Potrace: a polygon-based tracing algorithm

Peter Selinger

September 20, 2003

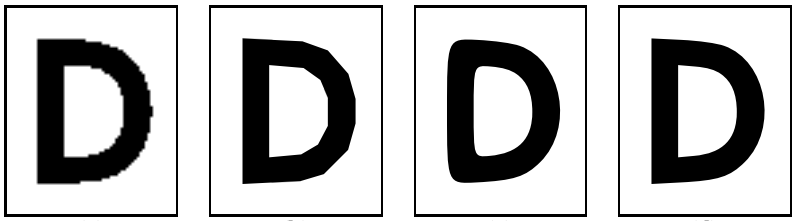
1 Introduction

Black-on-white images can be represented either as a bitmap or as a vector outline. A bitmap represents an image as a grid of black or white pixels. A vector outline describes an image via an algebraic description of its contours, typically in the form of Bezier curves. The advantage of representing an image as a vector outline is that it can be scaled to any size without loss of quality. Outline images are independent of the resolution of any particular output device. They are particularly popular in the description of fonts, which must be reproducible at many different sizes. Examples of outline font formats include PostScript Type 1 fonts, TrueType, and Metafont. On the other hand, most actual input and output devices, such as scanners, displays, and printers, ultimately produce or consume bitmaps. The process of converting a vector outline to a bitmap is called *rendering*. The converse process of turning bitmaps into outlines is called *tracing*.

It is clear that no tracing algorithm can be perfect in an absolute sense, as there are in general many possible outlines that can give rise to the same bitmap. The process of tracing cannot be used to generate information that is not already present. On the other hand, out of the many possible outlines that could give rise to a given bitmap, clearly some are more plausible or aesthetically pleasing than others. For example, a common way of rendering bitmaps at a high resolution is to draw each black pixel as a precise square, which gives rise to "jaggies" or staircase patterns. Clearly, jaggies are neither pleasant to look at, nor are they particularly plausible interpretations of the original image. There is probably no absolute measure of what constitutes a good tracing algorithm, but it seems clear that some algorithms give better results than others.

In this paper, we describe a tracing algorithm which is simple, efficient, and tends to produce excellent results. The algorithm is called *potrace*, which stands for *polygon tracer*. However, the output of the algorithm is not a polygon, but a smooth contour made from Bezier curves. The name of the algorithm derives from the fact that it uses polygons as an intermediate representation of images.

The potrace algorithm is designed to work well on high resolution images. Thus, a typical application is to produce a vector outline from a company or university logo that has been scanned at a high resolution. Another possible application is the conversion of bitmapped fonts to outline fonts, if the original bitmapped fonts are available at a high enough resolution. No tracing algorithm will work well on very small scales, such as



(a) (b) (c) (d)

Figure 1: Corner detection. (a) the original bitmap; (b) too many corners; (c) too few corners; (d) good corner detection.

bitmaps for a typical 10pt screen font rendered at 75dpi. However, it will do a decent job of tracing non-exact shapes, such as scanned handwriting or cartoon drawings, even at relatively moderate resolutions.

Any good tracing algorithm has to perform several functions. Two of these functions are to find the most plausible curve which approximates a given outline, and to detect corners. There is a tradeoff between these two goals. If too many corners are detected, the output will look like a polygon and will no longer be smooth. If too few corners are detected, the output will look smooth but too rounded. An example is shown in Figure 1.

Another important function performed by a tracing algorithm is to decide which features of the bitmapped image are relevant, and which features are artifacts of the scanning or rendering process. Those features which can be explained as artifacts should be filtered out completely, because if even a slight hint of these features remains, this can lead to visible imperfections in the output. Consider a straight line of positive, but very small, slope. When rendered as a bitmap, such a line will lead to a staircase pattern, where the individual steps of the stair could be far apart. No matter how far the steps are apart, the output should be a straight line, or else it will be visually annoying. This example also shows that tracing is not in general a local operation, i.e., it cannot be based on merely looking at fixed-size neighborhoods of a point.

Although the potrace algorithm is very efficient, it produces nicer output than other comparable algorithms. For instance, Figure 2 compares the output of potrace 1.0, with its standard settings, to that of autotrace 0.31.1, another freely available tracing program (see http://autotrace.sourceforge.net/). In addition to its superior graphical output, potrace also compares favorably to autotrace in terms of speed and file size: The bitmap in Figure 2 took potrace 0.27 seconds to process, compared to 1.69 seconds for autotrace. Potrace produces an EPS file of 15790 bytes, compared to 39788 bytes for autotrace.

2 Description of the potrace algorithm

The potrace algorithm transforms a bitmap into a vector outline in several steps. In the first step, the bitmap is decomposed into a number of paths, which form the boundaries

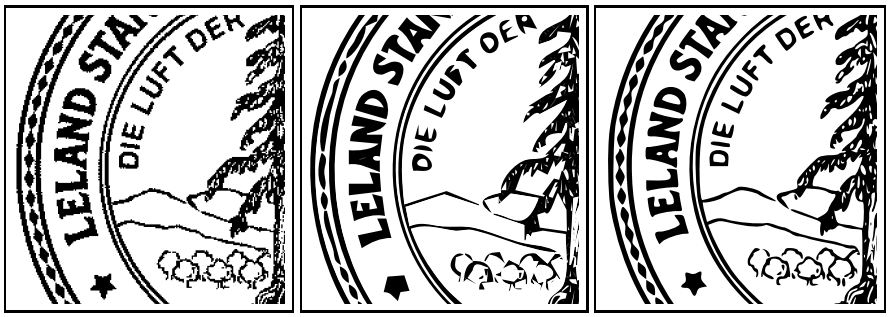


Figure 2: A detail from the seal of Stanford University; the original scanned image, left; the output of autotrace, center; the output of potrace, right.

between black and white areas. In the second step, each path is approximated by an optimal polygon. In the third step, each polygon is transformed into a smooth outline. In an optional fourth step, the resulting curve is optimized by joining consecutive Bezier curve segments together where this is possible. Finally, the output is generated in the required format. The following subsections describe each of these steps in more detail.

