# Rasterizing Curves of Constant Width

John D. Hobby

*Abstract*

This paper gives a fast, linear-time algorithm for generating high-quality pixel representations of curved lines. The results are similar to what is achieved by selecting a circle whose diameter is the desired line width, and turning on all pixels covered by the circle as it moves along the desired curve. However, we replace the circle by a carefully chosen polygon whose deviations from the circle represent subpixel corrections designed to improve the aesthetic qualities of the rasterized curve. For nonsquare pixels, equally good results are obtained when an ellipse is used in place of the circle. We introduce the class of polygons involved, give an algorithm for generating them, and show how to construct the set of pixels covered when such a polygon moves along a curve. The results are analyzed in terms of a mathematical model for the uniformity and accuracy of line width in the rasterized image.

# Rasterizing Curves of Constant Width

John D. Hobby

## 1. Introduction

A basic problem in raster graphics is that of rendering straight and curved lines as sequences of black and white pixels. The problem is best understood in the case where the line width is the minimum possible value: one pixel unit. The method illustrated in Figure 1a was discussed by Freeman as early as 1961 [3, 4]. Freeman's rule is based on a cross-shaped region

$$\bar{R} = \{\,(x,y) \mid (x = 0 \text{ and } -\tfrac{1}{2} \le y < \tfrac{1}{2}) \text{ or } (y = 0 \text{ and } -\tfrac{1}{2} \le x < \tfrac{1}{2})\,\}.$$

If we set up our coordinate system so that pixel centers lie in the set $\mathbb{Z}^2$ of points with integer coordinates, Freeman's rendering of a curve $C$ contains exactly the pixels centered at points $(m,n) \in \mathbb{Z}^2$, where $C$ intersects the region

$$\bar{R} + (m,n) = \{\,(x{+}m, y{+}n) \mid (x,y) \in \bar{R}\,\}.$$

In other words, we place a copy of $\bar{R}$ at each grid point and select the copies of $\bar{R}$ that intersect $C$.

A common variation is to replace $\bar{R}$ by its convex hull $H(\bar{R})$ as in Knuth's "diamond rule" [9]. This rule illustrated in Figure 1b is almost equivalent to Freeman's rule because it is difficult for the curve $C$ to intersect the diamond $H(\bar{R}) + (m,n)$ without intersecting $\bar{R} + (m,n)$. Differences can occur only at endpoints of $C$ or when the slope passes $\pm 1$ as shown in Figure 1c.

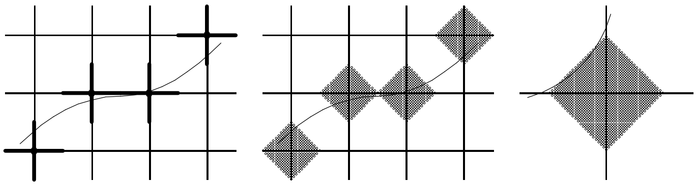



Figure 1: (a) Freeman's rule for rendering one-pixel-wide lines; (b) the diamond rule; (c) a case where the diamond rule plots an additional pixel.

Freeman's rule and the diamond rule both generate one pixel in every column for curves with slope bounded between $-1$ and 1, and one pixel in every row for curves with slope greater than 1 in absolute value. Well-known implementations of Freeman's rule include Bresenham's famous line algorithm [1] and Pitteway's algorithm for conic sections [11].

There is less of a consensus on how to render curved lines that are more than one pixel wide, but one natural approach is used by Tung [14]: To render a curve $C$ with width $d$, let $R_d$ be a circle of diameter $d$ centered on the origin (including the interior), and take all pixels whose centers lie in $C + R_d$, where $C + R_d$ is the set of all points $(x{+}x', y{+}y')$ where $(x,y) \in C$ and $(x',y') \in R_d$. Thus $C + R_d$ is a continuous version of the desired line, and we rasterize this region by taking the set of pixels whose centers it contains.

Tung's approach is well-known, and generalizations of it are often used for color graphics [15, 17], but other sources usually restrict $C$ to be a straight line. This avoids difficulties in rasterizing $C + R_d$ due to the complexity of its boundary curve; e.g., the boundary is an algebraic curve of degree six in the general case of the circle $R_d$ and a parabola $C$. Even when the complexity of

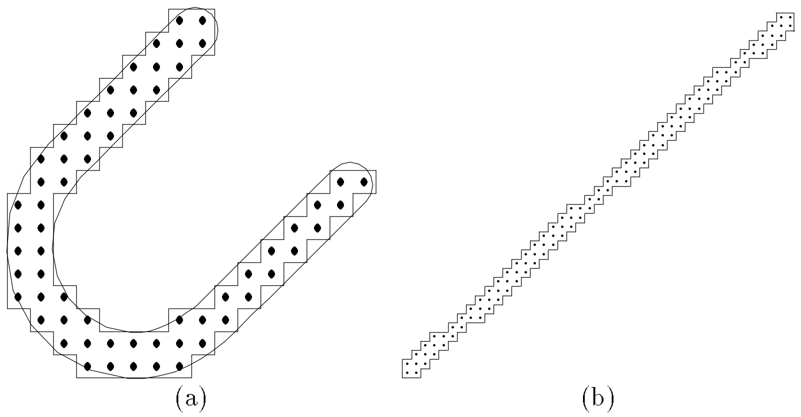


Figure 2: (a) The region $C + R_2$ superimposed on its rasterization, where $C$ is a U-shaped curve; (b) the rasterization of $C + R_2$, where $C$ is a line of slope 0.935.

rasterizing $C + R_d$ is tolerable, Figure 2 illustrates that there is poor control over the width of the rasterized curves, even when $d \in \mathbb{Z}$.

Let $R_1$ be a circle of diameter one centered on the origin, and compare Freeman's rule for generating a one-pixel-wide rendering of a curve $C$ with the process of rasterizing $C + R_1$ as shown in Figure 3c. Freeman's rule is widely used because it is relatively easy to find $(C + \bar{R}) \cap \mathbb{Z}^2$ and the "one pixel in every row or column" property tends to produce more uniform line width.

A well-known way to achieve better control over line width for $d > 1$ is to use Freeman's rule or the diamond rule, but turn on more than one pixel at a time as suggested by Knuth [9] and Fishkin and Barsky [2]. The main idea is to construct a set of pixels $B_d$ and turn on all pixels covered when a copy of $B_d$ is placed on top of each pixel in Freeman's rendering of a curve $C$. For instance, we can let $B_d = R_d \cap \mathbb{Z}^2$ and take the union of all $B_d + (m, n)$ where $C$ intersects $\bar{R} + (m, n)$. When $d$ is close to an even integer it works better to let $B_d = R_d \cap \left(\mathbb{Z}^2 + (\frac{1}{2}, \frac{1}{2})\right)$ and take $B_d + (m - \frac{1}{2}, n - \frac{1}{2})$ for each pixel $(m, n)$ in Freeman's rendering of $C + (\frac{1}{2}, \frac{1}{2})$. In any case, it can take a significant amount of time to turn on all the required pixels for each $(m, n)$.

A reformulation of Freeman's rule and the generalization to $d > 1$ leads to an algorithm that allows superior control of line width and is faster than any method considered so far. Figures 3a and 3b illustrate the reformulation of Freeman's rule: The set $\bar{R} + (m, n)$ intersects $C$ if and only if there exist points $r \in \bar{R}$ and $c \in C$ such that $r + (m, n) = c$. This happens exactly when $(m, n)$ is contained in the set

$$C - \bar{R} = \{\, c - r \mid c \in C \text{ and } r \in \bar{R} \,\}.$$

Since $C - \bar{R}$ and $C + \bar{R}$ differ only in the type of tie breaking used, Freeman's rule is equivalent to taking pixels whose centers lie in $C + \bar{R}$ instead of in $C + R_d$.

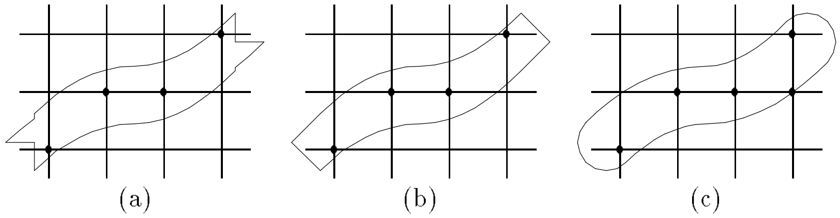
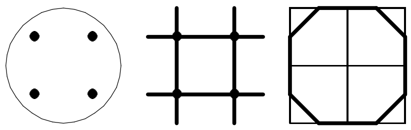


Figure 3: (a) A reformulation of Figure 1a as the rasterization of a region; (b) an similar reformulation of Figure 1b; (c) the rasterization of $C + R_1$ for the same curve $C$.

The union of all $B_d + (m, n)$ for $(m, n) \in C + \bar{R}$ or all $B_d + (m - \frac{1}{2}, n - \frac{1}{2})$ for $(m - \frac{1}{2}, n - \frac{1}{2}) \in C + \bar{R}$ is just the set of all $b + c + r$ for $b \in B_d$, $c \in C$, $r \in \bar{R}$, and $b + c + r \in \mathbb{Z}^2$. Thus the generalization to

$d > 1$ is equivalent to rasterizing the region $B_d + C + \bar{R} = C + (\bar{R} + B_d)$ by taking the set of pixels whose centers lie inside. In other words, Knuth's method achieves better control over rasterized line width by using regions like $\bar{R} + B_d$ instead of $R_d$ as shown in Figure 4.



(a) (b) (c)

Figure 4: (a) $R_2$ and $B_2$; (b) the region $\bar{R} + B_2$ with points in $B_2$ marked by dots; (c) the convex hull $H(\bar{R} + B_2)$ superimposed on a unit grid. Note that $\bar{R} + B_2$ is composed of four copies of $\bar{R}$ and has no interior.

Instead of using $\bar{R} + B_d$, it seems natural to consider replacing $\bar{R} + B_d$ by its convex hull as shown in Figure 4c. The result is analogous to replacing Freeman's rule by the diamond rule: We occasionally pick up an extra pixel when the curve direction passes certain rational slopes, but such changes have little effect on rasterized line width.

