1991

How to Compute Offsets Without Self-Intersection

Ching-Shoei Chiang

Christoph M. Hoffmann
Purdue University, cmh@cs.purdue.edu

Robert E. Lynch
Purdue University, rel@cs.purdue.edu

Report Number:
91-072

HOW TO COMPUTE OFFSETS WITHOUT SELF-INTERSECTION

Ching-Shoei Chiang
Christoph M. Hoffmann
Robert E. Lynch

CSD-TR-91-072
October 1991

How To Compute Offsets Without Self-Intersection*

Ching-Shoei Chiang
Christoph M. Hoffmann
Robert E. Lynch

Department of Computer Science
Purdue University
West Lafayette, Ind. 47907

October 11, 1991

Abstract

Traditional techniques for computing offsets are local in nature and lack good criteria for eliminating possible self-intersections of the offset. Methods based on integrating differential equations or on image processing do not lack such criteria, but seem to require constructing the solution in the ambient space, i.e., in one dimension larger than the offset. We investigate such methods.

1. INTRODUCTION

Given a plane curve C, its *offset* by a distance d is a curve $\mathrm{Off}(C,d)$ such that the points of $\mathrm{Off}(C,d)$ are at distance d from C. Similarly, given a surface S in 3-space, its offset by a distance d is a surface $\mathrm{Off}(S,d)$ such that the points of the offset surface are at distance d from S. These informal definitions can be made more precise in one of two ways, depending on whether the distance is measured locally or globally.

Denote the Euclidean distance of two points p and q with $d(p,q)$. Let C be a curve in $\mathbf{R}^2$, and p any point. Define the *global distance* of p to C by

$$\mathrm{dist}_g(p,C) = \inf\{d(p,q) \mid q \in C\}$$

*Work supported in part by ONR Contract N00014-90-J-1599, by NSF Grant CCR 86-19817, and by NSF Grant ECD 88-19817.

where $d(p,q)$ is the Euclidean distance of two points. For a smooth curve C, a *local distance* can be defined as

$$\mathrm{dist}_\ell(p,C) = d(p,q)$$

where q is on C and the line $\overline{pq}$ is perpendicular to the tangent of C at q. For most reasonable curves C these definitions make sense. The offsets for surfaces are defined analogously.

In the following, we assume that the curve C is smooth, except at finitely many points. Counting coinciding line segments as two points, moreover, we assume that any straight line intersects C in finitely many points. Likewise, we require surfaces to be smooth except at finitely many points or curves, and that any straight line intersects the surface in finitely many points, in the same sense. Note that piecewise algebraic curves and surfaces satisfy these requirements, assuming they consist of finitely many pieces. We call the curve or surface to be offset the *base curve* or *base surface*, respectively.

The local and global offsets can be defined as follows. In the global offset, the global distance is used, so that

$$\mathrm{Off}_g(C,d) = \{p \in \mathbf{R}^2 \mid \mathrm{dist}_g(p,C) = d\}$$

and

$$\mathrm{Off}_g(S,d) = \{p \in \mathbf{R}^3 \mid \mathrm{dist}_g(p,S) = d\}$$

For the local offset, complications arise from the fact that at each point the "correct" local distance is to be used. Here, it is convenient to think of geometric optics: The curve C or surface S is considered a continuum of point light sources emitting light at time $t = 0$. Then the local offset at distance d is the wave front of light at time $t_d = d/c$,[11] where c is the speed with which the front propagates. It is assumed that the front propagates with uniform speed and locally normal to the front. A physical analogue of the global offset is a grass fire front.[2] Briefly, the fire begins along C, and spreads burning uniformly. At time t, the fire front is the global offset at distance tc, where c is the speed with which the front propagates. Figure 1 illustrates the difference between the two offsets. Note that the offset has two real components, but only one of these is shown in the figure.

Local offsets are simple to define analytically, given a parametric or implicit representation of the curve or surface.[13,15,21] Moreover, the local offset of an algebraic curve or surface again is an algebraic curve or surface. Local offsets play a role in geometric optics.[11] Global offsets, in contrast, are a subset of the local offset, and are important in engineering applications.[21] Since they involve global distance computations, they are algorithmically more difficult to determine. Briefly, the accepted strategy for computing global offsets is

