part with these opaque types, i.e. one cannot specify some fields of the record to be "visible" from outside. They have to be made accessible via procedure calls. Using Modula–2's open array parameter mechanism a single procedure can allocate the memory and setup the brush including its outline without the caller having explicit access to the "internals". Using opaque types the application can "own" several brushes while not being restricted to a certain limit, as it would be the case when identifying the brushes via a number that is actually the subscript into a "private" array. Here is the brush descriptor:

```
CONST
   maxBrushSize = 8190 (* any arbitrarily large number *)

TYPE
   ScanLine = RECORD left, right: INTEGER END;
   OutLine = ARRAY [0..maxBrushSize–1] OF ScanLine;
   Brush = POINTER TO BrushDesc;
```

```
BrushDesc = RECORD
   width: INTEGER;
   baseLine: INTEGER;
   x,y: INTEGER;
   buffer: POINTER TO OutLine;
   outLine: OutLine;
END;
```

The fields *width*, *baseLine* and *outLine* are the properties of the brush, while *x*, *y* and *buffer* are needed during the algorithm only (thus *buffer* may be deallocated again afterwards). This suggests to provide a turtle graphics type interface for drawing lines with such a brush.

```
PROCEDURE BrushDown(brush: Brush; atX,atY: INTEGER);
PROCEDURE MoveBrush(brush: Brush; toX,toY: INTEGER);
PROCEDURE BrushUp(brush: Brush);
```

In a preliminary *BrushDown* step the shape of the brush is combined with information about position, whereby *atX* is moved to *x*, added to each of *outLine*'s *left* and *right* fields and brought to *buffer* (after allocating it), while *atY* is moved to *y*. During the subsequent calls to *MoveBrush* the original *outLine* is used together with *toX* and *toY* to modify *x*, *y* and *buffer*, details of which are given in the next paragraph. At the end *BrushUp* prints the line segments that remained in *buffer* due to the delayed printing in the preceeding step(s) and disposes of *buffer*. Note that *BrushDown* immediately followed by *BrushUp* leaves a single imprint of the brush.

The actual algorithm for moving the brush mainly has to distinguish movements with vertical component from those without. Following the sketches of the preceeding section, mainly Figures 4.3 and 4.4, the complete algorithm is as follows (formulated on a fairly high level of abstraction):

```
PROCEDURE MoveBrush(brush: Brush; toX,toY: INTEGER);

BEGIN
   WITH brush↑ DO
      IF toY = y THEN
         (* extend line segment either to the right or to the left *)
         IF toX > x THEN ∀i: buffer↑[i].right  := Max(buffer↑[i].right, toX + outLine[i].right)
         ELSE            ∀i: buffer↑[i].left   := Min(buffer↑[i].left,  toX + outLine[i].left)
         END
      ELSIF toY > y THEN
         (* draw lowest line segment of buffer *)
         WITH buffer↑[0] DO Fill(left,y-baseLine,right-left,1) END;
         (* adjust the other width-1 line segments *)
         ∀i≠width-1:   buffer↑[i].left  := Min(buffer↑[i+1].left,  toX + outLine[i].left);
                       buffer↑[i].right := Max(buffer↑[i+1].right,toX + outLine[i].right);
         (* and add another line segment on top *)
         buffer↑[width-1].left  := toX + outLine[width-1].left;
         buffer↑[width-1].right := toX + outLine[width-1].right;
      ELSE
```

```
        (* draw highest line segment of buffer *)
        WITH buffer↑[width-1] DO Fill(left,y-baseLine+width-1,right-left,1) END;
        (* adjust the other width-1 line segments *)
        ∀i≠0:          buffer↑[i].left  := Min(buffer↑[i-1].left,  toX + outLine[i].left);
                       buffer↑[i].right := Max(buffer↑[i-1].right,toX + outLine[i].right);
        (* and add another line segment at the bottom *)
        buffer↑[0].left  := toX + outLine[0].left;
        buffer↑[0].right := toX + outLine[0].right;
      END;
      x := toX;
      y := toY;
    END;
END MoveBrush;
```