2.1 Paths

2.1.1 Path decomposition

We imagine our bitmapped image to be placed on a coordinate system such that the corners (and not the centers) of each pixel have integer coordinates. Let us further assume that the background color of the image is white, and the foreground color is black. By convention, the parts of the coordinate plane that lie outside the bitmap boundaries are assumed to be filled with white pixels.

We now construct a directed graph from our bitmap as follows. Let p be a point of integer coordinates; such a point is adjacent to four pixels. The point is called a *vertex* if the four pixels are not all of the same color. If v and w are vertices, we say that there is an *edge* from v to w if the Euclidean distance between v and w is 1, and if the straight line segment from v to w separates a black pixel from a white pixel, so that the black pixel is to its left and the white pixel is to its right when traveling in the direction from v to w. Let us call the resulting directed graph G with the vertices and edges just described.

A *path* is a sequence of vertices $\{v_0, \ldots, v_n\}$ such that there is an edge from v_i to v_{i+1}, for all $i = 0, \ldots, n-1$, and such that all these edges are distinct. A path is called *closed* if further, $v_n = v_0$. The *length* of a path is the number of edges in it, i.e., n. The goal of path decomposition is to decompose the graph G into closed paths, i.e., to find a set of closed paths in which each edge of G occurs exactly once.

Potrace uses the following straightforward method to decompose a bitmap into paths. Start by picking a pair of adjacent pixels of different color. This can be accomplished, for instance, by picking the leftmost black pixel in some row. The two chosen

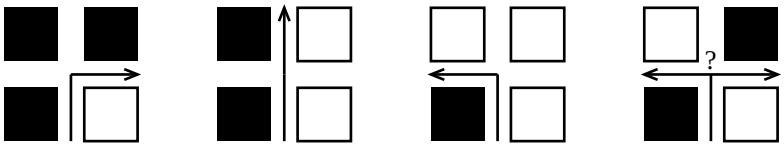


Figure 3: The path extension algorithm

pixels meet at an edge; we orient this edge so that the black pixel is to its left and the white pixel is to its right. This edge defines a path of length one. We then continue to extend this path in such a way that each new edge has a black pixel on its left and a white pixel on its right, relative to the direction of the path. In other words, we move along the edges between pixels, and each time we hit a corner, we either go straight or turn left or right, depending on the colors of the surrounding pixels as shown in Figure 3. We continue until we return to the vertex where we started, at which point we have defined a closed path.

Every time we have found a closed path, we remove it from the graph by inverting all the pixel colors in its interior. This defines a new bitmap, to which we apply the algorithm recursively until there are no more black pixels left. The result is a set of closed paths to be passed to the next phase of the potrace algorithm. The later phases of the potrace algorithm look at each path independently.

2.1.2 Turn policies

In the situation in Figure 3(d), we have a choice of whether to take a left turn or a right turn. This choice has no effect on the success or failure of the path decomposition algorithm, as we will end up with a set of closed paths either way. However, the choice does have an effect on the shape of the closed paths chosen.

In the potrace algorithm, the choice of whether to turn left or right is governed by a *turn policy*, which can be defined via the `--turnpolicy` command line option. Possible turn policies are: *left*, which always takes a left turn, *right*, which always takes a right turn, *black*, which prefers to connect black components, *white*, which prefers to connect white components, *minority*, which prefers to connect the color (black or white) which occurs least frequently within a given neighborhood of the current position, *majority*, which prefers to connect the color which occurs most frequently, and *random*, which makes a (more or less) random choice. The default turn policy is *minority*.

The reason that *black* and *white* are distinct turn policies from *right* and *left* is that some pixel colors may get inverted during the course of the path decomposition algorithm. The *black* and *white* policies look at the original pixel colors to determine the direction of the turn.

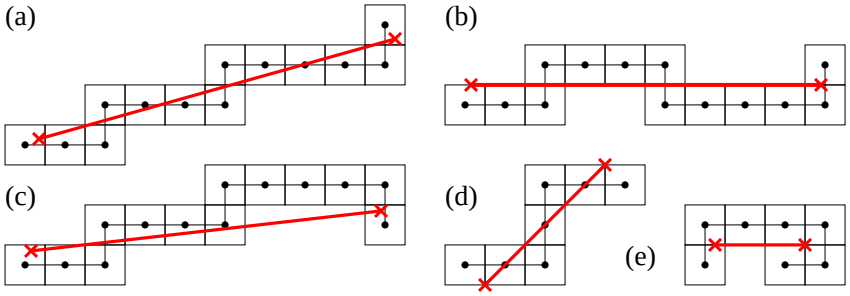


Figure 4: Examples of straight and non-straight paths. The vertices of the path are shown as dots, and their 1/2-neighborhoods are shown as squares. (a), (b), and (d) are straight, whereas (c) and (e) are not.

2.1.3 Despeckling

Despeckling can be performed by dropping all paths whose interior consists of fewer than t pixels, for a given parameter s. The parameter t can be set with the `--turdsize` command line option. The area of the interior of a path can be efficiently computed by the formula

$$Area = \int y\,dx = \int yx'\,dt.$$

2.2 Polygons

The second phase of the potrace algorithm has as its input a closed path as defined in Section 2.1. The output is an optimal polygon which approximates this path. We start by making precise what is meant by "optimal" and by "approximates".

