Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*

Victor J. Milenkovic**
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

ABSTRACT
Two methods are proposed for correct and verifiable geometric reasoning using finite precision arithmetic. The first method, data normalization, transforms the geometric structure into a configuration for which all finite precision calculations yield correct answers. The second method, called the hidden variable method, constructs configurations that belong to objects in an infinite precision domain—without actually representing these infinite precision objects. Data normalization is applied to the problem of modeling polygonal regions in the plane, and the hidden variable method is used to calculate arrangements of lines.

1. Introduction

Geometric reasoning using finite precision arithmetic presents great difficulties because of round-off error. Yet reasoning in a finite precision domain is an area worth investigating because finite precision floating point arithmetic is fast, widely available, and widely used in practice. The common alternative, algebraic systems, are not subject to error, but they can be much less efficient. The goal of this work are verifiably correct finite precision implementations of geometric algorithms.

Implementations based on finite precision arithmetic pose two problems. First, as has been stated, a finite precision implementation of a proven geometric algorithm is not necessarily correct; because of round-off error, it may fail on valid input. Second, finite precision arithmetic does not have the power to allow an implementation to exactly match the behavior of an implementation based on infinite precision arithmetic. At best it can retain

*Extensions of the results in this article are described in Tech. Rept. CMU-CS-88-168 with the same title.

**Present address: Harvard University, Center for Research in Computing Technology, Cambridge, MA 02138, U.S.A.

Artificial Intelligence **37** (1988) 377–401

only some of the properties of interest. For example, a polygon modeling system may maintain planar topology, allow lines to intersect no more than once, and determine region areas to within a prespecified accuracy, even though it does not correctly determine the number of vertices of the modeled polygon. Solving these two problems requires several steps. The implementor must

- choose a useful set of properties to be retained by the finite precision implementation,
- design the implementation,
- prove that the implementation has the chosen properties.

This paper proposes two general methods for the .design of correct finite precision geometric algorithms: *data normalization* and the *hidden variable* method. It illustrates the use of these methods with concrete examples ‘of correct implementation designs. The application of the first method, a modeling system for planar polygon regions, has been implemented and is in use as part of a research tool. The application of the second method seriously addresses for the first time the problem of round-off error in calculating the topological arrangement of lines in the plane.

2. Error Resulting from Finite Precision: An Example

This section examines a simple example of the type of error ‘that can occur when an implementation uses the most naive form of floating point computation. Suppose we have seven line equations and we wish to determine the topological arrangement of the lines. If, as in Fig. 1, two points of intersection ***A*** and ***B*** are very close together, round-off error may place them in the wrong order on line ***L***. The result, topologically, is shown in Fig. 2. This arrangement has the topology of a two-holed doughnut. Any algorithm that depends on planar topology would fail if presented with such a data structure.

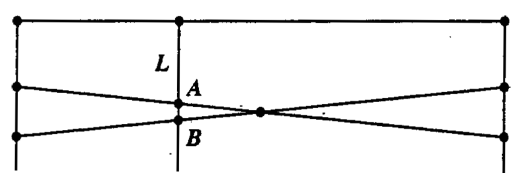

Fig. 1. Error-prone geometric configuration.

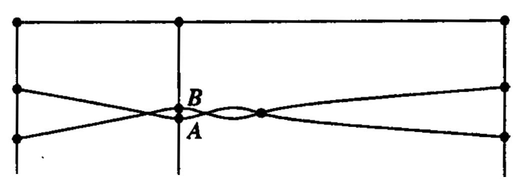

Fig. **2.** Topologically incorrect interpretation.

3. Two Approaches to Finite Precision Implementations

Here then is the crux of the problem: Because of round-off error, we cannot depend solely on numerical tests to determine the structure or topology of a particular geometric object. The choice of topology is sometimes arbitrary, given our lack of information. Yet we want to model geometric domains, such as Euclidean or planar projective geometry, with well-defined topological constraints. This paper proposes two approaches to resolving this problem of ambiguity, *data normalization* and the *hidden variable method.*

– *Data normalization.* Alter the structure and parameters of the geometric object slightly so as to arrive at an object for which all numerical tests are provably accurate. After this normalization process there are no arbitrary choices because calculation always gives a definitive and correct answer.

– *Hidden variable method.* Choose a topological structure so that the following holds true: there exist infinite precision parameters for the object, close to the given finite precision parameter values, such that with the infinite precision values, the problem has the chosen structure. This approach is called the hidden variable method because the topology of the infinite precision version is known but not its numerical values.

In Fig. 3 we see how the normalization method might deal with the problem that was shown in Fig. 1. Very close vertices, which can lead to numerical singularities, are either separated, as on the left, or merged, as on the right. With this method we lose the geometric properties of the lines because the lines have been broken up into noncollinear segments, but we retain the properties of planar polygons.