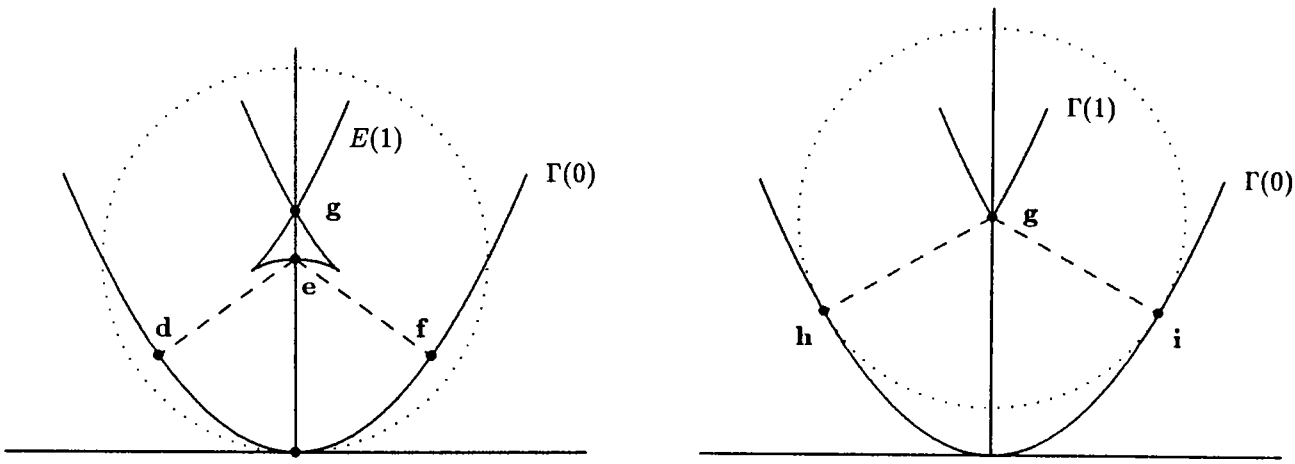

Figure 1: Local and global (one-sided) offset of a curve

> Determine the local offset. Then remove all those parts that are not at global distance d, by trimming certain branches or regions bounded by self-intersections.[21]

We propose as alternative the following approach:

> Integrate the global distance function up to distance d,[7] and extract the global offset as a level set of this function.

Our approach requires discretizing the curve or surface in the ambient space.

The locus of the self-intersections at which local offset regions border global ones can be characterized by the *medial-axis transform*, also called the *skeleton*.[2,19] Briefly, the skeleton is the locus of all those points p to which there exist at least two points on the base curve or surface at equal global distance. Computing the intersection of the skeleton with a d-offset is thus equivalent to determining those self-intersections at which the traditional approach will cut away nonglobal offset segments or regions.

Computing the skeleton is in general not simple. Algorithms have been given for planar curves composed from line segments and, in some cases, circular arcs;[25,27,28,32] and in 3-space for the surface of objects obtained from constructive solid geometry (CSG).[8,18] These algorithms are exact, but in 3-space they require surface representations using the dimensionality paradigm.[17,16] More complex geometric objects can be treated approximately,[1,29,34,35] for example, by constructing a Delaunay triangulation of a set of points that has been suitably chosen on the base curve or surface. The centers of the Delaunay triangles (tetrahedra) are then approximately on the skeleton and can be brought onto the skeleton by iterative computation. Self-intersections of the d-offset can also be characterized by certain systems of nonlinear equations,[18] but their solution

is usually no simpler and contains, furthermore, self-intersections that need not border global regions of the offset.

In view of the absence of an exact, simple, and efficient algorithm for determining the skeleton of complex geometries, we concentrate in this paper on directly evaluating the global distance function. All algorithms known to us discretize the ambient space, and then compute the global distance based on this discretization. Algorithms so evaluating distance are ordinarily insensitive to the problem dimension, and the algorithms we propose here are too. We have implemented two-dimensional versions, for offsetting curves, and only small changes would be needed for offsetting surfaces.

In Section 2, we review approaches to computing offsets devised by computer-aided geometric design (CAGD) and explain their local nature. In Section 3, we sketch the connection between the eikonal equation and the global distance function, and review the Euclidean distance transform in Section 4.

In Section 5, we reorganize the Euclidean distance transform, thereby deriving two algorithms that are better suited to computing the global offset. Both algorithms are further restructured in Section 6 adding interpolation and iteration, thus increasing the accuracy of the first two algorithms without changing the mesh size. Section 7 discusses the implementation.

2. CAGD APPROACH TO OFFSETS

Computer-aided geometric design (CAGD) has produced an extensive literature on offsets of curves and surfaces owing to the importance of the subject.[9,10,11,13,16,22,31] The majority of the methods construct approximate representations for local offsets of curves and investigate methods to trim them back to global offsets, or restrict applicability to those cases in which the local and global offsets coincide.

When approximating the offset, special properties of the base curve representation are customarily exploited. For example, offsets from (rational) parametric curves and surfaces can be constructed approximately by transforming the control points.[10] Exact offset representations that belong to the same class of curves or surfaces usually do not exist in the parametric case, except in very special cases.[14] Error estimates for the deviation of the approximation from the true offset can be given.[9] If the error is unacceptably large, the base curve or surface can be suitably subdivided and the offset approximated anew.

In Figure 1, the offset distance exceeds the smallest radius of curvature of the base curve. This always results in self-intersection,[20] and there are robust and efficient mathematical criteria that can be used to detect such situations.[9] In contrast, self-intersections such as the one shown in Figure 2 are harder to detect, and no efficient and comprehensive strategies for detecting them appear to be known. In consequence, it is not realistic to compute global offsets by trimming local offsets .