2.2.1 Straight paths

Given two points $z_0 = (x_0, y_0)$ and $z_1 = (x_1, y_1)$ in the coordinate plane, not necessarily of integer coordinates, we define their *max-distance* to be $d(z_0, z_1) = \max\{|x_1 - x_0|, |y_1 - y_0|\}$. Thus, the set of points of max-distance at most 1/2 from the point $(1/2, 1/2)$ is just the pixel centered at $(1/2, 1/2)$.

For any two points a, b in the coordinate plane, let $\overline{ab}$ denote the straight line segment connecting a and b. Here a and b are not required to have integer coordinates.

Given a non-closed path $p = \{v_0, \ldots, v_n\}$ as in Section 2.1, we say that a line segment $\overline{ab}$ *approximates* the path if $d(v_0, a) \leqslant 1/2$, $d(v_n, b) \leqslant 1/2$, and for each $i = 1, \ldots, n-1$, there exists some point c_i on $\overline{ab}$ such that $d(v_i, c_i) \leqslant 1/2$.

For a path $p = \{v_0, \ldots, v_n\}$, we say the *direction* at index i is $v_{i+1} - v_i$, where $i = 0, \ldots, n-1$. There are four possible directions: $(0,1)$, $(1,0)$, $(0,-1)$, and $(-1,0)$. A path is called *straight* if it is approximated by some line segment, and not all four directions occur in p.

Figure 4 shows some examples of straight and not-straight line segments. Note that in this figure, the dots represent vertices in the path, which lie at the corners, not at

the centers, of the pixels of the original bitmap. The squares shown are not pixels, but rather neighborhoods of path points.

Figure 4(e) shows an example of a path which is not straight, although it is approximated by some line segment. This is because all four directions occur in this path.

It is clear from the definition that if a path is straight, then so are all its subpaths. In order to compute whether a given path is straight or not, we use the stronger fact that straightness is a *triplewise* property, in the following sense. Suppose that a given path $p = \{v_0, \ldots, v_n\}$ does not use all four directions. Then p is straight if and only if for all triples (i, j, k) of indices such that $0 \leqslant i < j < k \leqslant n$, there exists a point w on the straight line through v_i and v_k such that $d(v_j, w) \leqslant 1$. This observation gives rise to a naive straightness testing algorithm which is of cubic complexity in the worst case; it proceeds simply by testing the above property for all triples (i, j, k).

In the potrace implementation, we use an optimization which allows us to find *all* straight subpaths of a given closed path of length n in time $O(n^2)$ in the worst case. Briefly, the trick is to compute, for every pair (i, j), a *constraint* on the position of all future v_k's. If i is fixed and j is increasing, it suffices to check the constraint once for each j. Moreover, a constraint consist of at most two inequalities and can be updated and checked in constant time.

2.2.2 Polygons

Now consider a closed path $p = \{v_0, \ldots, v_n\}$. Recall that $v_n = v_0$, so that this path is of length n. Any pair of indices $i, j \in \{0, \ldots, n-1\}$ defines a subpath $p_{i,j}$, which is $v_i, \ldots, v_j$ if $i \leqslant j$, or $v_i, \ldots, v_{n-1}, v_0, \ldots, v_j$ if $j < i$. Let us write $j \ominus i$ for the *cyclic difference* between i and j, which is defined as $j \ominus i = j - i$ if $i \leqslant j$, and $j \ominus i = j - i + n$ if $j < i$. Thus, the length of the subpath $p_{i,j}$ is precisely $j \ominus i$. In the following discussion, we often assume tacitly that additions and subtractions are taken modulo n.

We now want to construct a polygon from the closed path p. We say that there is a *possible segment* from i to j if $j \ominus i \leqslant n - 3$ and the subpath $p_{i-1,j+1}$ is straight in the sense of the previous definition. In other words, a subpath corresponds to a possible segment if it can be extended by one point in either direction and still be straight. This peculiar "clipping" of a vertex from both ends of a straight path is important to the overall quality of the output of the potrace algorithm; without it, there would be strange behavior around the corners.

Note that any path of length 3 is straight in the sense of Section 2.2.1, thus it is guaranteed that there is always a possible segment from i to $i + 1$.

A *polygon*, for the purpose of this phase of the algorithm, is a sequence of indices $i_0 \leqslant i_1 \leqslant \ldots \leqslant i_{m-1}$ such that there is a possible segment from i_k to i_{k+1} for $k = 0, \ldots, m-2$, and from i_{m-1} to i_0. Figure 5 shows a path and two possible polygons for it.

Note that the polygon segments shown in Figure 5 do not actually have to approximate their corresponding subpaths in the sense of the red line segments of Figure 4. They simply represent the fact that an approximating line segment exists.

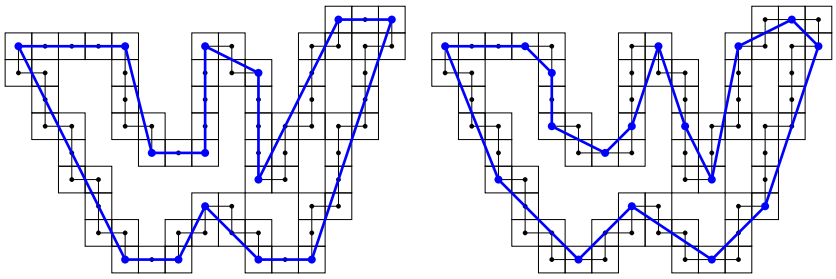


Figure 5: An optimal and a non-optimal polygon for a path

2.2.3 Penalties

Out of all possible polygons, we now want to find an optimal one. Our primary criterion for optimality is the number of segments: a polygon with fewer segments is considered more optimal than one with more segments. In Figure 5, the left polygon has 14 segments, whereas the right one has 17 segments. Thus, the left polygon is more optimal than the right one.

