Proceedings of the Seventh Canadian Conference on Computational Geometry,

Laval University, Quebec, August, 1995, pp. 55-60.

Practical Methods for Set Operations on Polygons using Exact Arithmetic

Victor J. Milenkovic*
University of Miami
Department of Math and Computer Science

Abstract

We present here numerical and combinatorial methods that permit the use of exact arithmetic in the construction of unions and intersection of polygonal regions. An argument is given that, even in an exact arithmetic system, rounding of coordinates is necessary. We also argue that it is natural and useful to round to a nonuniform grid, and we give methods for calculating the nearest grid point. The main result is a *shortest path* rounding algorithm that restores the combinatorial consistency of a polygon after its vertices have been rounded. This algorithm runs in linear time in the number of "near" vertex-edge pairs. It is optimal in the sense that it introduces the minimum combinatorial and geometric changes. We know of no other bounded-error rounding algorithm for nonuniform grids.

1 Introduction

There are many useful applications of set operations on polygons: graphics, maps, CAD/CAM, etc. There are efficient theoretical algorithms for performing the set operations: union, intersection, complement, difference. These are all based on the construction of an *arrangement* of line segments, where the line segments are the edges of the polygons. It is (or should be) widely known by now that one cannot naively implement these algorithms. Numerical issues inevitably crop up, limited precision or round-off error, and these can ruin the combinatorial correctness of the algorithm. Fortunately, there are nice ways to deal with these issues, either using exact or rounded arithmetic. There are some open questions, but these do not stand in the way of practical algorithms.

This paper focuses on numerical and combinatorial issues related to exact implementations. Usually, one assumes that the input vertices have integer coordinates and carries out all operations using rational arithmetic. This paper gives arguments for the following observations.

- Any practical modeling system for polygons must eventually do a significant amount of rounding of the coordinates of the vertices.
- It is often natural and useful to round each vertex to the nearest point in a *nonuniform* grid.

Of course, one cannot naively round the vertices to new locations because that might introduce intersections among the edges of the polygon (other than at their endpoints). To accomplish this rounding, we present a *shortest path combinatorial rounding* algorithm. This algorithm specializes the linear time algorithm for shortest path in a simple polygon of Guibas *et al.* [5]. The shortest path rounding algorithm has the following properties.

- It introduces the minimum possible *combinatorial* change in the polygon (assuming one can add and delete edges, but not vertices).
- It introduces the minimum possible *geometric* change in the polygon. The deviation of each edge is bounded by a constant–the maximum grid spacing.[1]
- It runs in linear time in the number of "near" vertex-edge pairs. A vertex is "near" an edge if it lies within the maximum grid spacing.

1.1 Related Work.

Greene and Yao [4] presented the first algorithm for rounding an arrangement of line segments to a grid. For a grid with m bits of

*This research was funded by the Textile/Clothing Technology Corporation from funds awarded to them by the Alfred P. Sloan Foundation and by NSF grants CCR-91-157993 and CCR-90-09272. Author's email address: vjm@cs.miami.edu.

[1] Introducing the minimum geometric change in the *absence* of combinatorial changes is NP-hard [8].

precision, the algorithm introduces $\Theta(m)$ extra vertices per edge. We presented a rounded arithmetic (not exact) algorithm for rounding a polygon to its set of vertices [10]. This algorithm introduces no new vertices, but it has possibly unbounded geometric error. In previous work presented to this conference, we showed how to round both sets of line segments in 2D [11] and sets of triangles in 3D [13] to a nonuniform grid with minimum error. The line segment algorithm introduces no new vertex locations, but the 3D algorithm does. The line segment algorithm uses a "shortest path" theorem which also appears in our work on rounded arithmetic algorithms [9, 14]. We also show how use this rounding algorithm in combination with a technique to reduce the amount of precision needed to construct an arrangement of lines or line segments (and round them to the integer grid) [12].

Hobby [6] presents a simple technique for rounding an arrangement of line segments to the integer grid. The basic idea is to "hook" each edge to every vertex whose "rounding cell" intersects the edge. (A "rounding cell" of a grid point is the set of points nearer to that grid point than any other.) This technique introduces no new vertices, runs in linear time, and introduces error at most equal to the grid spacing. However, it does not introduce the minimum possible combinatorial or geometric changes, and it does not work on a nonuniform grid.