If the base curves and surfaces are algebraic, then the (local) offset is again

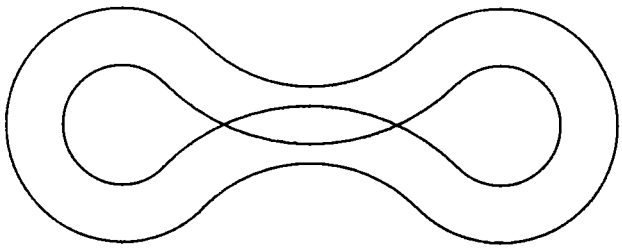

Figure 2: A Difficult Self-Intersection

algebraic, and in principle one can derive an implicit representation of the local offset using elimination techniques.[15] In practice, this approach is unattractive for several reasons:

- The symbolic computation required to derive the implicit form can be extremely expensive.[16]
- The algebraic degree of the offset can be very high, even for base curves and surfaces that have low algebraic degree.[12] Thus, subsequent computations with the implicit form could be numerically difficult.
- The implicit equation represents the local offset and offers no advantage for finding self-intersections.

Some of these difficulties are circumvented by the *dimensionality paradigm*[17,16] that represents offsets and other surfaces derived from geometric constraints as a system of nonlinear equations, thus side-stepping the expense of implicitizing. Again, the local offset is obtained and has to be trimmed if the global offset is wanted. Trimming could be based on global surface evaluation algorithms.[4] This approach is largely unexplored as yet.

3. EIKONAL EQUATION

Let S_x denote the partial derivative of a function S by x. The eikonal equation[6]

$$(S_x)^2 + (S_y)^2 = (S_t)^2 = \text{const} \qquad \text{and} \qquad (S_x)^2 + (S_y)^2 + (S_z)^2 = (S_t)^2 = \text{const}$$

describes a wave front that propagates with uniform speed. Courant and Hilbert[6], analyze the equation and its solution (18–19 and 84–94), explain the relation to geodesics (103–109 and 113–124), and the relation to wave propagation (124–131). For an informative discussion of simple examples see Strang,[33] 587–597.

For $t = 0$, let the level set $S(x, y, 0) = 0$ or $S(x, y, z, 0) = 0$ be a curve or surface D. Then the level set $S(x, y, t) = d$ or $S(x, y, z, t) = d$ is thus an offset of D at distance d. Note that for unit speed $d = \pm t$. The solution to the eikonal equation would become multi-valued at intersecting characteristics, i.e., along

the shocks in the solution. In those situations, the globally nearest solution is chosen, whence the eikonal equation describes the global offset.

The surface described by the eikonal equation is a ruled surface whose generators have fixed slope against the plane (or three-space) containing the boundary curve (or surface). The generators are the characteristics of the equation and project onto the normals of the boundary. The surface is developable and has been studied in geometry.[19] The intersection of the surface with parallel planes (parallel 3-spaces) is the global offset of the initial boundary. Our algorithms can thus be considered to be solvers for the eikonal equation. They are based on the geometric interpretation of the surface S.

4. EUCLIDEAN DISTANCE TRANSFORM

A number of algorithms have been described in the image processing literature that assign to each point in a regular grid its minimum distance to a subset of grid points.[3,7,26,30] In many cases, a distance metric has been used that is not Euclidean. Danielson[7] has proposed such a method for computing global Euclidean distance. We discretize the base curve as a set of grid points and use this method directly.

In Danielson's method two major passes are made over a two-dimensional grid, one from the top to the bottom, the other from the bottom to the top. In each major pass, two minor passes sweep every row, first from left to right, and then from right to left. There are two variants, one in which a five-point star configuration is used to update distances, the other using a nine-point star. At the end of the second major pass, each grid point has a distance assignment that is accurate to within $0.29h$ for the five-point star, and to within $0.076h$ for the nine-point star, where h is the grid spacing.

Let $p = (a, b)$ and $q = (a', b')$ be two points. The quantities $a - a'$ and $b - b'$ are called distance amplitudes. We store at each grid point two integers Δx and Δy that are the absolute value of the distance amplitudes to the grid point nearest to the curve. The true distance from that point is thus $\sqrt{(\Delta x)^2 + (\Delta y)^2}$. Using the squared distance, however, is better because then the algorithm can be implemented entirely in integer arithmetic.

In the top-to-bottom pass, the left sweep updates the (squared) distance of point (i, j) considering its neighbors in the row above and to the left. In the backwards sweep, the neighbor to the right is accounted for. The bottom-to-top pass is symmetric. When considering a neighbor, a distance to (i, j) is proposed that is based on the neighbor's distance, and updates the current distance of (i, j) if it is smaller. For example, considering the neighbor $(i-1, j)$, we proceed as follows:

> If $(\Delta x_{i-1,j} + 1)^2 + (\Delta y_{i-1,j})^2 < (\Delta x_{i,j})^2 + (\Delta y_{i,j})^2$ then update grid point (i, j) from grid point $(i - 1, j)$. That is, replace $\Delta x_{i,j}$ with $\Delta x_{i-1,j} + 1$ and $\Delta y_{i,j}$ with $\Delta y_{i-1,j}$.