The procedure *Fill* is the one introduced in chapter 3. In the final implementation the ∀i have to be replaced by an appropriate iteration while the *Min* and *Max* statements are actually programmed out with conditional statements. Using *Min* and *Max* makes the program clearer, but the call of a procedure with all its parameter passing overhead can't be afforded at that point. Further performance gain is obtained through intensive use of pointer arithmetic, thereby evitating the repeated array indexing. Its use lacks the elegance otherwise known from Modula-2 and is therefore better done in assembly language.

Notice that pointer arithmetic may be used to simulate two "open" WITH-statements with records whose common field identifiers would otherwise lead to ambiguities. This low-level technique is sketched out in the following program fragment.

```
PROCEDURE MoveBrush(brush: Brush; toX,toY: INTEGER);

  VAR
    out,buf0,buf1: POINTER TO ScanLine;
    min,max: INTEGER;

BEGIN
  WITH brush↑ DO
    ...
    (* assume out = ADR(outLine[i]), buf0 = ADR(buffer↑[i]) and buf1 = ADR(buffer↑[i+1]), then *)
    buffer↑[i].left  := Min(buffer↑[i+1].left,  toX + outLine[i].left);
    buffer↑[i].right := Max(buffer↑[i+1].right,toX + outLine[i].right);
    (* is the same as *)
    min := toX + out↑.left;
    IF min < buf1↑.left  THEN buf0↑.left  := min ELSE buf0↑.left  := buf1↑.left  END;
    max := toX + out↑.right;
    IF max > buf1↑.right THEN buf0↑.right := min ELSE buf0↑.right := buf1↑.right END;
    ...
  END;
END MoveBrush;
```

The ultimate implementation was done in NS32x32 assembly language whereby the preceeding Modula–2 unlike solution was accelerated by almost another 40%. In assembly language to each of *out*, *buf0* and *buf1* a general purpose register is assigned. Then pointer arithmetic reduces to immediate adds and the fields of the $(i+1)^{st}$ line segment can be accessed via the register pointing to the i^{th} line segment, just with a different offset (the idea of using a circular buffer eventually evitating some array copying was dropped, since overall it wouldn't have saved any memory accesses indexing the arrays by i MOD brush↑.width).

4.3. *Discussion of the algorithm*

The algorithm is correct. To see this one can e.g. understand the brush B as a closed and bounded region Ω in the Euclidian plane $\mathbb{R}^2$. After *BrushDown* this region is located at some fixed point (x_0,y_0):

$$\Omega_0 := B(x_0,y_0)$$

Each call to *MoveBrush* builds the union of this region with a brush located at a different point:

$$\Omega_i := \Omega_{i-1} \cup B(x_i,y_i), \text{ where } |x_i - x_{i-1}| \leq 1, |y_i - y_{i-1}| \leq 1 \text{ and } i > 0$$

Thus by induction over i the resulting union Ω describes the shape of the final picture. That much to the abstract algorithm (a subsequent consideration will come back right to this point).

For the concrete implementation $\mathbb{R}^2$ is reduced to a few pairs of integral numbers. With respect to that quantization an array of line segments also describes a closed region. As long as the trajectory of he line has no vertical component, this region is completely described by the buffer contained in the ›rush (*Min* and *Max* are responsible for the proper union of two regions with respect to the :–coordinate). If it *does* move vertically then exactly *one* line segment becomes redundant.

To see this remember that an incremental algorithm is assumed to supply the *toX* and *toY* parameters and that the implementation was originally restricted to simply connected regions with convex outlines. It therefore can't be more than one line segment. Secondly, the redundant line segment is drawn, so its contribution to the final picture is not lost but merely transformed into an equivalent amount of ink, correctly placed on a sheet of paper or so.