Polygons of the form $H(\bar{R} + B_d)$ belong to a special class of *pen polygons* introduced by Hobby [7]. Since not all pen polygons can be expressed as the convex hull of a set $\bar{R} + B$ for $B \subset \mathbb{Z}^2$, the following idea is almost a strict generalization of Freeman's rule and of Knuth's method: Given a curve $C$ and a positive real width $d$, we render $C + R_d$ by approximating $R_d$ with a pen polygon $\mathcal{P}(R_d)$ and then rasterizing $C + \mathcal{P}(R_d)$ by finding $\big(C + \mathcal{P}(R_d)\big) \cap \mathbb{Z}^2$. The added flexibility of pen polygons leads to superior control of rasterized line widths, and the properties of pen polygons lead to fast rasterization algorithms.

This work builds on ideas from [7] and applies them to the important practical problem of rasterizing curved lines with constant width. New contributions include a better algorithm for generating pen polygons of practical importance, techniques for generating the pixel output, and extensions to the case of devices with nonsquare pixels.

In Section 2, we define pen polygons and give an algorithm for computing the function $\mathcal{P}$ that finds a pen polygon appropriate for a given line width. We also show how to extend the algorithm so that pen polygons can approximate ellipses as required for generating constant-width lines on devices with nonsquare pixels.

In Section 3, we give an algorithm for rasterizing the region $C + P$ given a curve $C$ and a pen polygon $P$. To avoid reiterating known results, we assume the existence of a routine that takes a curve $C'$ that is monotonic in $y$, and determines for consecutive scanlines $y = i$, the pair of pixel centers that $C'$ passes between. We rasterize $C + P$ by applying this routine to portions of curves of the form $C + (\Delta x, \Delta y)$. Note that [7] defines the set of pixels required but gives no algorithm to compute it.

Finally, in Section 4 we get bounds on the running time, and we extend the mathematical models from [7] to analyze the accuracy and uniformity of width for the resulting rasterized curves.

## 2. Constructing Pen Polygons

The critical property of the region $\bar{R}$ defined in the introduction is that parallel supporting lines always have a vertical separation of one pixel or a horizontal separation of one pixel; i.e., if $C$ is a straight line $ax - by = c$, the region $C + \bar{R}$ is bounded by supporting lines $ax - by = c \pm d$, where $2d = \max(|a|, |b|)$.

Pen polygons are based on a generalization of this idea where opposite supporting lines have separation vectors in $\mathbb{Z}^2$. Thus a convex region $P$ is a pen polygon if for any two parallel supporting lines $\ell_1$ and $\ell_2$ with $P$ between them, there exist points $Q_1 \in \ell_1$ and $Q_2 \in \ell_2$ such that $Q_1 - Q_2 \in \mathbb{Z}^2$. The convex hull of $\bar{R}$ is a pen polygon where the difference vector is always either $(0, \pm 1)$ or $(\pm 1, 0)$.

It is shown in [7] that for a straight line $C$ of some fixed rational slope, the integral separation property ensures that the width of the rasterization of $C + P$ is independent of where $C$ falls on the pixel grid. For instance when $C$ is a line of slope one and $P$ is a hexagon with vertices $(\pm\frac{1}{2}, \pm 1)$ and $(\pm 1, 0)$ as shown in Figure 5, we can choose $Q_1$ and $Q_2$ to be opposite corners of $P$ so that $Q_1 - Q_2 = (1, -2)$. Thus if the supporting line containing $Q_1$ is $x - y = c$, then the other supporting line will be $x - y = c - 3$. Repositioning $C$ changes $c$, but with appropriate tie-breaking, there are always three integers $k$ for which the line $x - y = k$ is between the supporting lines. Since the pixel centers are spaced $\sqrt{2}$ units apart along lines $x - y = k$, we always include a total of $3/\sqrt{2}$ pixels per unit length along $C$. This avoids the problem illustrated in Figure 2 where one arm of the "U" has $\frac{2}{3}$ as many pixels per unit length.

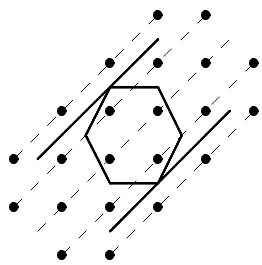


Figure 5: Pixel centers fall on equally spaced dashed lines, and solid lines show a pen polygon and the area swept out as it moves along a line of slope one. With appropriate tie-breaking rules, the area swept out by the pen will always contain exactly three dashed lines regardless of how the pen track is positioned on the pixel grid.

The primary motivation behind pen polygons is a desire to produce a uniform number of pixels per unit length for straight lines of rational slope, but as we see in Section 4.2, pen polygons also do a good job of controlling rasterized stroke width in the general case. Thus computing a pen polygon $\mathcal{P}(R_d)$ amounts to making sub-pixel corrections to the ideal shape $R_d$ in order to achieve better control over the rasterized stroke width.

The pen polygons of interest to us are all symmetrical about the origin in the sense that $z \in P$ if and only if $-z \in P$. It is shown in [7] that such *symmetrical pen polygons* are convex polygons with vertices of the form

$$z_1, z_2, \ldots, z_k, -z_1, -z_2, \ldots, -z_k,$$

where each $z_i$ belongs to the set $\frac{1}{2}\mathbb{Z}^2$ of points of the form $(\frac{m}{2}, \frac{n}{2})$ for $m, n \in \mathbb{Z}$. A few of the simplest symmetrical pen polygons are shown in Figure 6.

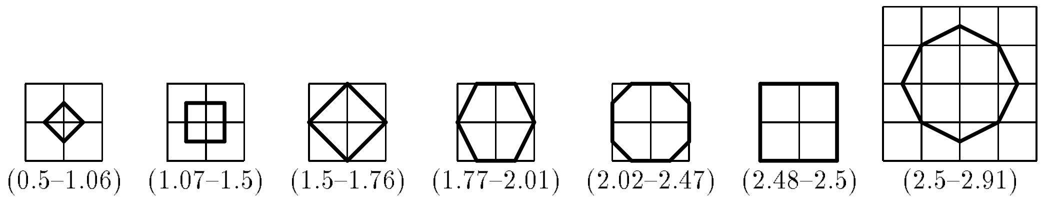



Figure 6: The pen polygons $\mathcal{P}(R_d)$ for $d$ in the indicated ranges. (Each pen polygon is superimposed on an integer grid.)

A supporting line of a symmetrical pen polygon with some rational slope $v/u$ will pass through some vertex $(\frac{m}{2}, \frac{n}{2})$. Hence it has an equation of the form $vx - uy = c$, where $u, v \in \mathbb{Z}$ and $c = \frac{1}{2}(vm - un) \in \frac{1}{2}\mathbb{Z}$. In other words, there are a countable number of possible locations for the supporting line, and they are separated from one another by multiples of the distance $1/(2\sqrt{u^2 + v^2}\,)$. This means that opposite supporting lines are separated by multiples of $1/\sqrt{u^2 + v^2}$.

As shown in [7], there are discrete possibilities for the width of rational-slope rasterized lines, and they correspond to the discrete possibilities for the separation between opposite supporting lines of the pen polygon. For instance, in the example of Figure 5, the supporting lines $x - y = c$ and $x - y = c - 3$ are separated by $3/\sqrt{2}$, and there are $3/\sqrt{2}$ pixels per unit length in the rasterization.

### 2.1. Pen Polygons that Approximate Circles

In order to draw lines of some width $d$, we need to generate a pen polygon $\mathcal{P}(R_d)$ that approximates the circle $R_d$ of diameter $d$ centered on the origin. For simple applications where only a small range of line widths are needed, it is possible to precompute the necessary polygons by hand. In fact, Figure 6 may contain all the required information. The purpose of this section is to explain how to compute $\mathcal{P}(R_d)$ when necessary.

The algorithm presented here is a simplified version of algorithm 1 from [7] with some optimizations and some new data structures. The version presented in [7] is somewhat impractical because of its extreme generality.

Since the supporting line positions are most tightly constrained for simple rational slopes, we define the function $\mathcal{P}(R_d)$ so that such supporting lines will be as close as possible to their ideal positions. This forces the rasterized stroke width to be as close as possible to $d$ at simple rational slopes.

Figure 6 shows the effects of this strategy of placing rational-slope supporting lines independently. For instance since $2.49 \approx 3.52/\sqrt{2}$, the separation of opposite supporting lines for $\mathcal{P}(R_{2.49})$ is rounded up to $4/\sqrt{2} \approx 2.83$ at slope one, and down to two at slope zero. Instead of making the pen polygons as circular as possible, we try to make them as close as possible to the desired circles.

We construct $\mathcal{P}(R_4)$ by starting with a square of size four as shown in Figure 7a and successively cutting corners off of it. In Figure 7b, we have added edges of slope $\pm 1$, each $3/\sqrt{2}$ units away from the origin. The completed pen polygon in Figure 7c is formed by adding edges of slope $\pm\frac{1}{2}$ and $\pm 2$, each $4.5/\sqrt{5}$ units away from the origin. Since all supporting lines of a circle of diameter four are two units away from the origin, the ideal values were $2.828/\sqrt{2}$ and $4.472/\sqrt{5}$ instead of $3/\sqrt{2}$ and $4.5/\sqrt{2}$.

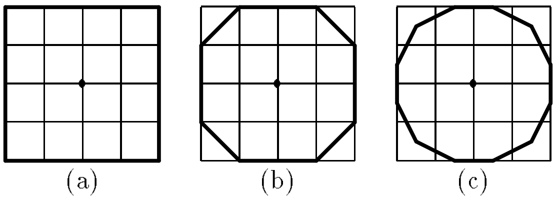



Figure 7: Three steps in the computation of $\mathcal{P}(R_4)$. At each stage, the tentative pen polygon is shown superimposed on a unit grid with the origin marked by a dot.