There is no essential difference between the shortest path rounding technique described here and the one given in [11] and [12]. However, neither of those justified the use of nonuniform grids. Furthermore, neither gave the specialized version of Guibas *et al.*'s shortest path algorithm that is required for this application. This paper gives a very practical and easy to implement version of this algorithm which in turn yields a practical implementation of the shortest path rounding technique. We have been using the technique described in this paper in an exact polygon modeling system since 1992. We actually were aware of Hobby's "rounding cell" technique when we were implementing the modeling system. (We mentioned it briefly as a cheap alternative at the Canadian conference in 1989.) In fact, in 1992 we ran some comparisons with the shortest path rounding. Rounding is such a small part of the cost of the algorithm that there was no discernible difference in running time. The difference in combinatorial structure was noticeable but not drastic. In a test with many rotations, unions, and intersections, the rounding cell version had no more than 10 to 20 percent more vertices on its boundary than the optimal shortest path rounding version. Since the difference in cost is insignificant, we have always used shortest path rounding (which also works for nonuniform grids).

1.2 Outline.

Section 2 reviews the basics of performing set operations on polygonal regions. Section 3 gives the reasons why even an exact arithmetic system must eventually round the coordinates of its vertices. Section 4 explains why it is natural and useful to round to a nonuniform grid, and it gives methods for calculating the nearest grid point. Finally, Section 5 gives the actual shortest path rounding algorithm.

2 Set Operations on Polygons: Basic Algorithms

In order to perform set operations on polygons, it is first necessary to choose a representation. All that is really necessary is a) a set of vertices and b) a set of edges. Each vertex is a geometric point (x, y). Each edge is a pair of vertices (either pointers or indices into a vertex table). It is often useful to order the vertices of an edge ab so that the inside of a polygon is "to the left".

An exact representation would use integer or rational coordinates for the vertices. A floating point representation would use floating point coordinates. Actually, one often uses the "float" or "double" data type (in C) or the equivalent in other languages to represent integer coordinates. First of all, the "double" or "extended" data type offers more precision than the standard 32-bit integer data type. Also, on most advanced computers, double precision floating point computations are equal in speed or *faster* than integer computations.

A nice compromise is *homogeneous coordinates.* The homogeneous coordinate (x, y, w) represents the geometric point $(x/w, y/w)$. If x, y, w are integers, then the homogeneous representation is equivalent to an exact *rational* representation with the restriction that the two coordinates have the same common denominator w.

Given polygons A and B, taking the union or intersection of their interiors is just slightly more work than computing the arrangement of their line segment edges. There are many more recent algorithms for line arrangements, but the Bentley-Ottmann algorithm [1] is still very practical.

Running a line segment arrangement algorithm has the effect of computing all the intersections of edges of A with edges of B. Also, if a vertex of A is identical to a vertex of B or if it lies on an edge of B, the arrangement algorithm will also detect this. We say that A and B have been *accommodated* to each other: each edge has been appropriately subdivided by intersection with the edges and vertices of the other polygon. It is not difficult to modify the line segment arrangement algorithm to *tag* each edge ab of A as one of the following: a) interior to B; b) exterior to B; c) identical to an edge cd of B with the *same* orientation ($a = c$ and $b = d$); d) identical to a edge cd of B with the *opposite* orientation ($a = d$ and $b = c$). Each edge cd of B is similarly tagged.

Now suppose we want to create a polygon C whose interior is the intersection of the interiors of A and B. (This is called the *regularized* intersection of A and B.) We add to C a) a copy of each edge of A that is *interior* to B; b) a copy of each edge of B that is *interior* to A; c) a single copy of each pair of edges belonging to both A and B with the *same* orientation. To create a polygon D which is the union of A and B (plus their interiors), we add to D a) a copy of each edge of A that is *exterior* to B; b) a copy of each edge of B that is *exterior* to A, c) a single copy of each pair of edges belonging to both A and B with the *same* orientation.

2.1 Required Numerical Primitives. A number of numerical primitives are required to implement any line segment arrangement algorithm. We describe here how to implement these primitives for homogeneous coordinates.

First of all, it is helpful to let w be positive. If it is not for a particular vertex (x, y, w), replace this vertex by $(-x, -y, -w)$.