At all times the complete description of the intermediate picture is composed of these two parts: the array of line segments not drawn yet and the ink already on paper, hence also after the last incremental step. Then with a call to *BrushUp* the line segments still in buffer are brought to paper and the ink part alone makes up for the complete picture.

To further gain confidence in the correctness of the final assembly language implementation circles were drawn with rather unrealistically thin pen–type brushes where ∀i outLine[i].left = outLine[i].right. They reproduced fine. Notice, however, that this holds true for the drawing modes *paint* and *erase* only, i.e. when the bits are replicated into the frame buffer with an OR and a AND NOT operation respectively. In the case of *invert* (XOR) the results of the brushing algorithm are difficult to justify under certain circumstances.

To see this the brushing of a circle is illustrated at various intermediate levels first. In Figure 4.5 the shaded areas represent the parts of the final figure that have already been drawn, while the blank areas are the ones still contained in the buffer.

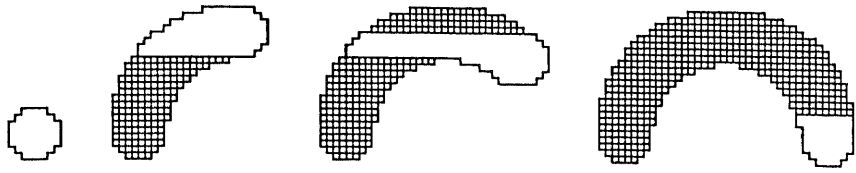

Figure 4.5

In the middle example filling stops going upwards and re-starts at the top. Yet this somewhat peculiar behaviour is in perfect accordance with the algorithm. But if the curvature increases, e.g. in the case of the vertex of a polygon, then due to the nature of the trajectory the brush overprints itself. Before visualizing possible consequences thereof, the principal effect of inverting selfintersecting curves is shown next.

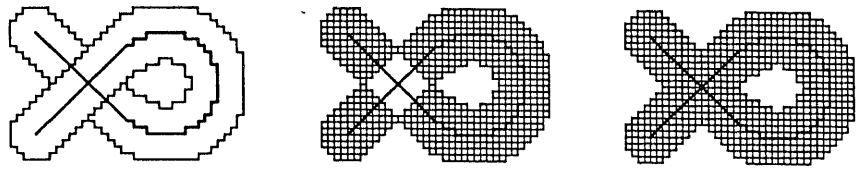

Figure 4.6

The left part of Figure 4.6 shows the outline of a picture produced by steering a round brush along an α-shaped trajectory, while the right part shows the final picture as one would expect it. The middle example finally shows the actual results of the algorithm. Due to the involutory nature of the inverting operation the area overprinted is wiped out again.

Now that the present approach models drawing, the emphasis could again be put on the incremental nature thereof. Instead of inverting the final figure over the existing picture the involutory property is associated with the brush itself. While the brush is being dragged accross the paper it applies "inverting" paint to it. From this point of view the middle example thus appears to be correct again and the case of inverting vertices can be resumed with.

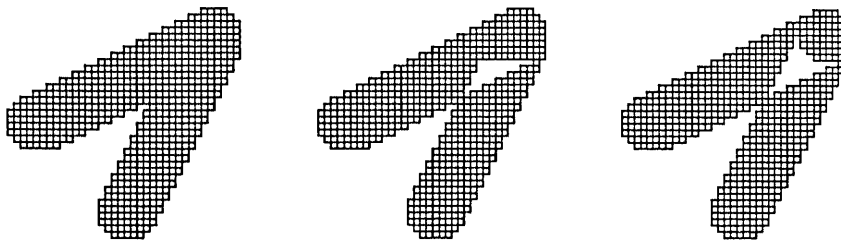

Figure 4.7

Figure 4.7 (left) shows one such vertex, together with the result anticipated following the preceding considerations (right) and what the algorithm actually produces (middle).