In Fig. 4, we see two possible solutions under the hidden variable method. On the left is one possible topological arrangement: the three interior lines

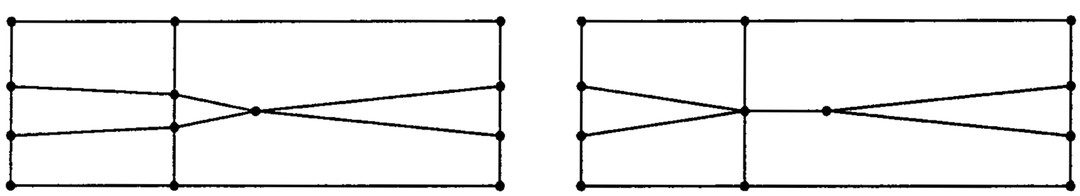

Fig. 3. First method: Altered problem.

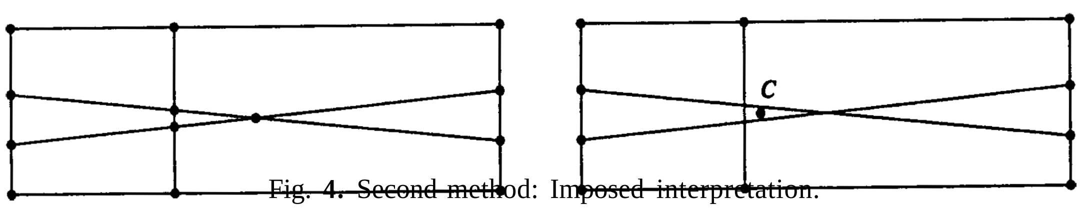

Fig. 4. Second method: Imposed interpretation.

form a triangle. On the right, the three lines pass through a single point C. This choice is possible because one can easily prove that there exists three nearby lines which are indeed concurrent at point C. (The diagram looks awkward because the error has been magnified so as to be visible.) An erroneous interpretation as was seen in Fig. **2** is impossible because no set of "hidden lines" will form that configuration. At least some of the constraints of the domain (Euclidean geometry) are satisfied in that pairs of lines intersect exactly once and the object has planar topology. However we do not have access to the (infinite precision) parameters of the actual lines which satisfy the topology we have chosen. Thus it may be harder to extract information from the model.

The examples given in Sections **2** and 3 are meant to give the reader some idea of the sort of geometric error that can occur and the different means by which the error may be avoided. The following sections contain rigorous definitions of verifiable finite precision implementations. Section 4 contains a model of finite precision arithmetic which can be used as a replacement for the axioms of real arithmetic. The model chosen is perhaps the most pessimistic possible in the sense that nothing is known with greater accuracy than the maximum round-off error. Under this model, the two methods of verifiable geometric computation are demonstrated by means of concrete examples. In Section 5, the normalization method is applied to the problem of modeling polygonal regions in the plane, and in Section 6, the hidden variable method is used to construct the arrangement of lines in the plane. In each case the design is verifiable under the assumptions of the given model of finite precision arithmetic, and the satisfaction of a chosen set of properties is guaranteed.

4. Model of Finite Precision Computation

In order to reason about round-off error, some model of finite precision computation is necessary. The methods of this paper are based on the simplest assumption that round-off is a bounded but random error added to the result of each computation. In evaluating a complicated expression, multiple round-off errors will generally cancel each other, but for purposes of proving correctness, the maximum possible total error must be used.

The implementor must derive an error bound ϵ. This bound depends on the algorithm being implemented and the type of finite precision arithmetic used to implement it. For every expression evaluated by the implementation, the following must hold true:

- The error may be as large as ϵ.
- The error is no larger than ϵ.
- No program can assume an accuracy of a derived result greater than can be verified under the first assumption.

As an example of the last property, suppose an algorithm must calculate the intersection of two lines. The only way to verify an answer under the finite precision model is to evaluate the expressions which determine the distance of the intersection point to each line. This evaluation may differ from the correct value by as much as ϵ. At best, the calculated intersection point may be any point within ϵ of the two lines, possibly quite far from the actual intersection point if the lines form a small angle. In Fig. 5, point **Z** is the true intersection, but the calculated intersection may be as far away as **C.**

The value of ϵ depends on two factors: the maximum round-off error per arithmetic operation and the number of arithmetic operations per expression. Suppose we use a floating point representation with q bits of accuracy. Evaluating any expression involving quantities of magnitude M can result in errors on the order of $2^{-q}M$. Suppose we bound the magnitude of all real numbers that an implementation uses by some value MAX. Then the smallest guaranteed significant quantity is,

$$\text{SIG} = 2^{-q}\text{MAX}.$$

An expression such as the area of a triangle $p_1 p_2 p_3$,

$$\text{Area} = \tfrac{1}{2}(p_1 \times p_2 + p_2 \times p_3 + p_3 \times p_1),$$

involves six multiplications and five additions (or subtractions). Assuming that the additions can be performed with total error at most $\frac{1}{2}$SIG and that each multiplication can have round-off error no more that $\frac{1}{2}$SIG, the maximum possible error in the computation of the area is,

$$\tfrac{1}{2}(\tfrac{1}{2}\text{SIG} + 6 \cdot \tfrac{1}{2}\text{SIG}) < 2\text{SIG}.$$

Similarly, most expressions can be evaluated so as to have error no more than a small constant times SIG. To determine the value of ϵ, the implementor looks at the number of round-offs in the most complex expression evaluated by the system.