Primitive 1: Comparing x-coordinates. Vertex a has lesser x-coordinate than vertex b if and only if $a_x b_w - a_w b_x < 0$.

Primitive 2: Comparing a vertex to a line. Vertex a lies on the left side of line cd if and only if $[a, c, d] > 0$, where

$$[a, c, d] = \begin{vmatrix} a_x & a_y & a_w \\ c_x & c_y & c_w \\ d_x & d_y & d_w \end{vmatrix}.$$

Primitive 3: In homogeneous coordinates, the intersection point of lines ab and cd is

$$[b, c, d]a - [a, c, d]b,$$

where scalar multiplication and vector addition (subtraction) is carried out in the usual fashion. This intersection point lies on both *edges* ab and cd if and only if Primitive 2 tells indicates that a and b are opposite sides of cd and that c and d are on opposite sides of ab. Computing the intersection requires three times the input precision if $a_w = b_w = \cdots = f_w = 1$. In general, it requires four times the input precision.

Primitive 4: The intersection of lines ab and cd lies to the left of edge ef if and only if

$$[a, e, f][b, c, d] - [a, c, d][b, e, f] > 0.$$

Evaluating this expression requires four times the input precision if $a_w = b_w = \cdots = f_w = 1$. In general, it requires six times the input precision.

2.2 Other Issues. In the presence of degeneracies, the primitives take on zero values in the expected fashion. For instance, if a lies on line cd, then $[a, c, d] = 0$. Degeneracies can be handled by careful programming of special cases or by the use of symbolic perturbation methods.

3 The Need for Rounding

In the context of this paper, *rounding* takes on two forms: *combinatorial* and *numerical*. *Combinatorial rounding* modifies the set of edges, but not the positions of the vertices. *Numerical rounding* alters the positions of the vertices. Usually *numerical rounding* creates the need for some *combinatorial rounding* in order to maintain a consistent (non-self-intersecting) polygon. Just as there are two types of rounding, there are two *reasons* for rounding: combinatorial and numerical. We consider here examples of these types of reasons.

3.1 Combinatorial Reasons for Rounding. In a drawing system or any system that models physical reality, it is often useful vertices to have *gravity*. Hence, if vertex a is close enough to edge cd, it may be necessary to split cd into ca and ad. If this is not done, it becomes very difficult to model a contact between vertex a of polygon A and edge cd of polygon B. If the coordinates are all integers and if $a_w = c_w = d_w = 1$, then a is very unlikely to lie on cd. In fact, if $c_x - d_x$ and $c_y - d_y$ are coprime, then a cannot lie on cd unless $a = c$ or $a = d$.

Suppose we wish to put polygon A in simultaneous contact with polygons B and C. This can be done by exact rational translation. However, each coordinate of the translated copy of A will require a representation with about three times the input precision. For practical purposes, it is much less expensive to move A within one unit of B and C and then "attach" the appropriate edges of B and C to A.

Please note that replacing cd by ca and ad might cause ca or ad to "run into" other vertices. In other words, some vertex e lies on one side of cd but the other side of ca. In that case, one must apply the rounding algorithm of Section 5.

3.2 Numerical Reasons for Rounding. One might think that intersecting an arbitrary number of polygons might cause the precision requirements to grow without bound. Actually, this is not the case. Using the primitives of Section 2.1, one can compute the combinatorial structure of the intersection using only a four-fold increase in precision. Representing the vertices requires a six-fold increase. The trick is to use the "original" endpoints of an edge wherever possible. For example, if $c'd'$ is a subedge of cd, then instead of computing the sign of $[a, c', d']$, compute the sign of $[a, c, d]$.

Unfortunately, some common operations spoil this scheme. For example, if we compute the convex hull of a derived polygon, then the convex hull might contain new edges joining derived vertices. These new edges are not part of some "original" edge. Triangulating a polygon (and extracting the triangles as new polygons) also has the same effect. Many applications use either convex hulls or triangulations.

Canny *et al.* [2] describe a scheme for carrying out exact *rotations* of points in two dimensions. Only a dense set of rotation matrices have rational coefficients, and therefore it is necessary to approximate the rotation angle. Under their scheme, an approximate rotation by θ with m bits of accuracy ($\theta(1 \pm 2^{-m})$) increases the precision of the homogeneous coordinates of A by m bits.