Among the polygons of the same number of segments, some are still more preferable than others. We associate to every possible segment a *penalty*. Given a possible segment from i to j, associate to it the straight line segment $\overline{v_i v_j}$ (shown in blue in Figure 5). The penalty associated with the segment is equal to the Euclidean length of $\overline{v_i v_j}$, times the standard deviation of the Euclidean distances of the path points from $\overline{v_i v_j}$. In symbols, the penalty is equal to

$$P_{i,j} = |v_j - v_i| \cdot \sqrt{\frac{1}{j \ominus i + 1} \sum_{k=i}^{j} dist(v_k, \overline{v_i v_j})^2},$$

where $dist(a, \overline{bc})$ denotes the Euclidean distance of a point from a straight line, and it is understood that the sum counts from i to j in a cyclic manner. In words, the further the path points stray from the segment, the greater the penalty.

The formula for $P_{i,j}$ was chosen because it can be computed efficiently; namely, let $(x,y) = v_j - v_i$ and $(\overline{x},\overline{y}) = (v_i + v_j)/2$. Then we have

$$P_{i,j} = \sqrt{cx^2 + 2bxy + ay^2},$$

where

$$\begin{aligned} a &= E(x_k^2) - 2\overline{x}E(x_k) + \overline{x}^2, \\ b &= E(x_k y_k) - \overline{x}E(x_k) - \overline{y}E(y_k) + \overline{xy}, \\ c &= E(y_k^2) - 2\overline{y}E(y_k) + \overline{y}^2. \end{aligned}$$

Here $E(x_k^2) = \frac{1}{j \ominus i + 1} \sum_{k=i}^{j} x_k^2$ is the expected value of x_k^2 for $k = i, \ldots, j$, and similarly for the other "E" notations.

Note that the sums can be computed ahead of time, by making a table of sums of the form $\sum_{k=0}^{j} q_k$, for each quantity q_k to be summed. After making such tables, which

takes time and space linear in the length of the given path, the above formula for $P_{i,j}$ can be computed in constant time for each given i and j.

2.2.4 Optimal polygon

We can regard the given closed path $p = \{v_0, \ldots, v_n\}$ as a directed graph with vertices $0, \ldots, n-1$, where there is an arrow from i to j if there is a possible segment from i to j. To each sequence $i_0 \to i_1 \to \ldots \to i_k$ of arrows, we can associate a penalty, which is an ordered pair (k, P), where k is the number of arrows in the sequence, and P is the sum of their numerical penalties as discussed in Section 2.2.3. Penalties are compared lexicographically, i.e., (k, P) is preferable to (k', P') if either $k < k'$, or $k = k'$ and $P < P'$.

In this way, the problem of finding an optimal polygon reduces to the problem of finding an optimal cycle in a directed graph. We use a variant of a standard graph-theoretic algorithm to solve this problem efficiently. Once the graph has been computed, an optimal cycle can be found in time $O(nm)$, where n is the size of the input path, and m is the length of the longest possible segment.

We remark that it is this optimization step which makes our algorithm non-local, because we have to consider an entire path at once; each part of the optimal polygon depends potentially on all the other parts. The previous phase, which computes a path from a bitmapped image, and the following phase, which transforms a polygon into a vector outline, are strictly local, in that they only look at a few adjacent points at a time.

2.3 From polygons to vector outlines

The final phase of the algorithm transforms the polygon obtained in the previous phase into a smooth vector outline. In a preliminary step, we adjust the position of the vertices of the polygon to fit the original bitmap as best as possible. In the main step, we calculate corners and curves based on the lengths of adjacent line segments and the angles between them.

2.3.1 Vertex adjustments

The output of the previous phase of the algorithm is a polygon $\{i_0, \ldots, i_{m-1}\}$ associated to a closed path $\{v_0, \ldots, v_n\}$. We refer to the indices $i_0, \ldots, i_{m-1}$, as well as to their associated points $v_{i_0}, \ldots, v_{i_{m-1}}$, as the *vertices* of the polygon. As our polygon is cyclic, we follow the usual convention of taking indices modulo m.

For the purpose of calculating penalties, we have placed the vertex i of the polygon precisely at the corresponding path point v_i, which is a point with integer coordinates in the coordinate plane (i.e., located at a meeting point of four pixels in the original bitmap). While this placement of vertices allowed us to calculate penalties efficiently, it is not necessarily the optimal arrangement. We now associate to each vertex i_k a point a_k in the coordinate plane, not necessarily of integer coordinates, such that a_k is near v_{i_k}, and such that, for any two consecutive vertices i_k and i_{k+1} of the polygon, the resulting line segment $\overline{a_k a_{k+1}}$ is reasonably close to the original subpath $v_{i_k}, \ldots, v_{i_{k+1}}$.

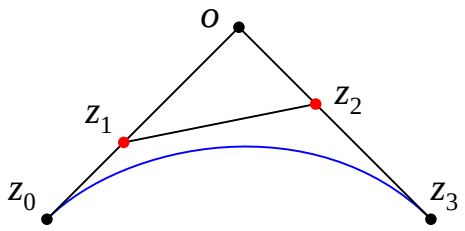


Figure 6: A typical Bezier curve

We use the following algorithm for placing the points a_k: for each consecutive pair of vertices i_k and i_{k+1}, calculate the straight line $L_{k,k+1}$ which optimally approximates the points $v_{i_k},\ldots,v_{i_{k+1}}$, in the sense that it minimizes the squares of their Euclidean distances to the line. Now if i_{k-1}, i_k, and i_{k+1} are consecutive vertices, then we ideally want to place a_k at the intersection of $L_{k-1,k}$ and $L_{k,k+1}$. However, we do not want a_k to be too far from the original vertex v_{i_k}. Thus, we let a_k be a point in the unit square with max-distance $d(a_k,v_{i_k}) \leqslant 1/2$ such that the sum of the square of the Euclidean distances from a_k to $L_{k-1,k}$ and $L_{k,k+1}$ is minimal. In particular, if the intersection of $L_{k-1,k}$ and $L_{k,k+1}$ lies in this unit square, then we place a_k at the intersection; else, we place it at a point near v_{i_k} which is "close" to the intersection.