Given two quantities calculated under this model of computation, a geometric reasoning system can sometimes show that one is greater than the other or that one is less than the other, but it can never show that the two quantities are

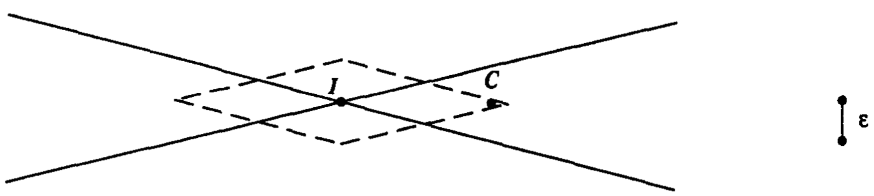

Fig. 5. Error in calculated intersection position.

equal. It is hard to imagine a more pessimistic viewpoint, but it is interesting to see what can be done with these assumptions.

5. Data Normalization

This paper illustrates the method of data normalization by applying it to the problem of modeling polygonal regions in the plane. Central to this approach is a set of rules which a validly normalized object must satisfy. These rules can be correctly tested using finite precision computations. The system provides four types of operations:

- *make:* creation of new normalized objects;
- *move:* translation and rotation operations;
- *combine:* union, complement, and all the other regularized set operations on planar polygonal regions;
- *examine:* a point-in-region predicate, for example.

These operations generate normalized objects so long as they are given normalized objects as input. Since normalization is a precondition and post-condition of all the system operations, the system cannot enter an invalid (unnormalized) state.

The system contains the following 'types of objects,

- *vertices:* ordered pairs of finite precision values representing points in the plane;
- *edges:* ordered pairs of vertices representing oriented line segments.

As shown in Fig. 6, the interior of a polygonal region is defined to lie to the left of its bounding edges.

To define this system, we must choose a value of ϵ such that the distance between a point and another point and the distance between a point and a line segment can be calculated with accuracy $\frac{1}{10}\epsilon$. Given this value of ϵ, the five normalization rules are:

(1) No two vertices are closer than ϵ.

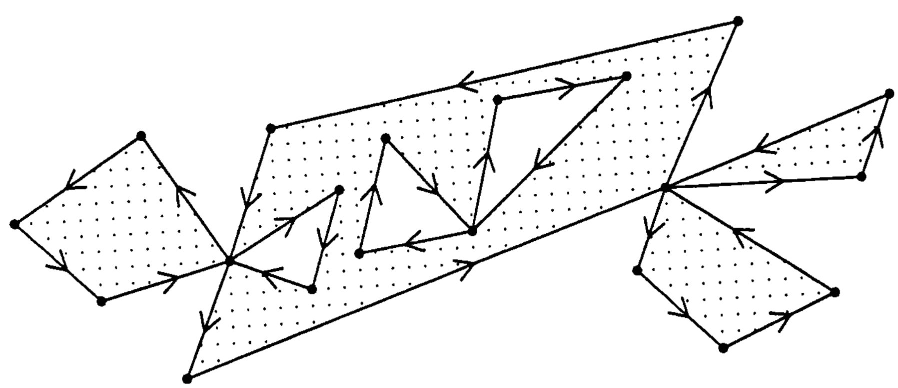

Fig. 6. Example polygonal region.

(2) No vertex is closer than ε to an edge of which it is not an endpoint.
(3) No two edges intersect except at their endpoints.
(4) For each vertex, the angularly sorted list of edges containing that vertex alternates between incoming edges and outgoing edges.
(5) For each point in the plane, the *topological winding number* (defined below) is either 0, 1, or undefined.

The first rule can be checked under all circumstances; the second can be checked so long as the first rule holds; and in general, it is possible to check a rule so long as all the previous rules in the list are satisfied. Once the first two rules are satisfied, the model is no longer subject to topological ambiguities. For example, a set of edges which share a common endpoint can be sorted by angle, and it can be determined whether **a** vertex lies inside or outside a closed loop of edges. Thus the rules after the first two are basically the same as those of an infinite precision implementation.