We conjecture that any polygon modeling system that provides set operations, Euclidean transformations, scaling, convex hulls, and triangulation cannot avoid an *exponential* growth in the number of bits of precision as a function of the number of operations.

Thus, truly useful and practical exact arithmetic modeling systems require that coordinates be rounded to lower precision. If vertex a is rounded to new position a', then edge $a'b$ might "run into" some other vertex c. In other words, c lies on opposite sides of ab and $a'b$. This is another instance in which it is necessary to apply the shortest path rounding algorithm of Section 5.

4 Nonuniform Grids: Reasons and Methods

A *uniform grid* is the cartesian product of two uniformly spaced discrete sets, such as the integers. Thus the integer grid is a uniform grid. A *nonuniform grid* is a cartesian product of nonuniformly spaced discrete sets. On any modern computer, the set of representable floating point numbers is a nonuniform set of rational numbers. The set decreases in density with distance from the origin. Thus the set of floating point geometric points is a nonuniform grid.

The integer grid appears at first glance to be the natural choice for exact arithmetic. We will first argue that it is more natural to use a nonuniform set of rational numbers for the coordinates. Secondly, we will give algorithms based on continued fractions or basis reduction to calculate these coordinates.

4.1 Nonuniform Grids: Reasons. Floating point is not synonymous with numerical rounding. There is no reason a modeling system cannot use an *exact* floating point representation. The next section summarizes how this can be done. Computer designers chose floating point as a numerical representation for a number of very important and practical reasons. Chief among these is that relative (rather than absolute) accuracy is independent of scale. Many of these reasons also hold for polygon modeling and its applications. Therefore, one very well might choose a nonuniform floating point representation for polygons.

Suppose we do not use a floating point representation but instead use a uniform integer grid for the input vertices. As we have seen, intersection vertices have rational coordinates. Eventually, as we have argued, it is necessary to numerically round some of the rational values back to integers. However, it is not natural to round *all* values back to integers at the same time. Perhaps it would make more sense to round only those which have numerators and denominators larger than some threshold. Rounding some coordinates to integers and leaving others alone is mathematically equivalent to rounding to a nonuniform grid. (Since the set of coordinates after rounding is nonuniform, it does not matter how they got that way.)

Finally, even if one rounds all the coordinates simultaneously, it might not make sense to round everything back to integers. Perhaps we might represent each coordinate as a fraction p/q, where p and q are m-bit integers. This set of coordinates densely, but not uniformly, covers the set of values in the range -1 to 1. Similarly, we might use the (very useful) homogeneous coordinates (x, y, w), where x, y, and w are rounded to m bits. This is also yields a nonuniform grid of representable geometric points.

4.2 Nonuniform Grids: Methods. Fortune and Van Wyk, in their work on the LN system [3], give a method for performing infinite precision floating point arithmetic. The representation uses lists of floating point numbers whose (implicit) sum is the number being represented. Each number in the list has a different exponent, and it "covers" a different range of the bits of the represented number.

Canny *et al.*, in their work on rational rotation matrices in two dimensions [2], discuss the use of *continued fractions* for generating good rational approximations to real numbers. This work can also be applied to the problem of finding a lower precision representation for a fraction p/q. It is possible to find p'/q' which approximates p/q to m bits of accuracy such that p' and q' each have about $m/2$ bits of precision.

This author, in his work on rational rotation matrices in three dimensions [15], discusses the use of *basis reduction* to find good rational approximations to unit *quaternions*. Theoretical work of Lovasz [7] implies that basis reduction can generate good rational approximations p_1/q, p_2/q, p_3/q to real numbers α_1, α_2, and α_3. For an approximation with m bits of accuracy, p_1, p_2, p_3, q must have about $0.75m$ bits of precision. This work can be directly applied to numerically rounding homogeneous coordinates. Lovasz *et al.* implies the following result about basis reduction. Given

(x, y, w), it is possible to find (x', y', w') which approximates (x, y, w) with m bits of accuracy such that x', y', w' have about $(2/3)m$ bits each, for a total of about $2m$ bits.

5 Rounding to Nonuniform Grids

This section presents *shortest path rounding*, a method for rounding the vertices of a polygon to a nonuniform grid. This method is optimal from the point of view of introducing the minimum amount of numerical error and the minimum change to the combinatorial structure of the polygon.