To make the corner-cutting process more precise, consider the case of a pen polygon based on $R_{11}$, and note that we can construct one quadrant of the pen polygon and allow the rest of the polygon to be dictated by symmetry constraints. Figure 8 shows how we might maintain one quadrant of the tentative pen polygon, always considering the vertices in counter-clockwise order when looking for a corner to cut off. Figure 8a shows one quadrant of the initial 11-unit square, and Figure 8b shows the corner at $(5.5, -5.5)$ replaced by an edge of slope one. Figure 8c is obtained by adding an edge of slope $\frac{1}{2}$ to cut off the corner at $(2.5, -5.5)$; in Figure 8d, we remove $(1.5, -5.5)$ and add an edge of slope $\frac{1}{3}$; in Figure 8e, the new edge has slope $\frac{1}{4}$. Since the desired supporting line of slope $\frac{1}{5}$ passes through the point $(.5, -5.5)$, that vertex is preserved and we move on to the vertex at $(2.5, -5)$ in Figure 8e. This vertex also cannot be cut off because we have already decided that the supporting line of slope $\frac{1}{3}$ should pass through it. Thus, we obtain Figure 8f by adding an edge of slope $\frac{2}{3}$ to replace the vertex at $(3.5, -4.5)$. We then add edges of slopes $\frac{3}{4}$ and $\frac{4}{5}$ as shown in Figures 8g and 8h. The vertex at $(5, -3)$ in Figure 8h is retained because the desired supporting line of slope $\frac{5}{6}$ passes through it.

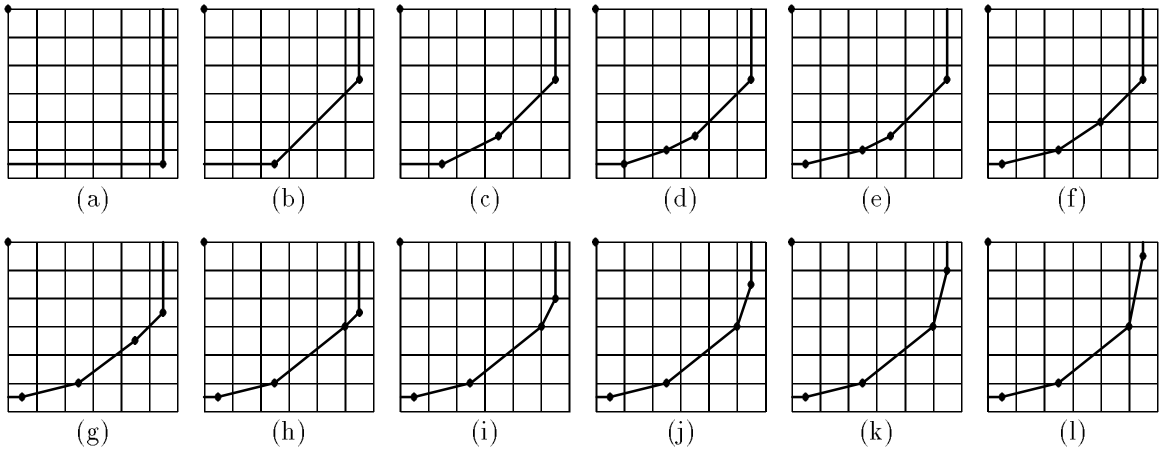


Figure 8: The computation of a pen polygon based on $R_{11}$. At each stage, one quarter of the tentative pen polygon is shown superimposed on a unit grid with the origin marked by a bold dot. Smaller dots mark the vertices of the tentative pen polygons.

The algorithm just suggested has been implemented by Knuth [10]. It performs satisfactorily, but as Figure 8 illustrates, the results are not always ideal: The edge of slope one in Figure 8h gets replaced by edges of slopes two, three, four, and then five, but the final polygon in Figure 8l still has a vertex at $(5, -3)$. We must require the final polygon to retain some point on the edge from $(2.5, -5.5)$ to $(5.5, -2.5)$ in Figure 8b, but it is hard to justify choosing $(5, -3)$ instead of $(4, -4)$. Thus as in [7], whenever we add a new edge to the tentative pen polygon, we select some point of $\frac{1}{2}\mathbb{Z}^2$ on that edge to be retained during subsequent processing.

New edges are placed as close as possible to supporting lines for $R_d$, and new *retention points* are placed as close as possible to the point of support; i.e., for an edge of slope $v/u$, we select the point of $\frac{1}{2}\mathbb{Z}^2$ closest to $ux + vy = 0$. Thus we compute $\mathcal{P}(R_{11})$ by starting as in Figure 8a, but with retention points at $(0, -5.5)$ and $(5.5, 0)$. The new retention points in Figures 8b and 8c are $(4, -4)$ and $(2.5, -5)$ respectively. No new retention points are created in Figures 8d and 8e because we have already decided to retain $(2.5, -5)$. Similarly in Figure 8f, both $(2.5, -5)$ and $(4, -4)$ are existing retention points

The retention point at $(4, -4)$ prevents the creation of an edge of slope $\frac{3}{4}$, so we create an edge of slope two with a retention point at $(5, -2.5)$ as shown in Figure 9g. New edges of slopes $\frac{3}{2}$, 3, and 4 are added in Figures 9h–j, but no new retention points are needed.

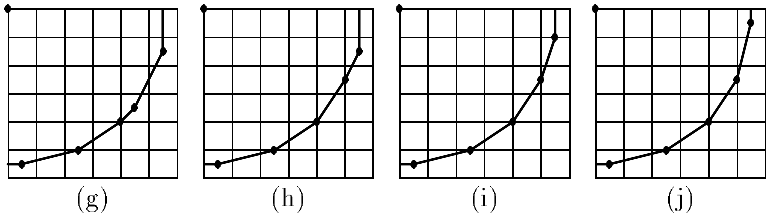


Figure 9: The final stages in the computation of $\mathcal{P}(R_{11})$ using retention points. The initial stages are as shown in Figures 8a–f.

We output vertices of the finished polygon in counter-clockwise order and improve on [7] by keeping the unfinished portion in a sequence of cells $s[t]$, $s[t-1]$, …, $s[1]$ maintained as a stack. Each $s[i]$ contains a vertex location $s[i].z$ and the counter-clockwise adjacent edge $s[i].r$. There is also an edge node $l$ for the edge clockwise adjacent to $s[t].z$.

Each edge node contains a direction vector $w$, integers $l_1$ and $l_2$ that locate the retention point, and an integer $c$ that locates the line containing the edge. Thus if $(u, v) = s[i].r.w$, the retention point on $s[i].r$ is

$$s[i].z + s[i].r.l_1\,(u,v) = s[i-1].z - s[i].r.l_2\,(u,v),$$

and $s[i].r.$ lies on the line $vx - uy = s[i].r.c$. It is convenient maintain the polygon scaled up by a factor of two so that the $z$, $c$, $l_1$, and $l_2$ fields have integer values.

The algorithm of Figures 10 and 11 maintains some invariants that can be expressed in terms of the *turn_amt* function defined on line 33 of Figure 11: If $1 \le i \le t$ and $e$ is the edge clockwise adjacent to $s[i].r$, then either $turn\_amt(e, s[i].r) = 1$, or $s[i].z$ is the sole remaining point on some previously chosen supporting line. When $turn\_amt(l, s[t].r) = 1$ and $(\bar u, \bar v) = l.w + s[t].w$, we have for all $\delta$ that the points

$$s[t].z - \delta\, l.w \quad \text{and} \quad s[t].z + \delta\, s[t].r.w$$

lie on the line $\bar v x - \bar u y = l.c + s[t].r.c - \delta$. Thus $\delta$ measures how much is to be sliced off, and line 10 computes the maximum slice that retains the retention points on $l$ and $s[t].r$.

```
 1  procedure make_pen(d);
 2  t ← 1;
 3  l.w ← (1, 0);
 4  s[1].r.w ← (0, 1);
 5  l.c ← s[1].r.l₁ ← s[1].r.l₂ ← round(d);
 6  s[1].r.c ← l.l₁ ← l.l₂ ← round(d);
 7  s[1].z ← (l.l₂, −l.c);
 8  while t > 0
 9  do { if turn_amt(l.w, s[t].r.w) > 1 then { pop; goto top of loop; }
10       δ ← min(l.l₂, s[t].r.l₁);
11       if δ > 0 then { (ū, v̄) ← l.w + s[t].r.w;
12                       c̄ ← l.c + s[t].r.c;
13                       δ ← min(δ, c̄ − round(d√(ū² + v̄²))); }
14       if δ ≤ 0 then { pop; goto top of loop; }
15       if δ = l.l₁ + l.l₂
16       then if s[t].r.l₁ + s[t].r.l₂ = δ
17            then { t ← t − 1; l.w ← (ū, v̄); }
18            else { rchop(t);
19                   eset(l); }
20       else { if s[t].r.l₁ + s[t].r.l₂ = δ
21              then lchop
22              else { t ← t + 1;
23                     s[t].z ← s[t − 1].z;
24                     lchop;
25                     rchop(t − 1);
26                     set_rp; }
27              eset(s[t].r); } };
```

Figure 10: A routine for constructing one quadrant of the pen polygon $\mathcal{P}(R_d)$.

Lines 2 through 7 set up one quadrant of the initial square as in Figure 8a, and lines 9 through 27 form the body of a loop that tries to replace the vertex $s[t].z$ with an edge in direction $(\bar u, \bar v)$. If this operation fails, we use *pop* to output $s[t].z$ and go on to the next vertex. At this point the edge assigned to $l$ has been "passed over" so line 32 throws away the unneeded retention point.

When lines 10–14 can find a positive integer $\delta$, lines 15–27 create a new edge with $w = (\bar u, \bar v)$ and $c = \bar c - \delta$. Then line 15 tests if the new edge causes $l$ to disappear, and lines 16 and 20 test if $s[t].r$ disappears. Normally the tests fail and lines 22–27 push the new edge onto the stack. Otherwise the update process is simplified by allowing the new edge to replace an existing edge. For $d < 40$, this simplified update happens about 32.7% of the time we reach line 15: Line 17 gets executed 1.8% of the time, lines 18–19 20.7%, and line 21 10.2%.