5.1. Topological winding number

The *topological winding number* indicates the exterior, interior, and boundary of a polygonal region. This winding number can be defined as follows:

Definition 5.1. The *topological winding number* of a point p with respect to a polygon P: if p lies on some edge of P, the winding number is undefined. Otherwise, let L be the horizontal line (parallel to x-axis) through p, and let R be the ray extending to the right of P. As in Fig. 7, an edge AB *crosses* R positively if A is below L and B is on or above L and AB intersects L to the right of p. Edge AB negatively crosses R if BA positively crosses R. The number of positive crossings minus the number of negative crossings is the winding number of p.

In the finite precision version, the winding number can be calculated accurately if p is no closer than $\frac{1}{4}\varepsilon$ to any vertex or, edge of P.

Except on the boundary where it is not defined, the winding number has an integral value, For an arbitrary object satisfying rules (1)–(4), this value may be other than 0 or 1, but rule *(5)* states that the winding number can take on only these values, and so the polygon has a well-defined exterior and interior. Rules **(1)–(4)** make assertions about finite sets, and therefore are easily shown to be decidable. Rule (5), on the other hand, makes a statement about all

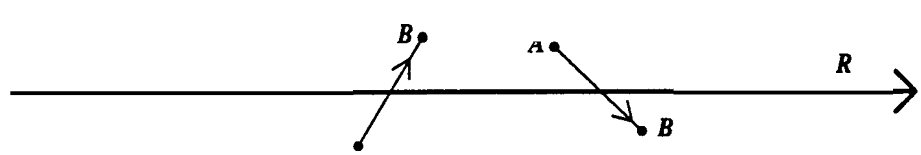

Fig. 7. Edges crossing **ray.**

points in the plane, including points for which the topological winding number cannot be calculated using finite precision. The following theorem reduces the set of points to be tested to a finite set.

Theorem 5.2. *If rules* (1)–(4) *hold and the topological winding number of every point* $\frac{1}{4}\varepsilon$ *to the left of the midpoint of each edge equals* 1 *(see Fig.* 8), *then rule (5) holds.*

Actually, if we partition the edges into connected components, it is sufficient to verify the condition of Theorem *5.2.* for just one edge from each component. Thus rule *(5)* can be tested by only a few applications of the finite precision winding number function.

One final note about the topological winding number: if the polygonal region is unbounded, the value defined by Definition 5.1 will be 0 for points in the exterior of the region and −1 for points in the interior. One can distinguish this case from the bounded case by testing a point as in Fig. 8. The point p should always be inside the region.

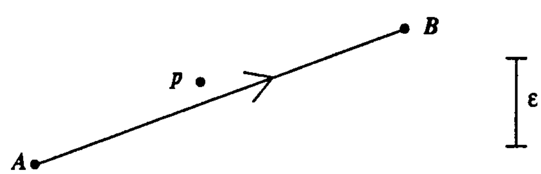

Fig. 8. Test point for **rule *(5)*.**

5.2. Accommodation

Three out of the four basic types of operations on polygonal regions—creation, transformation, or combination-can lead to violations of rules (1) and *(2)*. The polygon(s) being operated on must be altered to accommodate new or transformed vertex locations which may lie within ε of the current vertices and edges of the polygon(s). **A** basic operation called *accommodation* alters a polygon to accommodate a new vertex, using two more primitive operations *vertex shifting* and *edge cracking*. Normalization of a polygon consists of applying accommodation to the polygon for each vertex which violates rules (1) and (2).

For example, one of the most difficult operations on polygonal regions is the union. In Figs. 9 and 10 we see how vertex shifting and edge cracking allow one polygon to accommodate the vertices of the other. The following pseudo-program defines the operation of accommodation.

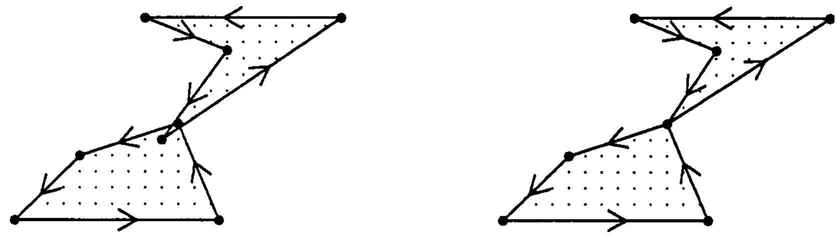

Fig. 9. Vertex shifting.

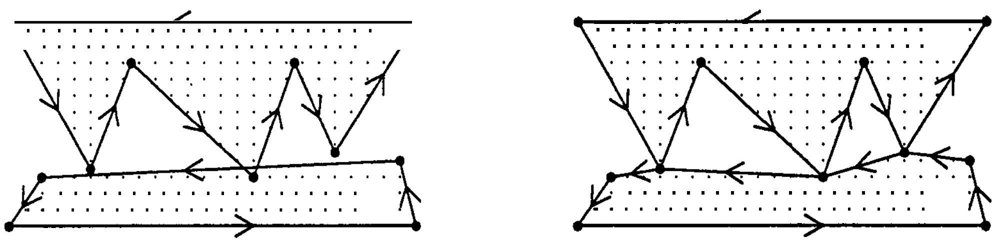

Fig. 10. Edge cracking.