We first present the mathematical definition of the of the shortest path rounding, and prove that it works and has the optimal properties claimed above. Next we give a specialization of an algorithm by Guibas *et al.* for calculating the shortest path in a simple polygon. The specialization calculates the shortest path rounding. Finally we discuss some practical issues in its implementation. The statement of this rounding method appeared in a short abstract form [11]. Versions of the shortest path theorem appear in a number of contexts related to the use of floating point arithmetic [9, 12, 14]. The result by Guibas *et al.* was cited, but the specialized version for shortest path rounding has not appeared elsewhere.

5.1 Shortest Path Rounding. Suppose we start with a numerically consistent polygon. For the given numerical values of its coordinates, no two edges intersect except at their endpoints. Next we round the vertices to a nonuniform grid. If a is a vertex, let $\rho(a) = (\rho(a_x), \rho(a_y))$ be the rounding function.

After applying ρ, we might find that vertex $\rho(c)$ is on the right of edge $\rho(a)\rho(b)$, even though c is on the left of ab. Clearly this is a combinatorial inconsistency. Shortest path rounding replaces the single edge $\rho(a)\rho(b)$ by the shortest path that either passes *through* other rounded vertices or has these vertices on the *correct* side. Think of it as if a and b are connected by a tight rubber band. As a and b move to $\rho(a)$ and $\rho(b)$ and as some other vertex c moves to $\rho(c)$, c presses up against and deflects the rubber band, but it does not pass through it.

If two new edges of the rounded polygon become identical they cancel each other in pairs.

Theorem[14] *If ρ is monotonic ($a_x < b_x$ implies $\rho(a_x) \leq \rho(b_x)$), the shortest path rounding produces a numerically and combinatorially consistent polygon.*

Furthermore, since the shortest path obviously takes the fewest turns and deviates the smallest possible amount from the straight line, the deviation is bounded by the maximum deviation of a rounded vertex–the maximum grid spacing.

5.2 Algorithm. Here is the algorithm for the shortest path rounding. The running time is linear in the number of vertices.

We assume that a and b are the *rounded* locations of the endpoints for the edge. Vertices $p_1, p_2, \ldots, p_n$ are the rounded locations of vertices that lie near the edge. They are ordered by increasing value of $p_i \cdot (b - a)$, where "$\cdot$" is the vector dot product. We do not pay for sorting the p_i, since the arrangement algorithm will generate them in sorted order.

When the algorithm makes a *combinatorial* test, p_i is *left* of ab (Primitive 2 in Section 2.1), it is referring to the *original* combinatorial structure of the *unrounded* polygon. Numerical tests: $[a, b, p_i] > 0$ are carried out with the *rounded* values of the coordinates.

In the following, *Path*, *Left*, and *Right* are double-ended stacks. PushHead, PopHead, PushTail, and PopTail do the obvious things. *Path*.Head[0] is the current "head" of the stack. *Path*.Head[1] is the element one away from the "head" end of the stack. After FindShortest is executed, *Path* contains the desired shortest path.

```
AddLeft (p, Path, Left, Right)
    while Left.Size > 1
        a ← Left.Head[1]
        b ← Left.Head[0]
        if [a, b, p] ≤ 0
            Left.PopHead
        else
            break
    Left.PushHead (p)

    if Left.Size = 2 and Right.Size > 1
        while Right.Size > 1
            a ← Right.Tail[0]
            b ← Right.Tail[1]
            if [a, b, p] ≤ 0
                Right.PopTail
                Left.PopTail
                Left.PushTail (Right.Tail[0])
                Path.PushHead (Right.Tail[0])
            else
                break
        Right.PushHead (p)
```

AddRight (p, *Path*, *Right*, *Left*) is analogous to "AddLeft" with the roles of Right and Left switched.

FindShortest $(a, b, p_1, p_2, \ldots, p_n)$
    Path.PushHead(a)
    Left.PushHead(a)
    Right.PushHead(a)

    for $i \leftarrow 1$ to n
        if p_i is *left* of ab
            AddLeft (p_i, *Path*, *Left*, *Right*)
        else
            AddRight (p_i, *Path*, *Left*, *Right*)
    AddLeft (b)
    AddRight (b)
    return *Path*

5.3 Practical Issues. First of all, for any edge ab, it is only necessary to consider vertices c which lie very near to ab. The threshold distance is the maximum rounding amount (the maximum grid spacing in the vicinity), which is usually very small. In general, very few edges will have *any* vertices nearer than the threshold (other than its endpoints). Assuming we can quickly identify these edges, the shortest path rounding will essentially run at the cost of changing all the vertex locations.