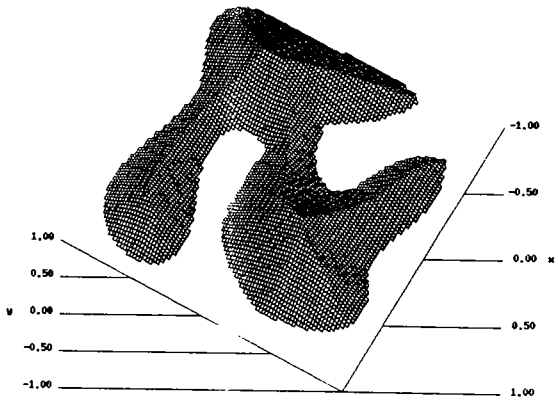

Figure 3: Euclidean distance transform on a grid 100×100

Updates from other neighbors are analogous.

Danielson's method works well when computing the distance of all points in a domain. An example of its output is shown in Figure 3, for a grid of 100×100. The method obviously generalizes to three dimensions.[3]

5. DISCRETE ALGORITHMS FOR GLOBAL OFFSETS

If the offset distance is small, that is, if the area enclosed by the base curve and its offset is small in comparison to the total domain area, then the fixed sweeps of Danielson's method result in more work than necessary. We therefore reorganize the algorithm so as to eliminate unnecessary computations.

We wish to reorganize the computation so that the grid points are processed by increasing distance from the boundary. If this can be done efficiently, then the offset of a closed curve can be computed in time that depends on the offset distance and the curve length, rather than on the entire enclosed area.

We process a grid point by assigning its distance. Based on this distance, a candidate distance is proposed for each of its neighbors, and each neighbor becomes a candidate for processing. There will be a queue for candidate grid points. Note that a grid point can appear several times in the queue, with proposed distances determined from different neighbors.

To process grid points by increasing distance, we add them to a priority queue indexed by the squared distance from the boundary. Initially, the discretized base curve grid points are added to this queue, with a zero distance key. Grid points to be added at a later time will have nonzero distance keys. Along with the distance key and the grid point coordinates, we store the distance amplitudes. The distance amplitudes Δx and Δy are signed, so we know the direction in which the nearest boundary point lies. By convention, the orientation of the vector $(\Delta x, \Delta y)$ is from a nearest boundary point (i_0, j_0) to the grid point $(i_0 + \Delta x, j_0 + \Delta y)$.

When removing the grid point (i, j) from the queue, it is processed as follows: Let (i, j) have distance amplitudes $(\Delta x, \Delta y)$ according to our sign conventions. Enter the distance of (i, j) into a matrix as element $M[i, j]$, unless $M[i, j]$ has already been assigned. Note that an entry already assigned cannot have a greater distance to the boundary, because all queue elements are processed by increasing distance. If $(\Delta x)^2 + (\Delta y)^2 < d^2$, where d is the offset distance, then add to the priority queue the entries $(i + r, j + s)$, with distance amplitudes $(\Delta x + r, \Delta y + s)$, where $r = -1, 0, 1$ and $s = -1, 0, 1$ and r and s are not both zero, and such that $(\Delta x + r)^2 + (\Delta y + s)^2 > (\Delta x)^2 + (\Delta y)^2$.

This scheme will still require time proportional to the matrix size unless we can initialize the entire matrix in constant time, independent of its size, access its entries in constant time, and determine for each accessed entry whether it has been reassigned since initialization. This can be done with standard techniques summarized in the appendix.

The work done by the algorithm to assign M amounts to examining each grid point (i, j) that is at a distance no greater than the offset distance and computing the distance amplitudes for its immediate neighbors. It is thus proportional to $N = A/h^2$, where A is the area enclosed by curve and offset, and h is the grid spacing. The priority queue updates require an additional logarithmic factor, so that the asymptotic complexity of the algorithm is $O(NlogN)$. We summarize the final algorithm.

Algorithm 1

1. Initialize M to *unassigned* and Q to consist of all curve points with distance amplitudes $(0, 0)$.
2. While Q is not empty, delete the next (i, j) from Q and perform step 3.
3. Let (a, b) be the distance amplitudes of (i, j). If $M[i, j]$ is already assigned, do nothing. Otherwise, assign the distance of (i, j) to $M[i, j]$. Enter into the queue Q all entries $(i + r, j + s)$, where $r = -1, 0, 1$ and $s = -1, 0, 1$, r and s are not both zero, and $(a + r)^2 + (b + s)^2 > a^2 + b^2$.

Note that Step 3 is slightly different for boundary points, unless we compute the offset on both sides.