ACCOMMODATE(polygon *P*, vertex *V*)
  {P satisfies rules (1) through (5)}
  SHIFT-VERTEX(*P*, *V*)
  {P satisfies rule (1) and normalization invariant}
  While there exists some edge ***AB*** of P within ε of some vertex
    {P satisfies rule (1) and normalization invariant}
    CRACK-EDGE(***P***, ***AB***)
  {P satisfies rule (1) and rule (2) and normalization invariant}
  {P satisfies rules (1)–(5)}

The following sections define *vertex shifting*, *edge cracking* and the *normalization invariant.* Subsequent sections contain proofs that the assertions in this pseudocode are true and that the while-loop terminates.

5.2.1. *Vertex shifting*

Suppose we have a polygon P and a new vertex ***V*** not part of P. Shift each vertex of P which lies within ε of ***V*** to coincide with the location of ***V***, and eliminate any double edges introduced by this shifting. At this point, the polygon satisfies rule (1). Note that shifting a vertex involves identifying it with the vertex it has been shifted to (otherwise we would have multiple vertices at the same location). The polygon may fail to satisfy rule (2) because

- the new vertex (and hence the shifted vertices) may be within ε of an edge,
- edges with a shifted vertex endpoint may have been moved to within ε of a vertex.

These two sources of violations of rule (2) are mutually exclusive for the following reasons. An edge either has a shifted vertex as an endpoint or it does not. If it does not, it has not been changed, and thus only the first type of violation can occur. If the edge does have a shifted vertex as an endpoint, then after shifting it has V as an endpoint. Since every shifted vertex has been shifted to V, any vertex which lies off the edge after shifting must be an unshifted vertex.

5.2.2. *Edge cracking*

Any edge which passes within ε of a vertex violates rule (2) of the normalization rules. The following steps define a process called *cracking* which eliminates the offending edge.

Step 1. Select all vertices that lie within ε of the edge AB to be cracked.
Step 2. Sort the vertices according to the position of their projections onto AB. Let the sorted list be $V,,V,, \ldots ,V,.$
Step 3. Replace AB by edges $AV,, V_1V_2, \ldots ,V_kB$, and eliminate any double edges that result.

The basic idea is to crack edges until no edge is a candidate for cracking. Since edge cracking does not move vertices (although it may eliminate some), it does not introduce violations of rule (1). Cracking an edge may introduce more edges to be cracked, but if the algorithm terminates, the resulting object must satisfy rule (2).

Incidentally, we may not be able to sort the vertices which lie near an edge if two of the vertices project to the same or nearly the same point. In this case, it can be shown that one of the two vertices must lie at least $\frac{1}{2}\varepsilon$ closer to the edge than the other. The more distant vertex can be left out of the list of vertices to be cracked to without resulting in a violation of the normalization invariant defined in the next section.

5.2.3. *Normalization invariant*

Designing an invariant for the pseudocode above is tricky because the partial results of vertex shifting and edge cracking do not necessarily satisfy rules (1) and (2). Without these, rules (3)–(5) cannot even be checked using finite precision. The solution to this problem is reminiscent of the hidden variable method described in Section 6.

Definition 5.3. A *polygonal approximation* to an edge AB is a sequence of edges, $AC_1, C_1C_2, \ldots ,C_kB$, such that each C_i lies within $\frac{1}{2}\varepsilon$ of AB, and the projections of the C_i on AB are in sorted order.

Definition 5.4. The *normalization invariant* for the accommodation algorithm is defined as follows: polygon P satisfies the invariant if there exist polygonal approximations to all the edges of ***P*** such that replacing each edge by its approximation results in a polygon which satisfies rules (3)–(5) (in the infinite precision domain).

In order to prove that vertex shifting and edge cracking preserve the normalization invariant, we need the following theorem.

Theorem 5.5. *If* ***AB*** *is an edge and vertices* $V_1, V_2, \ldots, V_k$, *lie farther than* ε *from* ***A*** *and* ***B*** *but closer than* $\frac{1}{2}\varepsilon$ *to* ***AB***, *then there exists a polygonal approximation to* ***AB*** *which passes to the left of any chosen subset of the* V_i *and passes to the right of the others.*

5.2.4. *Proof of assertions*

Does vertex shifting preserve the invariant? Recall that vertex shifting introduces violations of rule (2) by either moving edges toward vertices or moving vertices toward edges, but not both at the same time. Since the shift distance is less than ε, nothing get moved relative to something else by more than ε. Since the polygon satisfied rule (2) to start with, vertices and edges were separated by at least ε before the shifting. Therefore after shifting, no vertex could have been moved a significant distance to the wrong side of an edge, certainly not as far as $\frac{1}{2}\varepsilon$. Theorem 5.5 implies that there must exist polygonal approximations to the edges that pass by such vertices on the correct side, and thus the normalization invariant can be satisfied.