Although fixup techniques are successfully used elsewhere, fixing the output rather than eliminating the design flaws was not thought to be appropriate here. There are two solutions. If one can afford the memory consumption, the figure is first painted into a second frame buffer and then inverted over the primary buffer. Compared to a similar suggestion in section 4.0 the present case has at least a remarkable performance advantage which a subsequent consideration will come back to soon.

The alternative solution is to use a larger buffer for holding line segments not drawn yet, possibly comprising more than one line segment per scan line. This further delays the actual drawing while giving the algorithm a chance to test against overlapping line segments in order to eliminate the invert problem.

Apart from possibly having to decide upon a limit for the number of line segments kept in memory, little thought is required to apply the use of several line segments per scan line to multiply connected brushes with convex outlines. The theoretical assertion of the correctness of the algorithm has never made any other assumption about the region Ω than being closed and bounded. Therefore theoretically the algorithm can invariantly be used for more general brushes and for ultimately solving the invert problem, killing two birds with one stone. Its up to the individual to decide which results in terms of speed, algorithm complexity and memory requirement he or she is ready to pay what price for.

The following two points may give a decision foundation. With the single line segment per scan line algorithm each iteration requires updating n array elements, where n is the number of elements that *outLine* comprises and roughly denotes the vertical dimension of the brush. Therefore the amount of processing time is $\sim O(n)$ which for a large enough n is much better than the brute force algorithm $\sim O(n^2)$.

To give a more realistic idea of this advantage, both algorithms were used to draw straight lines, circles and splines for n = 32, corresponding to about a $^1/_{10}$ inch brush on a 300 pixels/inch printer. The newly developed brush algorithm outperformed the brute force algorithm by a factor of 10.

5. Conclusions

With the foundations given in this report a graphics package for drawing wide lines in the environment of the Ceres computer is close to its completion. It comprises the operations *Dot, Line, Polygon, Circle, Arc, Ellipse, EllipticalArc* and *Spline*. Three graphical attributes may be specified for drawing these lines and used independently (orthogonality).

1. *The underlying model (geometric/brushing).* If the geometric model is used, then the width of the lines has to be specified. For simplicity, polygons are always done with mitered line joints and no special line ends are used. For the brushing model the brush itself has to be supplied. Procedures are available to setup round brushes of any diameter or thin pen-type brushes of any width and inclination.

Note that in order to provide completeness, the geometric model had to be implemented also for ellipses and splines. In these cases the use of a round brush differs from the geometric model only if the curves are not closed, i.e. if there are line ends to deal with, which is true for elliptical arcs and open splines. A solution to this is to simulate the geometrical model by using a pen that is always held orthogonally to the trajectory. To do so closely spaced and appropriately chosen parts of the orthogonal trajectories are worked out and linked together by straight lines. The latter are entered in a ring buffer, much like was done with the output of the brushing algorithm, and drawn only when enough information is collected. Figure 5.0 shows this, although there the spacing has been widened for illustration purposes.

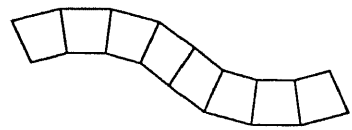

Figure 5.0

To work out the orthogonal trajectories for ellipses is pretty simple if the ellipse is understood as a function of two variables, see section 1.2. The spline algorithm uses forward differences that can be used instead of the differentials with little error here.

2. *The dash/dot pattern.* To substitute the lack of color or to illustrate "hidden" lines in 3D graphics, lines are often not continuous but disrupted in a regular fashion. For lines, polygons, circles and ellipses this is equivalent to having the procedure in charge draw a series of small lines, arcs and ellipses respectively, while for splines the program would have to rectify the curve in order to establish the notion of length along the trajectory. For such a spline γ in parametric form the euclidian length L is given by

$$L(\gamma) := \int \sqrt{\left(\frac{dx}{ds}\right)^2 + \left(\frac{dy}{ds}\right)^2} \cdot ds$$

This is again approximated by summing and using the forward differences instead of working out the differentials (ds = 1).

$$L(\gamma) \approx \sum \sqrt{dx^2 + dy^2}$$

The reason why this simplification is acceptable is because the splines are usually considerably wider than one pixel, hence the lengths of the individual dashes need not be that precise. The cost of doing an extra square root for each iteration was found to be 10% only and it doesn't cost anything, if it is not used at all, of course.