The offset curve can be extracted from M by examining all its entries. Alternatively, let a frontier point be any grid point with at least one neighbor whose distance exceeds the offset distance d. Clearly the offset is near frontier points, and can be extracted by processing them and their neighbors, without examining other matrix entries. Frontier grid points can be registered when they are entered into M.

The offset could be left in discretized form, or an approximation of it can be constructed by, say, interpolation. Greater accuracy is obtained by iteratively refining frontier gridpoints to true offset points.[17]

Note that Algorithm 1 generalizes to offsetting surfaces in three dimensions. Here we have to work with a three-dimensional array and three distance amplitudes, $(\Delta x, \Delta y, \Delta z)$. The details are routine.

Processing grid points by increasing distance has been accomplished by a priority queue. Since distance amplitudes, and hence the squared distances are integer-valued when measured in multiples of the grid spacing, we can reduce the processing time by a factor of $\log N$ if we can find the next closest unprocessed grid point in constant time. This can be done.

Note that every distance key of interest is an integer between 1 and D^2, where $D = \lceil d \rceil$ is the offset distance measured in multiples of the grid spacing. We create an index array I, where position $I[k]$ is reserved for the distance key k. The entry $I[k]$ heads a list of unprocessed grid points, all at a conjectured squared distance k.

We add the grid point (i, j) with distance amplitudes (a, b) to the list headed by $I[a^2 + b^2]$. Now it is clear that we can process the grid points by first processing the list at $I[1]$, then the list at $I[2]$, and so on. Since distances are nondecreasing, we will never add a grid point to any list that is closer to the boundary than the points in the list we are currently processing. In all other respects, the algorithm remains the same.

It would seem that the array I is in size proportional to D^2. However, note that the distance of an unprocessed grid point to be added is not too far from the grid point currently processed. When processing a point at the squared distance u_1, a new point to be added is at a squared distance no greater than $u_2 = (a+1)^2 + (b+1)^2$, where $a^2 + b^2 = u_1$ and $a, b \leq D$. Furthermore, every point at a squared distance less than u_1 has already been processed. But

$$u_2 - u_1 \leq 4D + 2$$

so that at any time there are at most $4D + 2$ nonempty lists. Thus, it suffices to allocate for I an array of size $4D + 3$ as long as this space is used as a circular queue.[24] Therefore, the space requirements for the index vector I are proportional to the offset distance.

Note that we have to initialize entries in I when they are used for the first time, as well as the intervening, empty items of smaller squared distance. Initialization requires $O(D^2)$ steps, but D^2 is dominated by N. We summarize this algorithm as follows, without going into the details of managing the vector I as a circular queue.

Algorithm 2

1. Initialize M to *unassigned*, and I to *empty*. Add the curve points to the list $I[0]$.
2. For k from 0 to D^2 do Step 3.

3. For each entry in list k do the following. If $M[i,j]$ is already assigned, do nothing. Otherwise, assign the distance of (i,j) to $M[i,j]$. Add all entries $(i+r, j+s)$ to list $I[(a+r)^2 + (b+s)^2]$, where (a,b) are the distance amplitudes of (i,j), and r and s have been selected as in Algorithm 1.

The time required by Algorithm 2 is $O(N)$, and so improves Algorithm 1 by a factor of $\log N$.

6. ITERATION AND INTERPOLATION ALGORITHMS

Since Algorithms 1 and 2 compute distances from grid points only, the discretization of the base curve or surface into a set of grid points automatically introduces errors proportional to the grid spacing. We now diminish these errors using both iteration and interpolation. The main consequence of the changed approach is that integer arithmetic is no longer appropriate. Also, the work per grid point processed increases, but this increase can be offset by working with a coarser grid.

For each grid point, we will determine the following information: As before, we compute the signed distance amplitudes Δx and Δy and the squared distance $(\Delta x)^2 + (\Delta y)^2$. In addition, we identify the boundary segment or patch on which a nearest point lies. For parametric boundary elements we also record the parameter value(s). For grid points near the boundary, the additional data is determined first, say by linear interpolation of the intersection of the boundary with the grid lines. Then the distance amplitudes and the squared distance are computed from it. More precisely, with s_1 and s_2 the parameter values at the intersection with the grid lines, the secant is drawn and the perpendicular to it through the grid point is determined. Suppose the footpoint intersects the secant in the ratio $v : u$. Then the parameter is approximated by $s = s_1 + v(s_2 - s_1)/(u+v)$, and the distance amplitudes by Δx and Δy. See also Figure 4. The grid point data could be refined by Newton iteration.

When processing the neighbor grid points, either with Danielson's algorithm or with our Algorithms 1 or 2, the computations updating the distance amplitudes are as before. To increase accuracy, we can store the distance amplitude $mh + \delta$ as the pair (m, δ), since distance amplitudes increase by integer multiples of the grid spacing h. The additional data describing the nearest boundary point are copied. This part of the algorithm is the first phase.