If we wish to prove that cracking an edge preserves the normalization invariant, we must consider three classes of vertices: those which lie on the wrong side of the edge (in the infinite precision domain), those which lie within ε of the edge (according to finite precision calculation), and those which lie further than ε from the edge. All vertices in the first class must also belong to the second because the normalization invariant tells us that the edge need not be deflected more than $\frac{1}{2}\varepsilon$ to pass on the correct side of the vertices in the first class. Thus the vertices in the first class lie within $\frac{1}{2}\varepsilon$ of the edge, and therefore finite precision calculation will show them to be well within ε of the edge. The cracking process replaces the edge with a chain of edges passing through the vertices in the second class. Since the second class includes the first class, the topological inconsistency caused by the first class will be eliminated by the new chain of edges.

The new edges may pass to wrong side of previously consistent vertices, but those vertices must all belong to the third class because vertices in the first and second class are part of the chain of new edges. The new edges are displaced from the eliminated edge by at most ε. Since the vertices in the third class were ε distant from the eliminated edge, they cannot be more than a small distance to the wrong side of the new edges, certainly not as much as $\frac{1}{2}\varepsilon$. As in the case of vertex shifting, Theorem 5.5 implies that the normalization invariant can be satisfied.

At the termination of accommodation, rule (1), rule (2) and the normalization invariant hold. Rules (1) and (2) imply that vertices and edges are separated by at least ε. The normalization invariant implies that there is some way of replacing each edge with a polygonal path such that the resulting object satisfies rules (3)–(5). But the polygonal path does not stray further than $\frac{1}{2}\varepsilon$ from the edge. Therefore, replacing the paths with the edges themselves should result in an object that satisfies rules (1)–(5).

5.2.5. *Termination of edge cracking*

The only thing remaining to show is that edge cracking does indeed terminate. In the edge cracking stage of the accommodation algorithm, the choice of the next edge to crack is arbitrary. There is a way of ordering the choices so as to assure termination of cracking. We can first perform those crackings which will increase the area of polygonal region. Then, we can perform the crackings which will decrease the area. It can be shown that a cracking that decreases the area cannot create the need for a cracking that increases the area. Since the set of vertices is fixed and finite, there are only a finite number of edge configurations, one of which will have the largest area and one, the smallest area. Thus both the area-increasing stage and the area-decreasing stage will terminate.

While edges are being cracked, the polygonal region may not have an area, even in infinite precision, because the polygon does not necessarily satisfy rules (3)–(5). The polygon whose existence is implied by the normalization invariant does have an area, but it is not unique. The area we seek is the limit of valid approximating polygons whose total edge lengths tend to a minimum. Incidentally, it is the opinion of the author that edge cracking terminates regardless of the order the edges are cracked.

5.3. Error bounds

We have shown that accommodation can be performed correctly, but how much error does it introduce? The measure of error is the area of discrepancy between the original polygonal region and the new normalized region.

Vertex shifting introduces a small amount of error. It can be shown that the region of discrepancy has area at most εp where p is the length of the perimeter.

Edge cracking, on the other hand, can introduce quite a considerable amount of error. In certain pathological cases, the entire region can disappear! The worst-case error is of order $n\varepsilon p$ where n is the number of vertices in the object. The worst case occurs when each vertex is nearly within ε of some other vertex, and thus a single vertex shift can cause a cascade of edge cracking. If the vertices and edges are separated by at least 2ε to start with, the region of discrepancy has area at most εp as in the case of vertex shifting.

5.4. Implementing the union

As an illustration of how accommodation can be used to implement operations on polygonal regions, let us consider the union: given polygonal regions P and Q, we wish to generate the region which represents the set of points lying in either P or Q.

The first step is to accommodate Q to all the vertices of P by repeated calls to the accommodation function. Then P must be accommodated to the vertices of Q.

At this point, the intersection points can be calculated and inserted. If edge AB of P and edge CD of Q intersect at point I, then replace AB by AI and IB and replace CD by CI and ID. Polygons P and Q must be accommodated to the new vertex I.

Finally, by applying the topological winding number function to the midpoints of edges, determine which edges of P lie outside or on the boundary of Q and which edges of Q lie outside or on the boundary of P. These are the edges of the union region.

As presented, this algorithm is rather inefficient. However an optimization similar to the one mentioned at the end of Section 5.1 can be made. If an edge is shown to be part of the boundary of the union polygonal region, then any edge connected to it via unshared vertices (vertices of P or of Q but not of both) is part of the boundary also.