Calculating a_k is easy, as it simply amounts to minimizing a quadratic function on a square. Also, the straight line $L_{k,k+1}$ is easily computed from the data points $v_{i_k},\ldots,v_{i_{k+1}}$ by using a standard method of "best fit": this line passes through the center of gravity $(E(x_k),E(y_k))$, where $k = i_k,\ldots,i_{k+1}$, and its slope is given by the eigenvector of the larger eigenvalue of the matrix $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$, where

$$\begin{aligned} a &= E(x_j^2) - E(x_j)^2, \\ b &= E(x_j y_j) - E(x_j)E(y_j), \\ c &= E(y_j^2) - E(y_j)^2. \end{aligned}$$

2.3.2 A class of Bezier curves

The purpose of this section is to make a simple, yet useful observation about Bezier curves. Recall that a Bezier curve is given by four control points z_0,z_1,z_2,z_3, and by the parametric equation $z = (1-t)^3 z_0 + 3t(1-t)^2 z_1 + 3t^2(1-t)z_2 + t^3 z_3$. For the purposes of our analysis, we restrict ourselves to the case where the straight lines through $z_0 z_1$ and through $z_3 z_2$ intersect at a point o (i.e., they are not parallel). Further, we restrict ourselves to curves that are *convex* and change direction by less than 180 degrees; this means that z_1 lies between z_0 and o, and that z_2 lies between z_3 and o. The situation is as shown in Figure 6. By a linear transformation of the coordinate system, we can assume that $z_0 = (-1,0)$, $z_3 = (1,0)$, and $o = (0,1)$. Any Bezier curve of this particular form is uniquely determined by two parameters $\alpha,\beta \in [0,1]$ such that $z_1 = (-1+\alpha,\alpha)$ and $z_2 = (1-\beta,\beta)$. Figure 7 gives an overview of the Bezier curves in this standardized

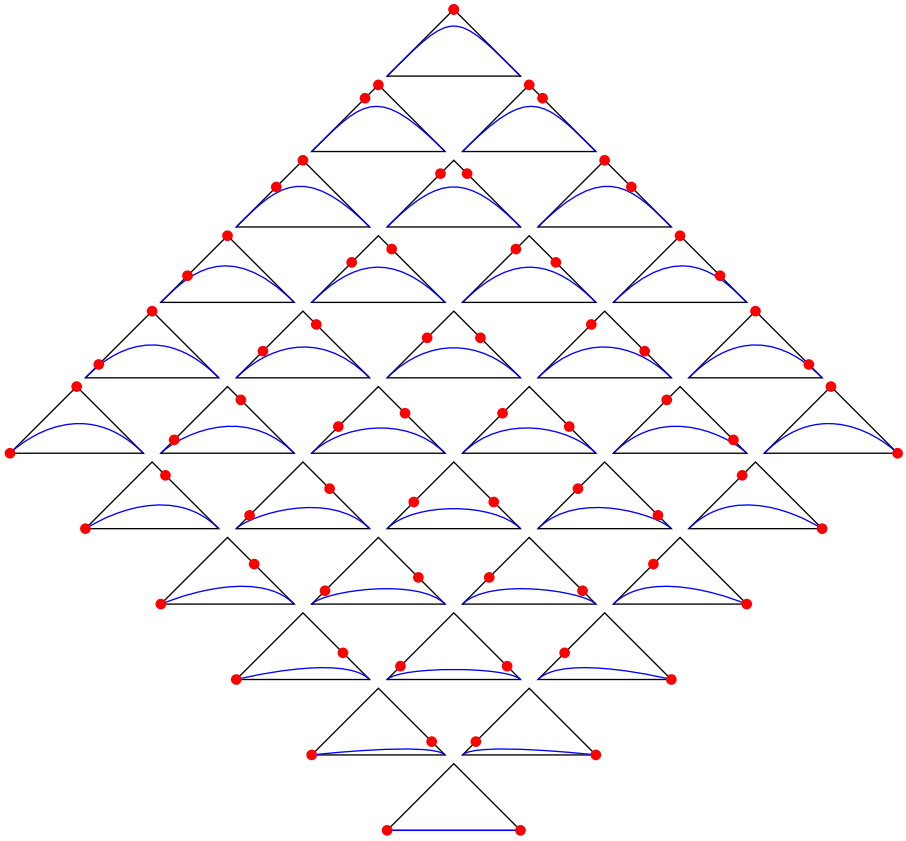


Figure 7: A 2-parameter family of Bezier curves

form for all values of α and β that are multiples of 0.2. In the illustration, the control points z_1 and z_2 are shown as red dots. We can see immediately from the illustration that the Bezier curves in any particular horizontal row are visually almost indistinguishable, except perhaps in the case when α or β are very close to 0. We will see that our algorithm never produces Bezier curves with α and β very small, so that we can ignore the latter possibility. It follows that we do not lose any interesting curves if we restrict ourselves to the case $\alpha = \beta$. This eliminates one degree of freedom from the set of possible Bezier curves that we need to consider, and thus it simplifies our task of finding optimal curves.