Basically, whenever we create a new polygon via the sweepline accommodation algorithm of Section 2, it is easy to keep track of which vertices are very near other edges. This does not appreciably add to the cost of the line segment arrangement algorithm. Each edge structure needs a pointer to a linked list of nearby vertices. Nearby is defined to be a small multiple of the maximum rounding amount. That is all that needs to be done.

There is a slight risk that rounding homogeneous coordinates will not be monotonic. A quick sweep through the set of nearby vertices for an edge can determine any nonmonotonicities. Such vertices should be recomputed using the (safer) rounding of rational coordinates. The continued fraction method is guaranteed to generate monotonic roundings. It is simple to convert rational coordinates to a homogeneous coordinates by clearing the fractions. Unfortunately, increases the required precision from $2m$ to $3m$, but this is not much and it should not happen too often.

6 Results and Future Work

In 1992, we implemented an exact arithmetic system for performing set operations on polygonal regions. This system uses shortest path rounding. It is really only a prototype, but we have been using it for industrial application problems. This summer we hope to implement a more "final" improved version.

The current system uses double precision numbers to store integers. This gives something like 50 bits of precision. Since computing vertex locations causes a four-fold increase in precision, the system can handle integer inputs only in the range -2048 to 2048. This is good enough for our applications, but we really need to use a technique like Fortune and Van Wyk's LN system to boost the precision we can handle. The system is uncrashable (barring undiscovered bugs in the exact algorithm!).

If anyone wants to experiment with the system, please contact me at vjm@cs.miami.edu.

References

[1] J. L. Bentley and T. Ottmann. "Algorithms for Reporting and Counting Geometric Intersections." *IEEE Transactions on Computing*, C28 (1979), pp. 643–647.

[2] J. Canny, B. Donald, E.K. Ressler. A Rational Rotation Method for Robust Geometric Algorithms. *Proceedings of the Eighth Symposium on Computational Geometry*, ACM, pages 251–160, June 1992.

[3] S. Fortune, C. Van Wyk. "Efficient Exact Arithmetic for Computational Geometry." *Proceedings of the Symposium on Computational Geometry*, ACM, 1993.

[4] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *27th Annual Symposium on the Foundations of Computer Science*, pages 143–152, IEEE, October 1986.

[5] L. J. Guibas and J. Hershberger and D. Leven and M. Sharir and R. E. Tarjan. "Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons". *Algorithmica*, vol. 2, 1987, pp. 209–233.

[6] J. Hobby. "Practical Segment Intersection with Finite Precision Arithmetic". *Manuscript*, AT&T Bell Labs, October 1993.

[7] L. Lovasz. *An Algorithmic Theory of Numbers, Graphs and Convexity*, CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.

[8] V. J. Milenkovic and L. R. Nackman. "Finding Compact Coordinate Representations for Polygons and Polyhedra." *IBM Journal of Research and Development*, vol. 34, no. 35, September 1990, pp. 753–769.

[9] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.* Technical Report CMU-CS-88-168, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, July 1988.

[10] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[11] V. J. Milenkovic. "Rounding Face Lattices in the Plane." *First Canadian Conference on Computational Geometry*, Montreal, Quebec, Canada, August 21-25, 1989 (abstract).

[12] V. J. Milenkovic. Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections Using Rounded Arithmetic, *30th Annual Symposium on the Foundations of Computer Science*, IEEE, pages 500-506, October 1989.

[13] V. J. Milenkovic. "Rounding Face Lattices in d Dimensions." *Proceedings of the Second Canadian Conference on Computational Geometry*, Jorge Urrutia, Ed., University of Ottawa, Ontario, August 6-10, 1990, pp. 40–45.

[14] V. J. Milenkovic. Robust polygon modelling. *Computer-Aided Design*, 25(9):546–566, September 1993.

[15] V. J. Milenkovic and V. Milenkovic. "Rational Orthogonal Approximations to Orthogonal Matrices." *Accepted with revisions* for a special issue of *Computational Geometry: Theory and Applications*, October 1993.