Having so constructed an approximate distance of grid points, the frontier grid points are now "polished" using Newton iteration. In consequence, their distance to the boundary will be as accurate as possible. Then, the intersection of the offset with grid lines is determined by linear interpolation. Again, Newton iteration can be used to polish the point. The details are routine. These iteration and interpolation steps comprise the second phase of the algorithm.

When a grid or offset point lies close to a self-intersection, the different distances proposed by several neighbors will be close. This fact should be recorded,

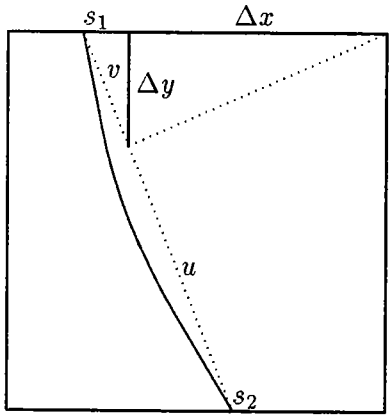

Figure 4: Grid Point Data Near Boundary

for the iteration in Phase 2 changes distances slightly, and so the point may ultimately lie closer to another part of the boundary. Thus, the originally proposed footpoint on the boundary may not be the true one. In this situation we have to iterate the distance from each footpoint associated with a neighbor who proposed approximately the same distance.

The algorithm so derived and based on the data structures of Algorithm 1 will be called Algorithm 3. Since the distance amplitudes are no longer integer multiples of the grid spacing, we cannot allocate unprocessed grid points into lists indexed by the squared distance in the simple way of Algorithm 2. But a more sophisticated scheme is possible: Suppose we allow a grid point at the greater distance u_1 to be processed before some grid points at the smaller distance u_2, as long as the difference in distance is not too big. More precisely, we must ensure that a grid point p is processed after all neighboring grid points have been processed that lie nearer to the boundary.

Now if q is a grid point at squared distance b from the boundary, and p is a neighboring grid point at squared distance a where $a > b$, then $a - b > h^2$. Therefore, we group all possible squared distances into intervals $[kh^2, (k + 1/2)h^2)$ and $[(k + 1/2)kh^2, (k + 1)h^2)$ and create lists of grid points for each distance group. Then a point in any group cannot generate a new grid point with a proposed distance in the same group. In consequence, Algorithm 2 can be modified using the interpolation scheme, at the cost of doubling the size of the list index I. The resulting algorithm is Algorithm 4.

7. EXPERIMENTAL RESULTS

We implemented Algorithms 1 and 2, and a version of the iteration and

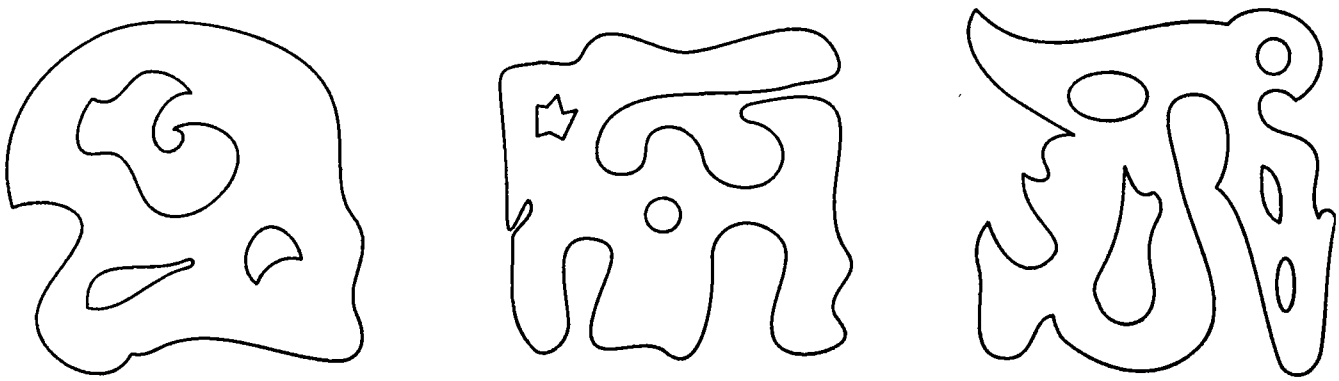

Figure 5: Sample domains: (A) left, (B) middle, (C) right.

interpolation method based on Danielson's algorithm. Input and output can be provided in several ways. We can interface to //ELLPACK[23] which provides us with graphical tools for defining 2D domains bounded by Bézier curves of arbitrary degree and with holes allowed. Alternatively, we can draw any shape with Framemaker, a commercial document preparation system. The shapes consist of cubic Bézier splines, ellipses, and line segments, and need not form domains.

We timed the execution of Danielson's method and of Algorithms 1 and 2 on the three sample shapes shown in Figure 5, using a grid size of 500×500. Danielson's method processed all 250,000 grid points, whereas Algorithms 1 and 2 processed only grid points up to the offset distance. The performance timed on a SUN SPARC 2 workstation is summarized in Table 1. All computations correspond to the 9-point star version. The results show that Algorithm 2 strictly improves Algorithm 1.