We should emphasize that we do not claim that all Bezier curves resemble the ones shown in Figure 7. Rather, this is the case *up to a linear transformation*. Thus, if z_0 and z_3 are given, there are two degrees of freedom in the placement of o, and one additional degree of freedom in the choice of α. By setting $\alpha = \beta$, a fourth degree of freedom has

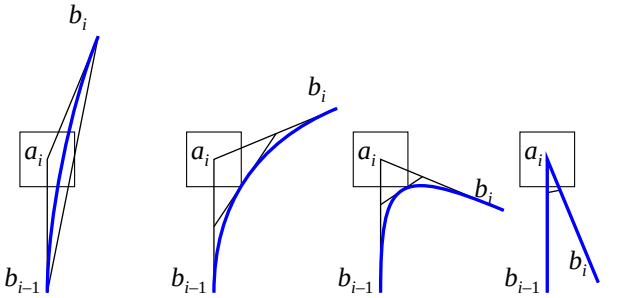


Figure 8: Corner detection and smoothing

been eliminated.

An interesting fact is that the area enclosed between a Bezier curve of the above form and the x-axis is equal to $\frac{3}{10}(2\alpha + 2\beta - \alpha\beta)$, or $\frac{3}{10}(4 - (2-\alpha)(2-\beta))$. From Figure 7, we find that two curves look very similar if they enclose areas of equal size. Thus, we may approximate any curve with parameters α and β by a new curve with equal parameters $\alpha' = \beta' = 2 - \sqrt{(2-\alpha)(2-\beta)}$.

Another interesting measure of a Bezier curve is the height of its highest point. In case $\alpha = \beta$, the highest point is reached when $t = 1/2$, and its y-coordinate is $3\alpha/4$.

2.3.3 Smoothing and corner analysis

The input to the last phase of the algorithm is the adjusted polygon from Section 2.3.1. Suppose the vertices of this polygon are $a_0, \ldots, a_{k-1}$. Let $b_0, \ldots, b_{k-1}$ be the midpoints of the edges of the polygon, i.e., $b_i = (a_i + a_{i+1})/2$. For each i, we now consider the corner $b_{i-1}..a_i..b_i$, and we decide whether to approximate it by a smooth curve, as shown by the blue line in Figure 8(a)–(c), or by a sharp angle, as shown in Figure 8(d).

We proceed as follows. First, we draw a unit square centered at the point a_i. Next, we find the line L_i which is parallel to $\overline{b_{i-1}b_i}$, which touches the square around a_i, and which is as close as possible to the line $b_{i-1}b_i$. Let c be the point where L_i intersects $\overline{b_{i-1}a_i}$, and let γ be the quotient of the lengths of $\overline{b_{i-1}c}$ and $\overline{b_{i-1}a_i}$. Let $\alpha = 4\gamma/3$ and consider the Bezier curve (of the kind discussed in Section 2.3.2) connecting b_{i-1} and b_i with parameter α. This curve is tangent to the three lines $b_{i-1}a_i$, L_i, and a_ib_i.

We use the parameter α just calculated to perform corner detection and to determine the final curve from b_{i-1} to b_i. There are two cases. If $\alpha \leqslant 1$, then we draw a smooth Bezier curve at this vertex, as shown in Figure 8(a)–(c). If $\alpha > 1$, there is no convex Bezier curve connecting b_{i-1} and b_i and tangent to L_i. In this case, we have detected a corner and we connect b_{i-1} and b_i via two straight line segments that meet at a_i, as shown in Figure 8(d).

Corner detection can be customized via the so-called *corner threshold parameter* $\alpha_{\max}$, which is configurable via the `--alphamax` command line option. If this param-

eter is set, then a vertex will be rounded if $\alpha \leqslant \alpha_{\max}$, and a corner if $\alpha > \alpha_{\max}$. Thus, smaller values of $\alpha_{\max}$ lead to more corners, as in Figure 1(b), and larger values lead to more rounded shapes, as in Figure 1(c). The default value is $\alpha_{\max} = 1$. If $\alpha_{\max} < 0$, then no smoothing is performed and the output of potrace is a polygon. If $\alpha_{\max} \geqslant 4/3$, then there will be no corners at all and the output is an everywhere smooth curve.

After corners have been detected, the value of α is further adjusted to be between 0.55 and 1. The lower bound $\alpha \geqslant 0.55$ was chosen to prevent the curve from becoming too "flat". Allowing $\alpha < 0.55$ often leads to strange looking images. The upper bound of 1 ensures that the resulting Bezier curve segment is convex.

The value $\alpha = 0.55$ was chosen because it tends to give a good approximation to a circle in case the input is a regular polygon. It was chosen to be close to the theoretical value

$$\alpha_0 = \frac{4}{3}(\sqrt{2}-1) \approx 0.552285,$$

which gives the best possible approximation by a Bezier curve segment to a quarter circle. More precisely, the Bezier curve with control points $z_0 = (1,0)$, $z_1 = (1,\alpha_0)$, $z_2 = (\alpha_0,1)$, and $z_3 = (0,1)$ lies between the unit circle and the circle of radius 1.00027253. Thus, this curve deviates from a true circle (of median radius) by less than 0.01363%. Although this approximation of a quarter circle by a Bezier curve segment is well-known, the exact bound is difficult to find in the literature; for instance, Faux and Pratt [1, p.134] falsely give this value as 0.13%, due to an apparent typographical error, whereas Knuth [2, p.14] gives it only as "less than 0.06%".

Note that our corner detection algorithm has the following property: Corners are favored both by sharp angles and by long segments. Thus, we detect a corner if two short segments meet at a very sharp angle, and also if two very long segments meet at a slight angle.