28 **procedure** $pop$;<br>
29 Output the vertex $\frac{1}{2}(s[t].z)$;<br>
30 $l \leftarrow s[t].r$;<br>
31 $t \leftarrow t - 1$;<br>
32 $l.l_2 \leftarrow l.l_1 + l.l_2$; $l.l_1 \leftarrow 0$;

33 **function** $turn\_amt(a, b)$; **return** $a.x * b.y - b.x * a.y$;

34 **procedure** $lchop$;<br>
35 $l.l_2 \leftarrow l.l_2 - \delta$;<br>
36 $s[t].z \leftarrow s[t].z - \delta * l.w$;

37 **procedure** $rchop(i)$;<br>
38 $s[i].r.l_1 \leftarrow s[i].r.l_1 - \delta$;<br>
39 $s[i].z \leftarrow s[i].z + \delta * s[i].r.w$;

40 **procedure** $set\_rp$;<br>
41 **if** $s[t-1].r.l_1 = 0$ **then** $s[t].r.l_2 \leftarrow 0$<br>
42 **else if** $l.l_2 = 0$ **then** $s[t].r.l_2 \leftarrow \delta$<br>
43 **else** { $s[t].r.l_2 \leftarrow \mathrm{round}\big((\bar{u}, \bar{v}) \cdot s[t-1].z/(\bar{u}^2 + \bar{v}^2)\big)$;<br>
44 $\qquad s[t].r.l_2 \leftarrow \max(0, \min(s[t].r.l_2, \delta))$; }<br>
45 $s[t].r.l_1 \leftarrow \delta - s[t].r.l_2$;

46 **procedure** $eset(e)$;<br>
47 $e.w \leftarrow (\bar{u}, \bar{v})$;<br>
48 $e.c \leftarrow \bar{c} - \delta$;

Figure 11: Support routines for constructing pen polygons.

We can reach line 14 with $\delta < 0$ in cases such as when $d = 2.47$ and $(\bar{u}, \bar{v}) = (1, 2)$. Then the ideal supporting line is $2x - y = 3$, but $2x - y$ is only 2.5 at $\frac{1}{2}s[t].z = (1, -.5)$.

A more unusual case occurs in the $set\_rp$ routine of lines 40–45. Lines 41 and 42 ensure that the new retention point will be coincident with an existing one on edge $l$ or edge $s[t-1].r$, if this is possible. Otherwise, line 43 uses the dot product $(\bar{u}, \bar{v}) \cdot s[t-1].z$ to locate the retention point as close as possible to the line $\bar{u}x + \bar{v}y = 0$. Ordinarily line 44 does not change $s[t].r.l_2$, but sometimes it is necessary to prevent $s[t].r.l_2$ from being out of range. This never happens when $d \le 58.54$, but it does happen at the slope five edge when $d = 58.55$.

## 2.2. Approximating Ellipses

In order to deal with devices where the vertical spacing between pixels differs from the horizontal spacing by some constant factor $a$, we need to construct a pen polygon whose height is approximately $d$ pixels and whose width is roughly $ad$ pixels. In other words, we need to define $\mathcal{P}(R_{d,a})$, where $R_{d,a}$ is an ellipse with height $d$ and aspect ratio $1/a$. To represent on such a device a curve $C$ with a width of $d$ vertical pixels, we work with the curve $X_a(C)$ obtained by applying the operator $X_a$ that scales $x$-coordinates by $a$. The appropriate pixel image is then obtained by rasterizing the region $X_a(C) + \mathcal{P}(R_{d,a})$ in the usual manner.

We now give changes to the code in Figures 10 and 11 that cause $make\_pen$ to compute $\mathcal{P}(R_{d,a})$. The initial square is turned into an initial rectangle by replacing "round($d$)" by "round($ad$)" on line 6. After scaling by two, the supporting line for the desired ellipse is

$$vx - uy = d\sqrt{\bar{u}^2 + a^2\bar{v}^2} \tag{1}$$

instead of just $vx - uy = d\sqrt{\bar{u}^2 + \bar{v}^2}$, so we change line 13 to read

$$\delta \leftarrow \min\big(\delta, \bar{c} - \mathrm{round}\big(d\sqrt{\bar{u}^2 + a^2\bar{v}^2}\big)\big);\ \}.$$

Since the retention point on the $(\bar{u}, \bar{v})$ edge should be as close as possible to $R_d$ after $x$-coordinates are scaled by $1/a$, we replace line 43 with

$$s[t].r.l_2 \leftarrow \text{round}\left(\frac{(\bar{u}/a^2, \bar{v}) \cdot s[t-1].z}{\bar{u}^2/a^2 + \bar{v}^2}\right).$$

## 3. Rasterizing Pen Polygon Envelopes

Since line widths are often relatively small, the time and space requirements of the pen polygon construction algorithm tend to be quite modest. Thus the overall time and space requirements of the line-drawing algorithm are usually dominated by the process of finding the pixel centers covered by a pen polygon $P$ as it moves along the given curve $C$. The purpose of this section is to show that this rasterization process is not much more difficult to implement than the well-known algorithms for rasterizing thin curved lines.

There are many different formulations for the problem of rasterizing thin curved lines. For our purposes, we shall assume the existence of a subroutine *plot* that takes a description of a curve $C'$ and generates for consecutive integers $j$, the value of $\lfloor x \rfloor$, where $(x, j)$ is the unique intersection of $C'$ and the line $y = j$. In other words if $C'$ is monotonic in $y$, $plot(C')$ finds for each scanline $y = j$ the pixels $(i, j)$ and $(i{+}1, j)$ that $C'$ passes between. Pratt gives an algorithm that solves this problem for conic sections [12]; Knuth solves a similar problem for parametric cubic curves [10]; Hobby also solves a similar problem for various class of algebraic curves. In fact, we may use any of the well-known algorithms for rasterizing thin lines according to Freeman's rule [8, 11, 16]: Apply Freeman's rule to $C' + (0, \frac{1}{2})$, $C' - (0, \frac{1}{2})$, or $C' - (\frac{1}{2}, 0)$, depending on whether $C'$ has slope between in the interval $[0, 1]$, in $[-1, 0]$, or outside $[-1, 1]$. If $S$ is the resulting pixel image, $plot(C')$ just finds pixels $(i, j) \in S$ where $(i + 1, j) \notin S$.

The relationship between thin lines and the region $C + P$ covered by $P$ as it moves along $C$ can be explained in terms of the theory of *tracings* and *convolutions* introduced by Guibas, Ramshaw, and Stolfi [5]. They give a procedure for deriving from $C$ and $P$ a *convolution tracing* that describes the boundary of the region $C + P$. The theory in [5] only applies directly when $C$ is polygonal, but it is not hard to make the necessary generalizations [7]. In fact, the entire theory generalizes naturally to algebraic curves [13].

The convolution tracing derived from a curve $C$ and a pen polygon $P$ can be thought of as a closed curve that divides the plane into regions according to the number of times $P$ passes over each point as $P$ moves along $C$. In other words, if $P$ were a paint brush, the convolution tracing would break up the plane according to the number of coats of paint received. For example, in Figure 12, the entire region enclosed by the convolution tracing is painted once, except for a small region just left of center that gets painted by two different corners of $\mathcal{P}(R_{2.2})$.

The convolution tracing for a symmetrical pen polygon $P$ and a curve $C$ consists of *curve segments* and *connecting segments* as delimited by the dots in Figure 12b. The curve segments are of the form $C' + z_i$, where $C'$ is a segment of $C$ and $z_i$ is a vertex of $P$; the connecting segments are straight lines of the form $e + c$ where $e$ is an edge of $P$ and $c$ is a point on $C$ where the tangent direction is parallel to $e$.

The rule for choosing a vertex $z_i$ for a curve segment $C'$ is to let $D$ be the tangent direction for some point on $C'$, and go around $P$ counter-clockwise until finding three adjacent vertices $z_{i-1}$, $z_i$, and $z_{i+1}$, where the direction $D$ is between $z_i - z_{i-1}$ and $z_{i+1} - z_1$ (inclusive). Thus the line through $z_i$ with direction $D$ is a supporting line for $P$ and an observer facing in direction $D$ would find $P$ on the left side of the line. Let $V(P, D)$ be a function that computes such a vertex $z_i$.

To form the convolution tracing in Figure 12b, let $P = \mathcal{P}(R_{2.2})$ as shown in Figure 12a, and break $C$ into a segment $C_1$ from $c_0$ to $c_1$, and a segment $C_2$ from $c_1$ to $c_2$. For $i = 0, 1, 2$, let $D_i$ be the direction of $C$ at $c_i$ so that $D_0 = (1, 1.1)$, $D_1 = (0, 1)$ and $D_2 = (-1, 1.1)$. If $z_i$, $z_{i+1}$, $\ldots$, $z_j$ are counter-clockwise adjacent vertices of $P$ where $z_i = V\big(P, (1, 0)\big)$ and $z_j = V(P, D_0)$, we start with the polygonal line obtained by connecting the points $c_0 + z_i$, $c_0 + z_{i+1}$, $\ldots$, $c_0 + z_j$. In this case

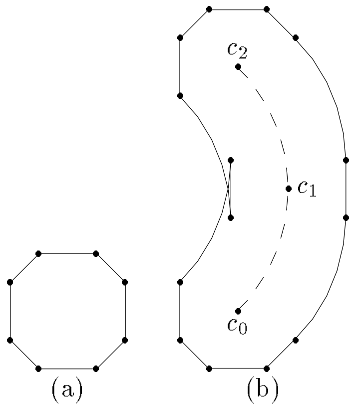



Figure 12: (a) The pen polygon $\mathcal{P}(R_{2.2})$; (b) the convolution tracing that describes the region $C + \mathcal{P}(R_{2.2})$, where $C$ is an 85° arc of a circle of diameter 6 as indicated by the dotted line.

$z_j = (2, -.5)$, so we then take the curve $C_1 + (2, -.5)$ up to $c_1 + (2, -.5)$. Since $V(P, D) = (2, .5)$ for directions $D$ between $D_1$ and $D_2$, we add a connecting segment from $c_1 + (2, -.5)$ to $c_1 + (2, .5)$ and continue with $C_2 + (2, .5)$. Finally, we complete the right side of the convolution tracing by adding a polygonal line from $c_2 + (2, .5)$ to $c_2 + V\bigl(P, (-1, 0)\bigr)$.