In any case it is important to describe the dash/dot pattern in terms of the width of the line (or brush) chosen such that when enlarging the pictures, the width of the line and its coordinates have to be scaled, but not the dash pattern. Furthermore it should be possible to obtain single dots with the dash pattern, too, i.e. to be able to have the procedure do a *BrushDown* immediately followed by a *BrushUp*. A data structure proven to be suitable is

```
TYPE
   DashPattern = RECORD
      length: INTEGER;
      on, off: BITSET;
   END;
```

where a '1' in *on* means that at that point the brush is to be "turned on", while a '1' in *off* implies the converse. To get a dash–dot–dash–dot pattern with dashes that are three times a long as wide, one could e.g. set *length* = 7, *on* = "1000010" and *off* = "0001010", as illustrated in Figure 5.1.

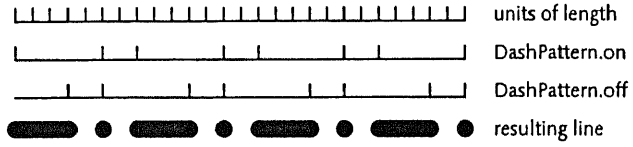

Figure 5.1

3. *The halftoning pattern.* Chapters 3 and 4 at some points eventually assumed the existence of a procedure Fill(x,y,w,h: INTEGER) which fills a rectangular area of the frame buffer with all '1's. If instead this procedure replicated some bit pattern, all the algorithms could equally well be used with halftoning. A reasonnable balance between flexibility (allow variable periodicity of pattern in both x– and y–direction) and efficiency (restrict x–periodicity to the size of a machine word) is offered by a pattern defined as follows.

```
TYPE
   GreyPattern = RECORD
      height: INTEGER;
      bits: ARRAY [0..height-1] OF BITSET;
   END;
```

The above definition allows to do a series of store instructions to replicate in horizontal direction, i.e. move a register to memory with post incrementing the address register. This is about what simple filling without pattern also has to do. Therefore drawing with a pattern can be obtained for free.

Care should be exercised which point the pattern is aligned to. While for drawing cross-hatched bar charts it is most desireable to have the 0^{th} bit of GreyPattern.bits[0] copied into that frame buffer's pixel whose lower left corner equals the x and y parameters of the replicating procedure, in the present case this would cause a serious problem. The rectangles replicated in all the algorithms are always 1 pixel high, which effectively reduces the grey pattern definitions to such of height 1. To produce e.g. a 50% shade of grey one would have to copy one bit set out of GreyPattern.bits, rotate it and produce an auxiliary pattern out of it before using it for replication. This problem vanishes if on the other hand the 0^{th} bit is always aligned with some fixed point of the frame buffer and not relative to the rectangle replicated (without loss of generality, this may be the origin). The effect of the alignment choice is illustrated in Figure 5.2 (relative alignment left, absolute alignment right).

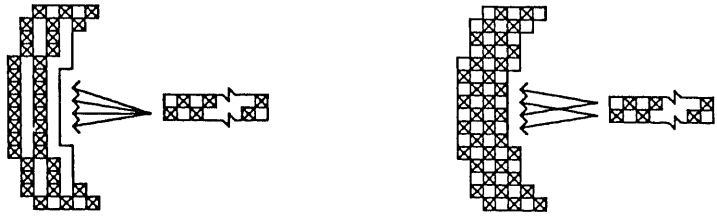

Figure 5.2

One should not conclude that one of the choices is wrong at all, they simply belong to different underlying models. The one required is much like cutting the figures out of some background layer that has been uniformely covered with some halftoning pattern beforehand.