2.4 Curve optimization

The output of the previous phase of the potrace algorithm, after corner analysis and smoothing, is a curve consisting of Bezier curve segments and straight line segments. The resulting curve is very close to the final output of potrace. However, there is an optional last phase of the algorithm, the curve optimization phase, which attempts to further optimize the curve by joining adjacent Bezier curve segments together if this is possible. Curve optimization only makes very small changes to the shape of the final curve; small enough that the difference is not normally visible. However, the resulting curve consists of fewer segments, and thus can be represented more compactly in the final output of the program. If curve optimization is not desired, it can be disabled by giving the `--longcurve` command line option to potrace.

The curve optimization algorithm is based on a few simple ideas. First, we only attempt to join adjacent curved segments, never straight line segments or corners. Second, we only join adjacent curve segments which agree in convexity, i.e., they all curve to the right or all to the left. Third, we only join adjacent curve segments if the total change in direction is less than 179 degrees. (We do not quite allow 180 degrees, in order to avoid unbounded quantities in the computations below). This leaves us to consider a sequence of segments like the one shown in blue in Figure 9.

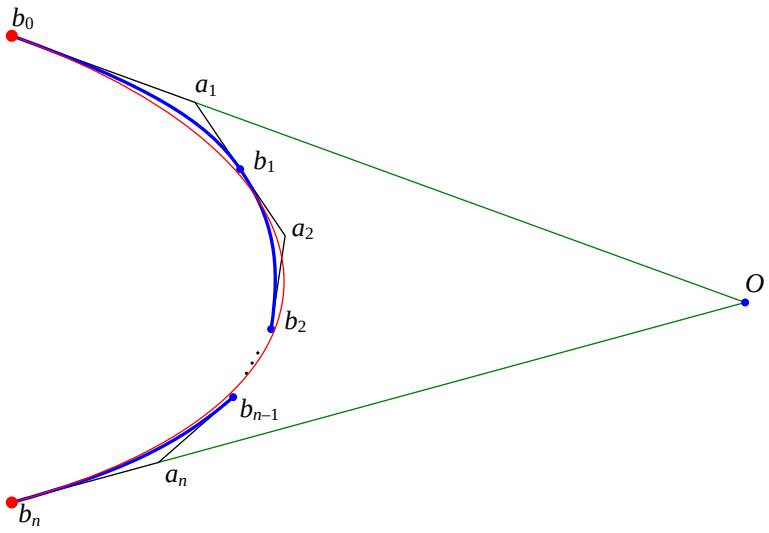


Figure 9: Curve optimization

The question is whether we can find a single Bezier curve from b_0 to b_n which approximates the given sequence of shorter Bezier curves. Suppose there was such a curve C. Clearly, C would have to be tangent to $\overline{b_0a_1}$ and $\overline{a_nb_n}$. We can thus find the point O where $\overline{b_0a_1}$ and $\overline{a_nb_n}$ intersect. Following our discussions in Section 2.3.2, this leaves only one degree of freedom in the curve to be considered, namely the parameter α. If we impose the further requirement that the area enclosed by the curve C should be equal to the total area enclosed by the original curve segments and the line $\overline{b_0b_n}$, then this uniquely determines the parameter α. Recall from Section 2.3.2 that the areas in question are easily calculated. This leaves us with a unique Bezier curve C which is a candidate for approximating the given segments. It is shown in red in Figure 9.

It remains to check whether C actually is an acceptable approximation to the given curve segments, and if yes, to assign it a numerical penalty. We do this by a simple tangency check. For each $i = 1,\ldots,n-1$, we find the point z_i on C where the tangent to C is parallel to $\overline{a_ia_{i+1}}$. We let d_i be the Euclidean distance of z_i to the line segment $\overline{a_ia_{i+1}}$. Further, for each $i = 1,\ldots,n$, we find the point z'_i on C where the tangent to C is parallel to $\overline{b_{i-1}b_i}$. We let d'_i be the Euclidean distance of z'_i to the line segment L_i define in Section 2.3.3, counted positive if z'_i is on the same side of L_i as a_i, and otherwise negative.

We say that the approximation is *acceptable* if all $d_i \leqslant \varepsilon$, $d'_i \geqslant -\varepsilon$, and the orthogonal projection of z_i onto the line $\overline{a_ia_{i+1}}$ lies between a_i and a_{i+1}. Here, the value ε is a constant called the *tolerance* of the curve optimization algorithm; it is pre-set to 0.2, and it can be altered via the `--opttolerance` option.

For an acceptable curve, we define its *penalty* to be the sum of the squares of all the distances d_i and d'_i. Finally, we use a standard graph-theoretic algorithm for

shortest path search to decompose a given sequence of curve segments into acceptable approximations, optimizing first the number of segments, then the total penalty.

2.5 Output generation

2.5.1 Scaling and rotation

The potrace algorithm has produced a family of curves, each of which consists of Bezier segments and straight line segments. The endpoints and control points of these segments are arbitrary points in the coordinate plane. Depending on the chosen backend and parameters, potrace now performs a linear transformation (to scale the image to the desired size, and possibly to rotate it).

2.5.2 Redundancy coding

When using one of the PostScript backends (PostScript or EPS), potrace uses a very compact numerical format to represent Bezier curves in the output. To do so, it takes advantage of redundancies in the curve parameters. In principle, 6 parameters are needed to describe each Bezier curve segment (1 endpoint and 2 control points). However, by eliminating redundancies in these parameters, potrace can encode each segment by using only 3 to 4 real numbers. One degree of freedom can be eliminated because we only use curves with $\alpha = \beta$, see Section 2.3.2. Another degree of freedom can be eliminated because the point b_i always lies on the line segment $\overline{a_i a_{i+1}}$, see Section 2.3.3. A third degree of freedom can often be eliminated because b_i is actually half way between a_i and a_{i+1} for those curve segments which were not affected by the curve optimization step of Section 2.4.