Instead of constructing the other half of the convolution tracing by starting at $c_2 + V\bigl(P, (-1, 0)\bigr)$ and going down to $c_0 + V\bigl(P, (1, 0)\bigr)$, it is more convenient to start from the bottom and use $-D_i$ in place of $D_i$ for $i = 0, 1, 2$: We start with the polygonal line from $c_0 + V\bigl(P, (1, 0)\bigr)$ to $c_0 + V(P, -D_0)$, taking the intermediate vertices of $P$ in clockwise order. Next comes $C_1 + V(P, -D_0)$, then a connecting segment from $c_1 + V(P, -D_0)$ to $c_1 + V(P, -D_2)$, followed by $C_2 + V(P, -D_2)$ and another polygonal line to $c_2 + V\bigl(P, (-1, 0)\bigr)$.

Given a curve $C$ that is strictly monotone in $y$, we can find the pixel centers in $C + P$ by using the convolution tracing to locate the left and right boundaries of $C + P$. After applying the *plot* routine to each of the segments in the right side of the convolution tracing, we can easily locate for each scanline $y = j$, the maximum integer $i$ such that $(i, j)$ is on or to the left of the right edge of $C + P$ at $y = j$. If more than one segment cuts $y = j$, the desired $i$ is the maximum over all relevant calls to *plot*. The left side of $C + P$ is obtained similarly except that we take the minimum instead of the maximum.

In the example of Figure 12, the right side of the convolution tracing is monotone in $y$, so no two segments can cut the same scanline; i.e., the dots that denote segment boundaries are encountered in order of increasing $y$. This is not true for the left side of the convolution tracing because of the connecting segment from $c_1 + V(P, -D_0)$ to $c_1 + V(P, -D_2)$. Such connecting segments whose vertical orientation is opposite that of the surrounding curve segments are what Guibas, Ramshaw, and Stolfi call *backward edges*. Their theory implies that backward edges cannot be necessary in order to describe the boundary of $C + P$ because they have an inverse effect in the rule that determines the correspondence between the convolution tracing and the number of "coats of paint" received: Ordinarily, when a scanline crosses a segment of the left side of the convolution tracing, the region to the right of the intersection gets one more coat of paint; for right-side segments, the region to the left of the intersection gets one more coat of paint. Since backward edges have the reverse effect, they cannot be necessary to delimit the boundary of $C + P$ because the number of coats of paint is zero to the left of the left edge of $C + P$, zero to the right of the right edge, and positive in between.

When $C$ is monotone in $y$, we rasterize $C + P$ by looking at the convolution tracing and applying *plot* to each curve segment and each connecting segment that is not a backward edge. For each scanline, we fill between the leftmost pixel that is to the right of some left-side segment and the rightmost pixel the is on or to the left of some right-side segment.

When $C$ is not monotone in $y$, we can break it at vertical extrema to yield *falls* $F_1, F_2, \ldots, F_m$.

Figure 13 shows how to process each fall separately, using a pen polygon stored in a global array $\bar{P}$. Let $k$ denote the number of vertices on or to the right of a vertical line through the middle of the pen polygon, and let these vertices be $\bar{P}[1]$, $\bar{P}[2]$, $\ldots$, $\bar{P}[k]$. For $i \le \lceil k/2 \rceil$, $\bar{P}[i]$ is the $i$th vertex in the output of *make_pen*; for $i > \lceil k/2 \rceil$, $\bar{P}[i]$ is the result of negating the $y$-component of $\bar{P}[k+1-i]$.

Figure 13 assumes that each fall is broken into segments by cutting whenever the curve direction passes the direction of one of the edges of the pen polygon. Thus for each segment $S$, there is an integer $oindex(S)$ such that we may take $V(P,D) = \pm\bar{P}[oindex(S)]$ whenever $D$ is the tangent direction for some point on $S$ and $P$ is the pen polygon represented by $\bar{P}$. The first and last segments in a fall $F$ are denoted *first_seg*$(F)$ and *last_seg*$(F)$; the successor to a segment $S$ is denoted *next_seg*$(S)$; and the endpoints of $S$ are denoted *first_pt*$(S)$ and *last_pt*$(S)$. Naturally *first_pt*$(F)$ for a fall $F$ refers to *first_pt*(*first_seg*$(F)$), etc.

The $L$ and $R$ arrays keep track of the leftmost and rightmost pixels in each scanline, and *fill_rows* turns on the appropriate pixels. The *ochange* procedure draws right-side connecting edges between $c+\bar{P}[j]$ and $c+\bar{P}[j']$, or left-side edges between $c-\bar{P}[j]$ and $c-\bar{P}[j']$. Since right-side and left-side edges have opposite vertical orientations, one side is omitted because it yields only backward edges. Lines 9 and 10 use *ochange* to start out the rasterization by drawing all necessary edges of $c+P$; lines 18 and 19 use *ochange* to finish the raterization by drawing edges of $c'+P$.

The code assumes that *plot* has a procedure parameter and that $plot(S,f)$ executes $f(\lfloor x \rfloor, j)$ for each point $(x,j)$ on the curve segment $S$ where $j \in \mathbb{Z}$. In an actual implementation, the desired actions would be executed directly by the plotting routine and there would be a special purpose plotting routine for short line segments with rational slopes. An example of such a routine is the variant of Bresenham's algorithm shown in Figure 14. It handles segments that start or stop at integer $y$-coordinates by treating them as though they contain the upper endpoint but not the lower. The loop invariant is that the segment contains a point $(\xi, j)$, where $i+\frac{r}{n} \le \xi < i+\frac{r+1}{n}$ and $0 \le r < n$.

Note that if $P$ is the pen polygon and $A$ is the vertical extreme between falls $F$ and $F'$, *do_fall*$(F)$ and *do_fall*$(F')$ each draw $k-1$ connecting segments from $A+P$. A substantial saving can be achieved if we omit edges of $A+P$ known to be interior to the region $C+P$ that is being rasterized. This is especially important in the common case shown in Figure 15b where $C$ is a smooth curve and $F$ and $F'$ are almost horizontal near $A$ so that there is no non-horizontal edge of $A+P$ on the boundary of $C+P$.

If there is a sharp corner at the vertical extreme, the curve should be thought of as reaching $A$ with some direction $D$ then turning either right or left to some new direction $D'$. Turning right corresponds to executing line 18 of *do_fall*$(F)$ and then line 10 of *do_fall*$(F')$; turning left corresponds to line 19 of *do_fall*$(F)$ and line 9 of *do_fall*$(F')$. For example, suppose we use $\mathcal{P}(R_4)$ with $k=6$ and vertices $\bar{P}[1]$, $\bar{P}[2]$, $\ldots$, $\bar{P}[6]$ as shown in Figure 15a. In Figure 15c, we turn right from $D=(5,4)$ to $D'=(5,-4)$ while line 18 draws a connecting segment from $A-\bar{P}[2]$ to $A-\bar{P}[1]$ and line 10 draws a segment from $A+\bar{P}[6]$ to $A+\bar{P}[5]$. If we go the long way around from $(5,4)$ to $(5,-4)$ via $(-1,0)$ as shown in Figure 15d, line 19 draws four segments from $A+\bar{P}[2]$ to $A+\bar{P}[6]$ and line 9 draws four segments from $A-\bar{P}[1]$ to $A-\bar{P}[5]$.

Thus we can optimize *do_fall* by introducing a global variable $l$ and replacing lines 18–20 with

> **if** $2j < k+1$ **then** $l \leftarrow 1$ **else** $l \leftarrow k$;
> $ochange(c', j, l, c'.y > c.y)$;
> **if** $t' > t$ **then** *fill_rows*$(t+1-n,\ t'-n)$ **else** *fill_rows*$(t'+1,\ t)$;

and lines 8–10 with

> **if** $t' > t$ **then** $clear(t+1, t')$ **else** $clear(t'+1-n, t-n)$;
> $ochange(c,\ k+1-l,\ j,\ c'.y > c.y)$;

The effect of the line "**if** $2j < k+1$ ..." is to decide to turn right if the fall ends at a positive slope. This simple test finds the best way to turn in common cases such as those shown in Figures 15b and 15c.

**procedure** $do\_curve(C)$;<br>
$n \leftarrow 2\bar{P}[k].y$;<br>
**for** each fall $F$ of $C$ **do** $do\_fall(F)$;

**procedure** $do\_fall(F)$;<br>
$c \leftarrow first\_pt(F)$; $c' \leftarrow last\_pt(F)$;<br>
$j \leftarrow oindex(first\_seg(F))$;<br>
$t \leftarrow \lfloor c.y + \frac{n}{2} \rfloor$; $t' \leftarrow \lfloor c'.y + \frac{n}{2} \rfloor$;<br>
**if** $t' > t$ **then** $clear(t + 1 - n, t')$ **else** $clear(t' + 1 - n, t)$;<br>
$ochange(c, 1, j, c'.y > c.y)$;<br>
$ochange(c, k, j, c'.y > c.y)$;<br>
**for** each segment $S$ in $F$<br>
**do** { $(dx, dy) \leftarrow 2\bar{P}[j]$;<br>
    $plot(S + \bar{P}[j], transition)$;<br>
    **if** $S \neq last\_seg(F)$<br>
    **then** { $j' \leftarrow oindex(next\_seg(S))$;<br>
        $ochange(last\_pt(S), j, j', c'.y > c.y)$;<br>
        $j \leftarrow j'$; } }<br>
$ochange(c', j, 1, c'.y > c.y)$;<br>
$ochange(c', j, k, c'.y > c.y)$;<br>
**if** $t' > t$ **then** $fill\_rows(t + 1 - n,\ t')$ **else** $fill\_rows(t' + 1 - n,\ t)$;

**procedure** $clear(y_1, y_2)$;<br>
**for** $j \leftarrow y_1,\ y_1 + 1,\ \ldots,\ y_2$ **do** { $L[j] \leftarrow \infty$; $R[j] \leftarrow -\infty$; }