We determined for which value of N Algorithm 2 is as fast as Danielson's algorithm. On average, the algorithm requires $3.035 \cdot 10^{-4}$ seconds per grid point processed, whereas Danielson's method requires $0.684 \cdot 10^{-4}$ seconds per grid point processed. Thus, Algorithm 2 does 4-5 times as much work per grid point. In consequence, Algorithm 2 is better than Danielson's method for those offset distances at which the enclosed area is less than 22% of the area of the entire grid.

Adding iteration and interpolation impacted performance significantly. The algorithm was run on a 30×30 grid using Newton iteration to get accurate distance at grid points and using linear interpolation to extract the offset. This required 0.65 sec. Visual inspection of the output showed that the offset so determined was roughly equal in quality than the offset determined by Algorithm 2 on a 300×300 grid, which required in contrast 8.74 sec.

Danielson's Algorithm

Domain A: 16.85 sec	Domain B: 17.61 sec	Domain C: 16.86 sec

Algorithms 1 and 2

Domain	Offset dist	N	Time Alg 1 (sec)	Time Alg 2 (sec)
A	10	32,718	14.27	9.13
	20	62,648	28.80	19.85
	30	91,558	42.56	28.42
B	10	41,587	19.20	12.50
	20	78,932	36.91	25.78
	30	111,797	51.60	33.60
C	10	47,419	21.20	13.61
	20	87,918	41.17	27.59
	30	114,755	57.20	34.08

Table 1: Performance of Danielson's algorithm and of Algorithms 1 and 2

8. CONCLUSIONS

Detecting self-intersections in offsets is a problem that has both a mathematical and a combinatorial character. Some self-intersections can be detected based on local criteria applied to boundary elements, but others require evaluating the spatial relationship between unknown parts of the base curve, and this is difficult for the traditional offset algorithms in the literature.

We have addressed the problem by discretizing ambient space, and by "posting" a geometric datum in this common space. Self-intersection is thereby reduced to the problem of deciding whether a particular array element has already been assigned. The approach poses an exacting trade-off between the accuracy that is achieved and the memory that is required. Greater accuracy demands denser grids, and so larger arrays are needed, especially in three dimensions. Algorithms 1 and 2 use this approach.

The memory requirements of Algorithms 1 and 2 are dominated by the matrix M. The matrix could be implemented differently. For example, by storing each element in a balanced search tree only $O(N)$ memory is needed. This is a big improvement for small offset distances, but costs a logarithmic factor in the access time. The impact on the overall performance can be judged from the speed differentials of Algorithms 1 and 2.

Hybrid schemes that combine the discrete algorithm with interpolation and iteration allow using coarser grids without sacrificing accuracy, and save both time and space. Algorithms 3 and 4 are two examples. Our experience with the implementation suggests exploring adaptive schemes in which an initial coarse grid is refined selectively only in critical areas. At this time, we have no experience with this idea.

8. REFERENCES

1. C. Armstrong, T. Tam, D. Robinson, R. McKeag, and M. Price. Automatic generation of finite element meshes. In *SERC ACME Directorate Research Conference*, England, 1990.

2. H. Blum. A transformation for extracting new descriptors of shape. In W. Whaten-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, MA, 1967.

3. G. Borgefors. Distance transformations in arbitrary dimensions. *Comp. Vision, Graphics Image Processing*, 27:321–345, 1984.

4. J.-H. Chuang. *Surface Approximations in Geometric Modeling.* PhD thesis, Purdue University, Computer Science, 1990.

5. J.-H. Chuang and C. Hoffmann. Curvature computations on surfaces in n-space. Technical Report CER-90-34, Purdue University, Computer Science, 1990.

6. R. Courant and D. Hilbert. *Methods of mathematical physics, II.* Interscience, 1962.

7. P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

8. D. Dutta and C. Hoffmann. A geometric investigation of the skeleton of CSG objects. In *Proc. ASME Conf. Design Automation*, Chicago, 1990.

9. G. Elber and E. Cohen. Error bounded variable distance offset operatorand surfaces. *International J. of Computational Geometry and Applications*, 1:67–78, 1991.

10. R. T. Farouki. The approximation of nondegenerate offset surfaces. *Computer Aided Geometric Design*, 3:15–43, 1986.

11. R. T. Farouki and J.-C. A. Chastang. Evolving wavefronts as algebraic curves. Technical Report RC-16381, IBM Yorktown Heights, 1990.

12. R. T. Farouki and C. A. Neff. Algebraic properties of plane offset curves. *Computer Aided Geometric Design*, 7:101–128, 1990.

13. R. T. Farouki and C. A. Neff. Analytic properties of plane offset curves. *Computer Aided Geometric Design*, 7:83–100, 1990.