This redundancy coding of Bezier curves is only performed in the PostScript backend, because it takes advantage of the macro capabilities of the PostScript language. Redundancy coding can be turned off with the `--longcoding` option, resulting in longer, but more readable output.

2.5.3 Quantization

For most backends, the final coordinates, which are real numbers, are *quantized*, which means they are rounded to the closest 1/10 pixel. Thus, the number of decimal digits needed to represent each coordinate is reduced by effectively placing all control points on a very fine grid. The coordinates of the points can then be output as integers. The default quantization constant of 1/10 usually gives good results; however, it is configurable via the `--unit` command line option.

3 A complete example

A complete example of a run of the potrace algorithm is shown in Figure 10. Part (a) shows the original bitmap. In part (b), note how the default "minority" turn policy keeps the black outlines along the outside of the figure connected, while at the same time keeping the white outlines inside the figure's hair connected as well. Also note

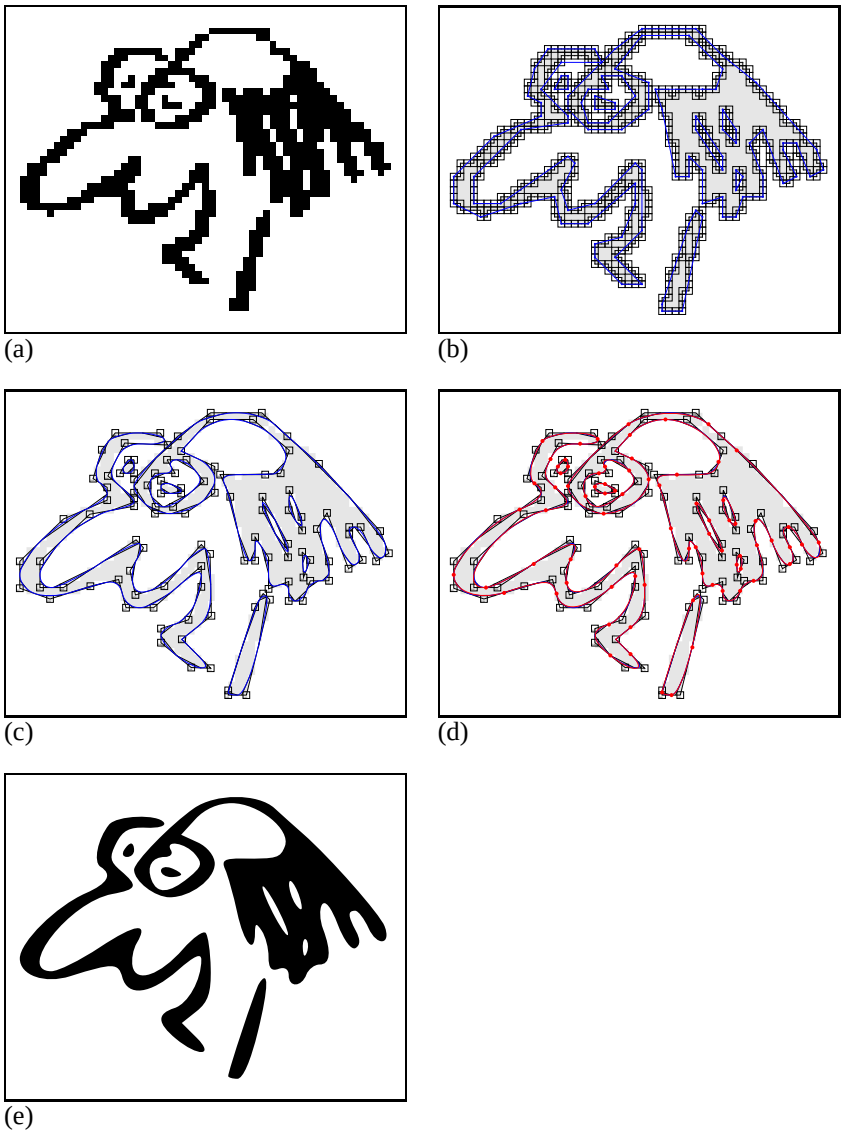


Figure 10: A complete example. (a) the original bitmap, (b) path decomposition and optimal polygon, (c) vertex adjustment, corner analysis, and smoothing, (d) curve optimization, (e) the final output.

that a speckle of size 1, inside the figure's hair, has been removed. Part (b) also shows the optimal polygon calculated for each path component. Part (c) shows the adjusted polygon vertices, relative to the underlying bitmap which is shown in grey. Each vertex is surrounded by its unit square. Also, the line segments L_i from Section 2.3.3 are shown, and the parameter α is written inside the unit square of each vertex; this is best seen by looking at the page at a very high magnification in Acrobat Reader. Corner analysis is performed at this step; note that for this particular bitmap, only very few corners are detected. Generally, corner analysis works better at higher resolutions. Smoothing is then performed; the resulting Bezier curve segments and line segments are shown in blue. Part (d) shows the result of curve optimization; the original curve is shown in blue, and the optimized curve is shown in red. Red dots indicate the new segment boundaries. Note that the number of segments has been reduced from 112 to 68, or by 40%. The final result of the algorithm is shown in Part (e).

Debugging output in the style of Figure 10(b)–(d) can be produced by giving the command line options `-d1` through `-d3` to potrace.

References

[1] I. D. Faux and M. J. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Horwood Series in Mathematics and its Applications, Editor: G. M. Bell. Ellis Horwood, New York, NY, USA, 1979.

[2] D. E. Knuth. *The METAFONTbook*, volume C of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.