**procedure** $fill\_rows(y_1, y_2)$;<br>
**for** $j \leftarrow y_1,\ y_1 + 1,\ \ldots,\ y_2$<br>
  **do** Turn on all pixels $(x, j)$, where $L[j] < x \leq R[j]$;

**procedure** $transition(x, y)$;<br>
$R[y] \leftarrow \max(R[y], x)$;<br>
$L[y - dy] \leftarrow \min(L[y - dy], x - dx)$;

**procedure** $rtrans(x, y)$; $R[y] \leftarrow \max(R[y], x)$;

**procedure** $ltrans(x, y)$; $L[y] \leftarrow \min(L[y], x)$;

**procedure** $ochange(c, j, j', up)$;<br>
**if** $j < j'$ **then for** $i \leftarrow j + 1,\ j + 2,\ \ldots,\ j'$<br>
        **do if** $up$ **then** $plot(\text{line\_seg}(c{+}\bar{P}[i{-}1], c{+}\bar{P}[i]),\ rtrans)$<br>
            **else** $plot(\text{line\_seg}(c{-}\bar{P}[i{-}1], c{-}\bar{P}[i]),\ ltrans)$;<br>
**if** $j > j'$ **then for** $i \leftarrow j,\ j - 1,\ \ldots,\ j' + 1$<br>
        **do if** $up$ **then** $plot(\text{line\_seg}(c{-}\bar{P}[i{-}1], c{-}\bar{P}[i]),\ ltrans)$<br>
            **else** $plot(\text{line\_seg}(c{+}\bar{P}[i{-}1], c{+}\bar{P}[i]),\ rtrans)$;

Figure 13: Routines for rasterizing curves. Variables $n$, $dx$ and $dy$ are assumed to be global.

**procedure** $plot\_seg(x, y, m, n, f)$;
$j \leftarrow 1 + \lfloor y \rfloor$;
$r \leftarrow \lfloor m(j - y) + nx \rfloor$;
$i \leftarrow \lfloor r/n \rfloor$; $r \leftarrow r - in$;
$a \leftarrow \lfloor m/n \rfloor$; $b \leftarrow m - an$;
**while** $j \leq \lfloor y + \frac{n}{2} \rfloor$
**do** { $f(i, j)$;
  $r \leftarrow r + b$; $i \leftarrow i + a$;
  **if** $r \geq n$ **then** { $r \leftarrow r - n$; $i \leftarrow i + 1$; }
  $j \leftarrow j + 1$; }

Figure 14: A version of *plot* that plots a line segment from $(x, y)$ to $(x+\frac{m}{2}, y+\frac{n}{2})$ as required by *ochange*.

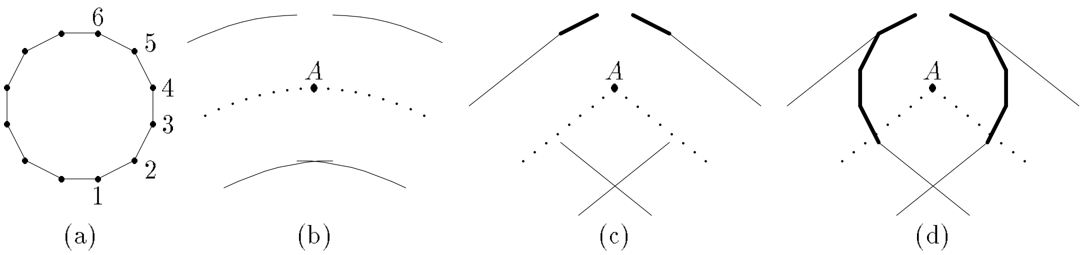



Figure 15: (a) The pen $\mathcal{P}(R_4)$ with $\bar{P}[1]$, $\bar{P}[2]$, ..., $\bar{P}[6]$ marked; (b) a smooth curve where no connecting segments are required for the vertical extreme at $A$; (c) and (d) two ways to plot connecting segments when there is a sharp corner at $A$. In b–d the curve $C$ is indicated by the dotted line, and solid lines give a portion of a convolution tracing that corresponds to $C + \mathcal{P}(R_4)$; the heavier lines are edges of $A + \mathcal{P}(R_4)$ that are actually plotted.

The other effects of these changes to *do_fall* are that we avoid applying *fill_rows* to scanlines occupied by $c' + P$, and we avoid applying *clear* to scanlines occupied by $c + P$. In other words for each scanline $y = j$, we continue collecting minima and maxima in $L[j]$ and $R[j]$ as long as the pen polygon intersects the scanline. This is necessary because, in the optimized program, a row of pixels can begin at a connecting segment from one fall and end at a connecting segment from the next fall.

Since the *clear* and *ochange* operations deleted from lines 8–10 are still needed before the first fall, these operations must be done just before line 3 in *do_curve*:

$c \leftarrow first\_pt(C)$; $c' \leftarrow last\_pt(first\_fall(C))$;
$t \leftarrow \lfloor c.y + \frac{1}{2}n \rfloor$;
$clear(t + 1 - n,\ t)$;
$ochange\big(c, 1,\ oindex(first\_seg(C)),\ c'.y > c.y\big)$;
$l \leftarrow 1$;

Similarly, operations removed from lines 18–20 need to be done just after line 3. The following code uses values of $c$, $c'$, and $t'$ computed by *do_fall*:

**for** each fall $F$ of $C$ **do** $do\_fall(F)$;
$ochange\big(last\_pt(C),\ oindex(last\_seg(C)),\ k + 1 - l,\ c'.y > c.y\big)$;
$fill\_rows(t' + 1 - n,\ t')$;

## 4. Analysis

We now analyze the time requirements of the actual rasterization algorithm and prove some theorems that reflect on the aesthetic qualities of the results. The rasterization algorithm works for any symmetrical pen polygon, but it is important to analyze the results of the pen polygon construction algorithm since time bounds and aesthetic qualities both depend on the pen polygon.

Suppose we wish to represent a curve $C$ with width $d$ on a device whose pixels are one unit high and have aspect ratio $a$. This involves applying *do_curve* to $X_a(C)$ and the pen $\mathcal{P}(R_{d,a})$, where $X_a$ scales $x$-coordinates by $a$. The running time depends on the time required for *plot* to process the necessary segments of $C$. This depends on the plotting algorithm and the family of curves involved, but it is at least linear in $A_y$, where $A_y$ is the total change in $y$ along $C$; i.e., $A_y$ is the arc length of $X_0(C)$.

Procedures *clear*, *fill_rows*, and *transition* do work proportional to $A_y + \Delta$, where $A_y + \Delta$ is the total number of times *fill_rows* processes a scanline. If we can show that $A_y$ dominates $\Delta$ and the overhead incurred elsewhere in the algorithm, it will be apparent that the overall running time is little more than that for rendering a one-pixel-wide version of $X_a(C)$. (The $O(A_y + \Delta)$ time bound for *fill_rows* depends on the output being given as run lengths. For bitmap output there is a term proportional to the number of pixels to be turned on, but this term is usually small in practice.)

The size of $\Delta$ and the number of operations counted as overhead are bounded by the number of segments into which $X_a(C)$ must be broken, the number of connecting segments plotted by *ochange*, and the total change in $y$ along these segments. There is at most one breakpoint between segments of $X_a(C)$ for each time the direction of $X_a(C)$ passes the horizontal or passes parallel to an edge of $\mathcal{P}(R_{d,a})$. Let us write this as $\alpha N(a,d)$, where $N(a,d)$ is the number of vertices of $\mathcal{P}(R_{d,a})$ and $\alpha$ depends on $a$, $d$ and $C$. In this notation the number of connecting segments is at most $(\alpha+1)N(a,d)$, and we can define $\alpha'$ so that the total change in $y$ along them is $(\alpha'+1)d$.

Although they depend on $a$ and $d$, the parameters $\alpha$ and $\alpha'$ are primarily measures of the complexity of $C$. They are typically on the order of unity and nearly equal. They can both be bounded in terms of the total curvature and number of inflections in $C$.

As is shown in the next section, $d$ dominates $N(a,d)$. Thus the total overhead is $O\big((1+\alpha+\alpha')d\big)$, and the main part of the algorithm requires $\Omega(A_y)$. In other words, the main part predominates when the vertical extent is significantly greater than the desired width. If the running time for *plot* really depends on $A_y$ instead of the true arc length, the algorithm is especially fast for nearly horizontal curves.

## 4.1. Properties of Pen Polygons

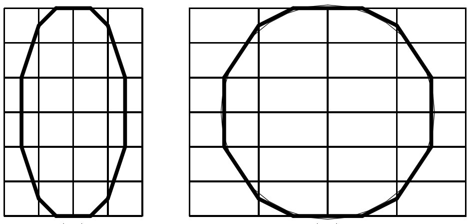
How well does $\mathcal{P}(R_{d,a})$ approximate $R_{d,a}$, and how many vertices does $\mathcal{P}(R_{d,a})$ have? The accuracy question is best studied in *device space* where pixels are one unit high and $1/a$ units wide as in Figure 16b. Since the algorithms of Sections 2 and 3 work in *pixel space* where pixels are one unit square as shown in Figure 16a, the pen polygon is based on the ellipse $R_{d,a}$ instead of the disk $R_d$. In order to produce lines $d$ units wide in device space, the pen should approximate $R_d$ when mapped back into device space. Thus we are interested in $E\big(X_{1/a}(\mathcal{P}(R_{d,a})), R_d\big)$, where $X_{1/a}$ scales $x$-coordinates by $\frac{1}{a}$ and

$$E(P, R_d) = \max_{(x,y)\in\text{boundary}(P)} \left| d - 2\sqrt{x^2+y^2}\,\right|.$$

Since $\mathcal{P}(R_{d,a})$ must have integer height and width, $E\big(X_{1/a}(\mathcal{P}(R_{d,a})), R_d\big)$ must be at least $\frac{1}{2}$ when $d \equiv \frac{1}{2}$ modulo one and the error must be at least $\frac{1}{2a}$ when $ad \equiv \frac{1}{2}$. Thus $\frac{1}{2}\max(1, \frac{1}{a})$ is the best error bound we can hope for. A proof would be rather tedious, but this bound appears to hold for almost all $a$ and $d$: (Note that we exclude small $d$ because the error approaches $\frac{\sqrt{5}-1}{2}$ when $d = \frac{1}{2}$ and $ad = n + \frac{1}{2} - \epsilon$ for small $\epsilon$ and large integer $n$.)