14. R. T. Farouki and T. Sakkalis. Pythagorean hodographs. Technical Report RC-15223, IBM Yorktown Heights, 1990.

15. C. M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, Cal., 1989.

16. C. M. Hoffmann. Algebraic and numerical techniques for offsets and blends. In S. Micchelli M. Gasca, W. Dahmen, editor, *Computations of Curves and Surfaces*, pages 499–528. Kluwer Academic, 1990.

17. C. M. Hoffmann. A dimensionality paradigm for surface interrogation. *CAGD*, 7:517–532, 1990.

18. C. M. Hoffmann. How to construct the skeleton of CSG objects. In A. Bowyer and J. Davenport, editors, *The Mathematics of Surfaces IV.* Oxford University Press, 1990.

19. C. M. Hoffmann and G. Vaněček. Fundamental techniques for geometric and solid modeling. In C. T. Leondes, editor, *Advances in Control and Dynamics*. Academic Press, 1991.

20. J. Hoschek. Offset curves in the plane. *Computer Aided Design*, 17:77–82, 1985.

21. J. Hoschek. *Grundlagen der Geometrischen Datenverarbeitung*. Teubner Verlag, Stuttgart, 1989.

22. J. Hoschek and N. Wissel. Optimal approximate conversion of splineapproximation of offset curves. *Computer Aided Design*, 20:475–483, 1988.

23. E. N. Houstis, T. S. Papatheodorou, and J. R. Rice. Parallel ellpack: An expert system for parallel processingpartial differential equations. *Math. Comp. Simulation Journal*, 31:487–508, 1989.

24. D. E. Knuth. *The Art of Computer Programming, Vol. 1*. Addison-Wesley, Reading, Mass., 1968.

25. D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Anal. and Mach. Intelligence*, PAMI-4:363–369, 1982.

26. U. Montanari. Continuous skeletons from digitized images. *JACM*, 16:534–549, 1969.

27. N. Patrikalakis and H. Gürsoy. Shape interrogation by medial axis transform. Technical Report Memo 90-2, MIT, Ocean Engr. Design Lab, 1990.

28. F. Preparata. The medial axis of a simple polygon. In *Proc. 6th Symp. Mathematical Foundations of Comp. Sci.*, pages 443–450, 1977.

29. M. Price, T. Tam, C. Armstrong, and R. McKeag. Computing the branch points of the voronoi diagram of a object using a point Delaunay triangulation algorithm. Draft manuscript, 1991.

30. A. Rosenfeld and J. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13:471–494, 1966.

31. J. Rossignac and A. Requicha. Offsetting operations in solid modeling. *CAGD*, 3:129–148, 1984.

32. V. Srinivasan and L. Nackman. Voronoi diagram of multiply connected polygonal domains. *IBM Journal of Research and Development*, 31:373–381, 1987.

33. G. Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, MA, 1986.

34. A. Wallis, D. Lavender, A. Bowyer, and J. Davenport. Computing voronoi diagrams of geometric models. In *IMPA Workshop on Geometric Modeling*, Rio de Janeiro, 1991.

35. X. Yu, J. A. Goldak, and L. Dong. Constructing 3D discrete medial axis. In *Proc. ACM Symp. Solid Modeling Found. and CAD/CAM Applic.*, pages 481–492, Austin, 1991.

APPENDIX: CONSTANT-TIME ARRAY INITIALIZATION

We solve the following problem: Given a matrix M of size $m \times n$, whose entries are random, (1) initialize every element of M to *unassigned* in constant time; (2) For any valid index (i, j), decide whether $M[i, j]$ is *unassigned*, in constant time; (3) Retrieve or store data in M in constant time. The solution to this problem is standard material in the theory of algorithms.

We use the matrix M itself, with each element a pointer into a stack S. An entry of the stack S is a pair consisting of a pointer to $M[i, j]$ and the value of $M[i, j]$. There is a variable T that records the top of the stack S. The pointers are implemented as integers, with $M[i, j]$ referred to by the value $(i - 1)n + j$. The pointer part of $S[j]$ is referred to as $S[j].p$, and the value part as $S[j].v$.

1. To initialize M, assign zero to T.
2. To test $M[i, j]$, retrieve its value k. If k is not in the range $1...r$, where r is the value of T, then $M[i, j]$ is unassigned. Otherwise, if $S[k].p \neq (i - 1)n + j$, then $M[i, j]$ is unassigned. Otherwise, $M[i, j]$ has the value $S[k].v$.
3. To retrieve the value of $M[i, j]$, do Step 2 above. To assign u to $M[i, j]$ we proceed as follows: If $M[i, j]$ is not unassigned, then assign u to $S[k].v$, where k is the value of $M[i, j]$. Otherwise, let k be the value of T; assign $k + 1$ to $M[i, j]$; assign $(i - 1)n + j$ to $S[k + 1].p$; assign u to $S[k + 1].v$; increment T by 1.

It is easy to see that this correctly implements all operations, and it is clear that each operation requires constant time, independent of the size of the array or the values of i and j. Moreover, it generalizes to arrays of any dimension.