5.5. Advantages, disadvantages, and an implementation

This particular example shows that data normalization is a viable method for correctly implementing geometric algorithms with finite precision. Unfortunately, many properties of the infinite precision domain are lost by normalization. The resulting objects are indeed planar and polygonal, but each normalization introduces a bounded amount of error which can be measured as the area of the discrepancy between the normalized and unnormalized regions. The total error in a sequence of operations grows with the number of normalizations, and it can be quite large in the worst case. The order of application of accommodation and edge cracking is arbitrary, and the result of the algorithm depends on the order in which these operations are applied. Data normalization is also difficult to generalize to more complex domains, such as curved objects. Still, the polygonal region modeling system summarized here is robust and satisfies well defined properties. **A** version of the implementation is a part of a modeling system for the VLSI manufacturing process designed at IBM Yorktown [4]. To date is has not failed in any way.

6. Hidden Variable Method

We illustrate the hidden variable method by applying it to the problem of determining the topological arrangement of n lines represented by their

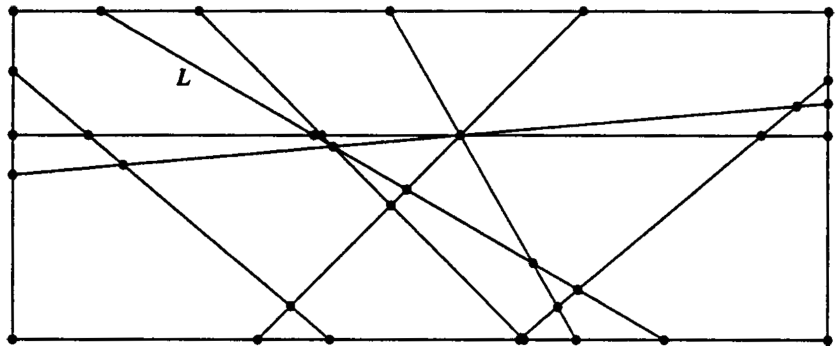

Fig. 11. Arrangement of lines.

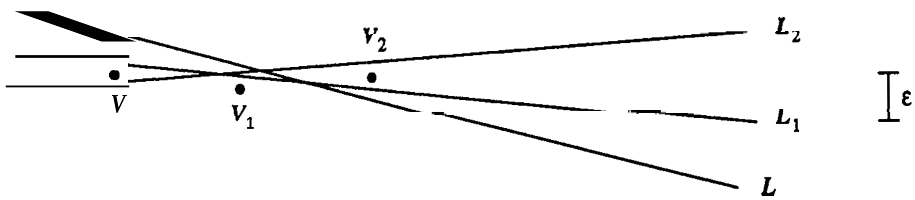

Fig. 12. Difficult case for arrangement algorithms.

equations. The output of the algorithm is a set of (finite precision) vertex locations and a data structure representing the topology of the arrangement (see Fig. 11).

The common methods currently used for calculating arrangements [1,2] are sensitive to round-off error. Figure 12 shows a configuration that can occur under the model of finite precision arithmetic used by this paper. If the implementation calculates that lines L_1 and L, intersect at V and that line L lies above V as shown, then L ought to intersect line L, before line L,. Yet with the value of ε shown, L could intersect L_1 at V_1 and L, at V_2, with V_1 appearing well before V_2 on L. The positions of V, V_1 and V_2 are impossible to reconcile with a planar topology.

6.1. Arrangements

Formally, the input of the arrangement algorithm is a set of line equations $\mathscr{L}$ expressed in finite precision. The output of the algorithm contains,

- a set of vertices $\mathscr{V}$,
- a set of edges $\mathscr{E}$,
- some symbolic representation $\mathscr{T}$ of the topological arrangement of the vertices and edges (Guibas and Stolfi [3] have devised a very good representation in their paper about creating Voronoi diagrams).

Each vertex has a numerical location (x, y). Each edge in $\mathscr{E}$ is associated with a *bundle* of lines (a subset of $\mathscr{L}$) which represents, by definition, the lines that contain that edge. The vertex endpoints of an edge must satisfy the line equations of the lines in that edge's bundle with an error no greater than ε.

The arrangement is limited to a bounding box which represents the maximum allowable magnitudes of the x- and y-coordinates. For every line in $\mathscr{L}$, the set of edges it contains forms a single unbroken chain from one end of the bounding box to the other.