**Conjecture 1** *If* $\delta = \frac{1}{2}\max(1, \frac{1}{a})$ *and* $d \geq 2\delta$ *then* $E\big(X_{1/a}(\mathcal{P}(R_{d,a})), R_d\big) \leq \delta$.

We can verify the conjecture experimentally by computing all possible $\mathcal{P}(R_{d,a})$ for some range of $a$ and $d$. We do this by running *make_pen* and keeping track of the range $a$ and $d$ values that would yield the same rounding decisions. In this way, it has been determined that the conjecture holds for each of the 346,509 potentially different $\mathcal{P}(R_{d,a})$ where $d < 30$ and $ad < 30$.



(a) (b)

Figure 16: (a) The pen polygon $\mathcal{P}(R_{6.2,0.5})$ superimposed on a unit grid; (b) The same diagram in device space where the pixels are $1/a = 2$ units wide with $R_{6.2}$ superimposed for comparison.

Consider the function $N(a,d)$ that gives the number of vertices in $\mathcal{P}(R_{d,a})$. Since [7] proves that any pen polygon with perimeter $p$ has at most $O(p^{2/3})$ vertices, it follows immediately that $N(a,d) = O(d^{2/3})$ for fixed $a$. The basic idea is that each edge has a direction vector $(u,v) \in \mathbb{Z}^2$ with $\gcd(u,v) = 1$, and the length of that edge is a multiple of $\frac{1}{2}\sqrt{u^2+v^2}$. To minimize the perimeter for a given number of edges, the set of edge directions should be of the form

$$S_r = \{\, (u,v) \in \mathbb{Z}^2 \mid \gcd(u,v) = 1 \text{ and } u^2 + v^2 < r^2 \,\}$$

for some $r$. It is not hard to see that $S_r$ has $\Theta(r^2)$ edge directions and their total length is $\Theta(r^3)$, hence the $\frac{2}{3}$ power relationship.

Getting lower bounds on $N(a,d)$ and the dependence on $a$ requires new results not found in [7]. Let us look at some rough arguments for what asymptotic behavior to expect and then compare this with the actual behavior. For a pen $P$ with a given number of edges, we minimize the perimeter of $X_{1/a}(P)$ by selecting edge directions of the form

$$S_{r,a} = \{\, (u,v) \in \mathbb{Z}^2 \mid \gcd(u,v) = 1 \text{ and } u^2/a^2 + v^2 < r^2 \,\}.$$

Since $S_{r,a}$ contains $\Theta(ar^2)$ directions $(u,v)$ and the sum of all $\sqrt{u^2/a^2 + v^2 < r^2}$ is $\Theta(ar^3)$, it appears that if $X_{1/a}(P)$ has perimeter $p$ then it has $O(a^{1/3}p^{2/3})$ edges. This gives an $O(a^{1/3}d^{2/3})$ bound on $N(a,d)$.

Since a pen polygon can have no more than two vertices with the same $x$ coordinate or the same $y$ coordinate, $N(a,d)$ also has upper bounds of $O(ad)$ and $O(d)$ for an overall upper bound of $O\big(\bar{N}(a,d)\big)$, where

$$\bar{N}(a,d) = \min(d, ad, a^{1/3}d^{2/3}).$$

Rather than attempt to prove matching lower bounds, let us see how the actual vertex count $N(a,d)$ behaves for practical ranges of $a$ and $d$. By checking all $\mathcal{P}(R_{d,a})$ for $d$ and $ad$ both less than 30, we find that the ratio

$$\frac{N(a,d)}{\bar{N}(a,d)}$$

is bounded between 1.964 and 12. If we further restrict $min(d, ad) \geq 1$, the ratio varies in a seemingly random fashion over the interval $[1.964, 6.325]$. For $a = 1$ and $1 \leq d < 500$ the range is $[2.102, 5.545]$.

The time required to compute $\mathcal{P}(R_{d,a})$ is clearly $\Omega(N(a,d))$ but this bound may on the low side because we cannot be sure when an edge is created that it will be part of the final polygon. For instance we may have $N(a,d) = O(1)$ for arbitrarily large $a$, but *make_pen* requires at least $k$ iterations to achieve edge slopes $\bar{v}/\bar{u} \geq k$ or $\bar{v}/\bar{u} \leq \frac{1}{k}$. This gives runtime $\Omega(\max(a, \frac{1}{a}))$ if $\mathcal{P}(R_{d,a})$ has an edge slope near $\frac{1}{a}$. Thus it seems reasonable to expect runtime at least on the order of

$$\bar{T}(a,d) = \max(a, a^{-1}, a^{1/3}d^{2/3}).$$

Let $T(a,d)$ is the actual number of iterations in the main loop of *make_pen*. By evaluating all $\mathcal{P}(a,d)$ for $1.5 < d < 30$ and $1.5 < ad < 30$, we find that $T(a,d)/\bar{T}(a,d)$ varies in a pseudorandom fashion over the range $[0.542, 3.853]$. For $a = 1$ and $1 \leq d < 500$ the range is $[0.542, 3.331]$. Thus we conjecture that $\bar{T}(a, d)$ is the correct asymptotic time bound when $d > 1.5$ and $ad > 1.5$.

If we don't want to depend on conjecture we can obtain a simple $O((1+a)d)$ upper bound on the runtime of *make_pen* by looking at the sum of $e.l_1 + e.l_2$ for each edge $e$ in the data structures. This sum is initially $2 \cdot \mathrm{round}(d) + 2 \cdot \mathrm{round}(ad)$, and it gets reduced by at least one during each iteration of the **while** loop: one edge is deleted by *pop* if $\delta < 1$; otherwise the net effect of *lchop*, *rchop*, and the new edge is to reduce the sum by $\delta$.

### 4.2. Aesthetic Consequences of Pen Polygons

As explained in Section 2, pen polygons are designed to produce a uniform number of pixels per unit length when rasterizing a straight line of rational slope. It is natural to ask whether the rasterizations of curved lines have any similar properties.

If we have a curve $C$ and a desired width $d$, we take the corresponding curve $C' = X_a(C)$ in pixel space and rasterize the region $C' + \mathcal{P}(R_{d,a})$ as shown in Figure 17a. The goal is for the rasterization to have a uniform width when viewed in device space as shown in Figure 17b.

Figure 17a illustrates an *integer offset property* described in [7]: The region $C' + \mathcal{P}(R_{d,a})$ is bounded by curves $C' + z$ and $C' - z$ where $z$ is a vertex of $\mathcal{P}(R_{d,a})$ selected as explained in Section 3. The rasterization $R$ is bounded by curves $R^+$ and $R^+ - 2z$ shown in bold, and the integer offset property is that these curves are identical except shifted by the integer vector $2z$. The idea is that this gives predictable results if we measure the width at a point $A'$ on $C'$ by intersecting $R$ with a line $\ell_{A'}$ of direction $z$ through $A'$.

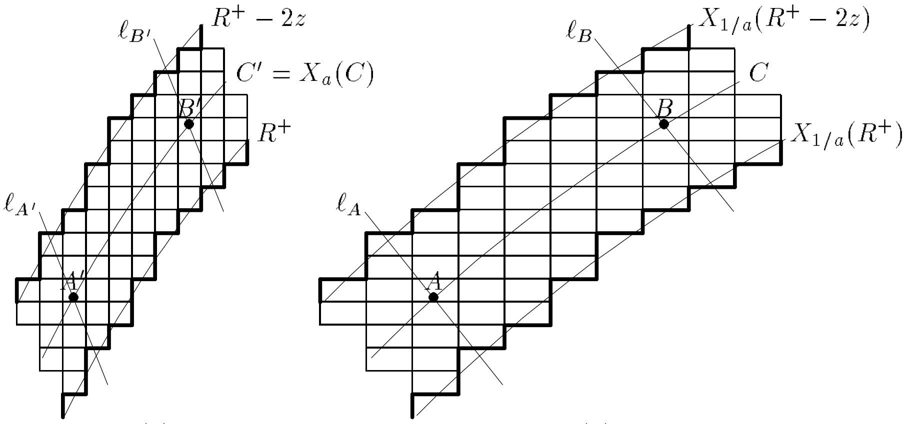



Figure 17: (a) The rasterization of a region $C' + \mathcal{P}(R_{6.2,0.5})$; (b) The same figure in device space based on aspect ratio $a = 0.5$. The rasterization is filled with pixel squares and has boundaries shown in bold.

More formally, assume that the pen polygon $P = \mathcal{P}(R_{d,a})$ has vertices

$$z_1, z_2, \ldots, z_k, -z_1, -z_2, \ldots, -z_k,$$

and the curve $C'$ can be broken into segments $C'_0, C'_1, \ldots, C'_m$ so that the region $C' + P$ is bounded by curve segments

$$C'_0 + z_{j_0},\ C'_0 - z_{j_0},\ C'_1 + z_{j_1},\ C'_1 - z_{j_1},\ \ldots,\ C'_m + z_{j_m},\ C'_m - z_{j_m}, \tag{2}$$

where $j_i = \mathit{oindex}(C'_i)$ for $i = 0, 1, \ldots, m$. Let points $A'$ and $B'$ lie on the same segment $C'_i$ so that none of the curve segments (2) other than $C'_i + z_{j_i}$ and $C'_i - z_{j_i}$ intersect the lines $\ell_{A'}$ and $\ell_{B'}$ through $A'$ and $B'$ with direction $z_{j_i}$. Under these conditions we say that $C' + P$ has *integer offset vector* $2z_{j_i}$ *between* $A'$ *and* $B'$.

The following uniformity theorem defines the width of a region $R'$ at a point $Q \in R'$ in some direction $D$ to be the length of the segment containing $Q$ formed by intersecting $R'$ with a line of direction $D$ through $Q$. Note that the width depends strongly on $D$ so we must make sure that the choice of $D$ is reasonable.