If the thin rectangles are not simply copied into the frame buffer, but combined with the latter via a logical OR, XOR or AND NOT, the same procedure can as well cover for painting, inverting or erasing. The actual implementation should furthermore comprise as a parameter some reference to the frame buffer (or device) it operates on, such that programs can work with background bit maps and/or with more than one screen at the same time.

It thus constitutes the single port by which the actual raster device is accessed. If the graphics package implemented on top of this is to be used with different hardware, e.g. a color screen or a display controller with pattern and mask registers, then this is the right place to set about it.

At the end remains the question, "Was all that work worth the amount of time spent for?" With the increased use of computer graphics in all kinds of application programs, it is certainly a field necessary to know about. But where to learn it from? Commercial products like PostScript [Post85] or DDL [DDL86] only say *what* they do or what a user thereof has to do to have them do it, but they are not interested in revealing *how* they do it. Textbooks talk pretty much about elementary line drawing or simple curve filling algorithms, but they don't say very much about how to do *wide* lines [Fole84], [Newm82]. And if so, some of the ideas were found to be difficult to share: *"(...) Although solid-area scan conversion from a geometric description of the thick lines would yield the same results, the incremental method is substantially faster. (...)"* ([Newm82], Chapter 17-2). The experiences of the author are most definitely in the opposite direction. The incremental method mentioned roughly corresponds to the

algorithm given in section 4.0, while the geometric approach is given in section 3.0. However the algorithm of section 4.2 was found to be 10 times as fast as the one for section 4.0 while the one of section 3.0 is another 5 times faster than the one of 4.2, again for a line of 32 pixels width. Already from this point of view the answer to the initial question must therefore be "Yes".

Acknowledgements

I wish to thank J.Gutknecht for having given me the chance to work independently so that I could spend a considerable amount of time on the project without having been put pressure on. Thanks also to R.Griesemer, M.Odersky and C.Szyperski for carefully reading the manuscript and making valuable suggestions and to U.Hiestand and Ch.Vetterli for patiently waiting for graphical operations that their software heavily relies on [Hies88], [Vett89]. Finally thanks to H.R.Schär [Schä86] whose formulae editing facilities were a pleasure to work with.

References

[Bres65] Jack E.Bresenham
Algorithm for computer control of a digital plotter
IBM Systems Journal, Vol. 4, No. 1, 1965

[Bres77] Jack E.Bresenham
A Linear Algorithm for Incremental Display of Circular Arcs
Communications of the ACM, Vol. 20, No. 2, February 1977

[DDL86] –
DDL Tutorial
Imagen Corporation 1986

[Eber87] Johann Jakob Eberle
Development and Analysis of a Workstation Computer
Ph.D. Thesis ETHZ No. 8431, 1987

[Fole84] James D.Foley, Andries Van Dam
Fundamentals of Interactive Computer Graphics
Addison–Wesley Publishing Company 1984

[Heeb88] Beat Heeb
Design of the Processor-Board of the Ceres-2 Workstation
Techn.Report IFI ETHZ No. 93, November 1988

[Hies88] Urs Hiestand
Konzipierung eines Graphikpaketes und Integration in Opus
Diploma Thesis ETHZ April 1988

[Kohe88] Eliyezer Kohen
Two-Dimensional Graphics on Personal Workstations
Ph.D. Thesis ETHZ No. 8719, 1988

[Newm82] William M.Newman, Robert F.Sproull
Principles of Interactive Computer Graphics
McGraw–Hill, 1982

[Post85] –
The PostScript Tutorial and Cookbook
Adobe Systems

[Schä86] Hans–Rudolf Schär
Die Integration mathematischer Formeln in den Text-Editor Lara
Informationstechnik it, 28Jahrgang, Heft 6, 1986

[Vett89] Christian Vetterli
OPUS - Ein Objekt orientiertes Publishing System
Techn.Report IFI ETHZ, to be published 1989