**Theorem 4.1** *Suppose $X_a$ maps points $A$ and $B$ on curve $C$ to points $A'$ and $B'$ on $C'$. If $C' + \mathcal{P}(R_{d,a})$ has rasterization $R$ and integer offset $2z$ between $A'$ and $B'$, then for any point $Q \in C$ between $A$ and $B$, the width of $X_{1/a}(R)$ at $Q$ in direction $X_{1/a}(z)$ is the length of $X_{1/a}(2z)$.*

Proof. Since $X_{1/a}(R)$ has boundaries $X_{1/a}(R^+)$ and $X_{1/a}(R^+) - X_{1/a}(2z)$ it only remains to show that they each intersect the line of direction $X_{1/a}(z)$ through $Q$ exactly once. It suffices to show that such a line intersects $C$ exactly once.

Since second and fourth quadrant edges of $\mathcal{P}(R_{d,a})$ cannot have negative slopes and first and third quadrant edges cannot have positive slopes, the slopes of $z$ and $C'$ cannot have the same sign. Thus a line of direction $z$ can cross $C'$ only once and therefore a line of direction $X_{1/a}(z)$ can cross $C$ only once. □

Roughly speaking, the theorem claims that the width is perfectly uniform and always equal to what one would expect by looking at $X_{1/a}(\mathcal{P}(R_{d,a}))$; i.e., the width is always the length of a diagonal of $X_{1/a}(\mathcal{P}(R_{d,a}))$ and conjecture 1 claims that this is close to $d$. In fact the theorem is not as strong as it appears to be because it says nothing about the direction in which the width is measured.

A more realistic way to measure the width of the rasterization is to choose two points on $C$ and measure the average width in the direction of their perpendicular bisector. Since it turns out that $\ell_A$ and $\ell_B$ are almost perpendicular to the line $AB$ in Figure 17b, the average width is essentially the area of $X_{1/a}(R)$ between lines $\ell_A$ and $\ell_B$ and divided by the distance from $A$ to $B$.

With this formulation it is possible to measure the error produced by the deviation of $\ell_A$ and $\ell_B$ from perpendicular to $AB$. The effect is inversely proportional to the distance between $A$ and $B$ and at most linear in the tangent of the deviation angle.

Figure 18 gives an example of a rasterization not based on a pen polygon where there is an integer offset vector of $(3,-1)$ but this is 62° away from perpendicular. The area of the rasterization between $\ell_1$ and $\ell_2$ is 5.91; dividing by the distance 3.04 gives an average width of 1.94. The average width of 1.01 between $\ell_2$ and $\ell_3$ reflects large variations in width allowed by the 62° deviation angle and the moderate distance over which the average is taken.

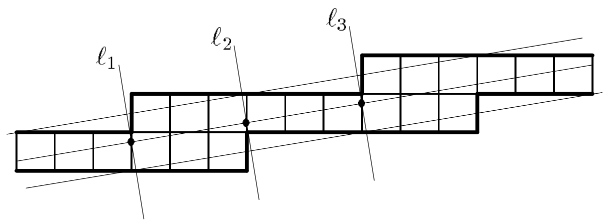



Figure 18: Computations for the width in direction $(1,-6)$ of the rasterization of $C + R_{1.48}$ with aspect $a = 1$, where $C$ is a line of slope $\frac{1}{6}$.

The angle of deviation from perpendicular is limited by the fact that $P' = X_{1/a}(\mathcal{P}(R_{d,a}))$ approximates the circle $R_d$. This is because $P'$ determines the function $2V(P', D)$ that gives the integer offset vector in terms of the curve direction. Enumeration of small pen polygons shows that the deviation angle is never more than 45° when $a = 1$. The theorem below shows that it is asymptotically $O\big((\frac{1+a}{d})^{1/2}\big)$ if conjecture 1 holds.

**Theorem 4.2** *Let $P' = X_{1/a}(\mathcal{P}(R_{d,a}))$, where $d$ is greater than some constant $\epsilon$. If opposite vertices of $P'$ are separated by at most $d + \epsilon$ and opposite edges are separated by at least $d - \epsilon$, then the* *maximum offset angle for $P'$ satisfies*

$$\tan\theta \le \frac{2\sqrt{\epsilon d}}{d-\epsilon}. \tag{3}$$

Proof. Figure 19 illustrates the construction. Let $Q_1Q_2$ be an edge of $P'$, and let $Q_{12}$ be the point on that edge closest to the origin $O$. Similarly, let $Q_2Q_3$ be the other edge incident on $Q_2$ and let $Q_{23}$ be the point of closest approach as shown in the figure. When $V(P', \bar{D}) = Q_2$, the angle of deviation from perpendicular is at most $Q_2OQ_{12}$ or $Q_2OQ_{23}$, whichever is greater. Since the length of $OQ_2$ is at most $\frac{1}{2}(d+\epsilon)$ and the lengths of $OQ_{12}$ and $OQ_{23}$ are at least $\frac{1}{2}(d-\epsilon)$, the lengths of $Q_2Q_{12}$ and $Q_2Q_{23}$ are at most $\frac{1}{2}\sqrt{(d+\epsilon)^2-(d-\epsilon)^2} = \sqrt{\epsilon d}$. Dividing by $\frac{1}{2}(d-\epsilon)$ yields (3). □

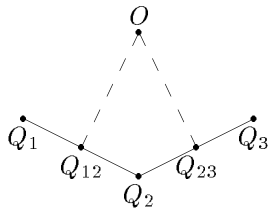



Figure 19: A construction for obtaining bounds on $\tan\theta$

While the deviation angle $\theta$ can approach the bound implied by conjecture 1 and the above theorem, the typical values are much better. When $d = 6.2$ and $a = 0.5$, the upper bound is $\theta \le 44.4°$. When $P' = X_2(\mathcal{P}(R_{6.2,0.5}))$ as shown in Figure 16b the maximum $\theta$ is 18.4° which occurs when $D$ is horizontal and $V(P', D) = (1, -3)$. When $V(P', D) = (2, -\frac{5}{2})$ as shown in Figure 17b, $\theta$ can be at most about 17.7° and the actual deviation from perpendicular to the line $AB$ in Figure 17b is only about 2°.

## 5. Conclusion

The two main algorithms discussed here are largely independent of one another. The *make_pen* routine discussed in Section 2 generates pen polygons that are appropriate for rendering lines of constant width, while the actual drawing routines in Section 3 work for any symmetrical pen polygon. Substituting one of the more complex pen polygon construction algorithms described in [7] would allow for calligraphic effects that generate lines of varying width. Alternatively, it is possible to avoid the pen construction algorithm for simple applications where the range of desired line widths is not too great and we can choose from a small repertoire of precomputed pen polygons such as those shown at the start of Section 2.

The algorithm can also be simplified when it is known in advance that the curve $C$ will be a straight line. It then becomes unnecessary to use an externally defined routine *plot* because everything can be done by *plot_seg* or a slight generalization thereof. In addition, the *transition* routine could write its results directly into the $L$ and $R$ arrays without taking maxima or minima. The discussion of convolution tracings in Section 3 becomes unnecessary because the region $X_a(C) + \mathcal{P}(R_{d,a})$ is a simple convex polygon.

In the general case of curved lines, the *plot* routine is likely to be significantly slower and more complicated than *plot_seg*. This routine has been left unspecified here because it is equivalent to the well-studied problem of rasterizing thin curved lines according to Freeman's rule. The speed of our algorithm comes from the fact that little time needs to be spent outside of the *plot* routine.

In addition to the speed advantages, we saw in Section 4.2 that we obtain better control over line width by rasterizing $X_a(C)+\mathcal{P}(R_{d,a})$ instead of $X_a(C)+R_{d,a}$. We showed that there is a way of measuring the average width of a rasterized image so that the width depends only on $X_{1/a}(\mathcal{P}(R_{d,a}))$ and the slope of $C$. Thus when the algorithm is asked to render a curve $C$ with width $d$, the width of the resulting image does not depend on where we look or where $C$ falls relative to the pixel grid. Further discussion of the aesthetic qualities of curves generated via pen polygons appears in [7], and other relevant issues appear in [6].

**References**

[1] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4:25–30, 1965.

[2] K. P. Fishkin and B. A. Barsky. Algorithms for brush movement in paint systems. In *Graphics Interface '84*, pages 9–16, 1984.

[3] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2):260–268, June 1961.

[4] H. Freeman and J. M. Glass. On the quantization of line-drawing data. *IEEE Transactions on Systems Science and Cybernetics*, 5(1):70–79, January 1969.

[5] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 100–111, 1983.

[6] John D. Hobby. Rasterization of nonparametric curves. *ACM Transactions on Graphics*, 9(3):262–277, July 1990.

[7] John Douglas Hobby. *Digitized Brush Trajectories.* PhD thesis, Dept. of Computer Science, Stanford University, 1985.

[8] B. K. P. Horn. Circle generators for display devices. *Computer Graphics and Image Processing*, 5:280–288, 1976.

[9] D. E. Knuth. *TEX and METAFONT, New Directions in Typesetting.* American Mathematical Society and Digital Press, Providence, Rhode Island, 1979.

[10] D. E. Knuth. *METAFONT the Program.* Addison Wesley, Reading, Massachusetts, 1986. Volume D of *Computers and Typesetting.*

[11] M. L. V. Pitteway. Algorithm for drawing ellipses or hyperbolæ with a digital plotter. *Computer Journal*, 10(3):282–289, November 1967.

[12] Vaughan Pratt. Techniques for conic splines. *Computer Graphics*, 19(3):151–159, July 1985.

[13] L. Ramshaw. Private communication, 1984.

[14] Tung Yun Mei. LCCD, a language for chinese character design. *Software—Practice and Experience*, 11:1273–1292, 1981.

[15] K. Turkowski. Anti-aliasing through the use of coordinate transformations. *ACM Transactions on Graphics*, 1(3):215–234, 1982.

[16] J. Van Aken and M. Novak. Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics*, 4(2):147–169, April 1985.

[17] J. T. Whitted. Anti-aliased line drawing using brush extrusion. *Computer Graphics*, 17(3):151–156, 1983.