6.2. Ideal case

Ideally we want there to exist a hidden, infinite precision set of lines $\mathscr{L}'$ and vertices $\mathscr{V}'$ which are "close to" the input lines $\mathscr{L}$ and the output vertices $\mathscr{V}$ and which truly satisfy the topological arrangement $\mathscr{T}$. Formally, there 'must exist mappings from $\mathscr{L}$ to $\mathscr{L}'$ and from $\mathscr{V}$ to $\mathscr{V}'$ which satisfy the following properties

(1) The line mapping is many-to-one and thus partitions $\mathscr{L}$ into equivalence classes—lines which map to the same line in $\mathscr{L}'$—are equivalent.

(2) The vertex mapping is one-to-one.

(3) Each bundle must be an equivalence class of the line mapping. In this way, the mapping from $\mathscr{L}$ to $\mathscr{L}'$ induces a mapping from $\mathscr{E}$ to $\mathscr{L}'$—each edge is mapped to the unique element of $\mathscr{L}'$ to which all the lines in that edge's bundle are mapped.

(4) If a vertex is an endpoint of an edge, the image of that vertex under the vertex mapping must lie on the line to which the induced edge mapping maps the edge.

(5) The elements of $\mathscr{L}'$ and $\mathscr{V}'$ are indeed arranged according the topology $\mathscr{T}$.

The reader can compare this ideal case with the practically realizable approximations of Sections 6.4 and 6.5.

Unfortunately, finite precision computations very probably do not have the power to assure the ideal result. Only an infinite precision algebraic system will always generate arrangements of lines that satisfy every theorem of Euclidean geometry. If it is not possible to model geometric lines using finite precision, we must choose a plane curve to model which has some but not all the properties that lines have. This paper focuses on a defining property of lines called monotonicity. The curves modeled by the hidden variable method in this section satisfy weaker properties called xy-monotonicity and approximate monotonicity.

6.3. Monotonicity

We can generalize the one-dimensional property monotonicity to two-dimensional curves and sequences of points.

Definition 6.1. The property ***v**-monotonicity*, where $\boldsymbol{v}$ is a nonzero direction vector in the plane, is defined for ordered sets of points such as curves and sequences. Let $\gamma(t)$ be a curve in the plane. The curve y is u-monotonic if the inner product of $\boldsymbol{v}$ and $\gamma(t)$ is either nondecreasing or nonincreasing with t. The definition for sequences of points is analogous.

Intuitively, this definition states that u-monotonic curves and sequences tend in only one direction parallel to $\boldsymbol{v}$ and do not backtrack.

A curve or sequence can be monotonic with respect to more than one direction. For example, if a curve or sequence is monotonic with respect to the direction of the x-axis and the direction of the y-axis, then it is said to be xy-monotonic. It is decidable using only finite precision calculations whether a sequence is xy-monotonic because xy-monotonicity can be checked using only comparisons of x-coordinates against other x-coordinates and y-coordinates against other y-coordinates. Finite precision comparisons (in particular floating point comparisons) are always accurate.

If a curve or sequence is u-monotonic with respect to all direction vectors $\boldsymbol{v}$, then it is simply said to be monotonic. Lines are the only monotonic curves, and a monotonic sequence of points is always collinear.

Finally, a curve is *approximately·monotonic* if it does not backtrack more than ε with respect to any direction.

Definition 6.2. **A** one-dimensional function f is approximately nondecreasing if

$$s > t \quad \text{implies} \quad f(s) > f(t) - \varepsilon.$$

The definition of approximately nonincreasing is analogous. **A** two-dimensional curve is approximately monotonic if it is either approximately nonincreasing or approximately nondecreasing with respect to every direction vector.

In Fig. 13 we see an example of an approximate monotonic curve which approximates a line.

In order to implement the hidden variable method, we will proceed from the bottom up. The system design described in the next section models xy-monotonic curves, and the arrangements generated by this lower-level system satisfy only a few geometric properties. The higher-level design described in

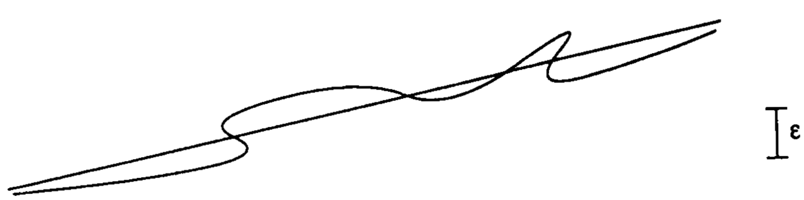

Fig. 13. Approximate monotonic curve.

the subsequent section uses the lower-level system as a subroutine. This higher-level system models approximate monotonic curves, and the arrangements it generates satisfy a larger set of properties included axioms and theorems of Euclidean geometry.

6.4. Lower level: Modeling xy-monotonic curves

The system described in this section treats the input lines as xy-monotonic curves. The hidden variables of the problem are the unrepresented shapes of xy-monotonic curves which approximate the input data lines. The system derives vertex locations and a topological structure such that the hidden curves pass through the derived vertices and are arranged according to the derived topology.

6.4.1. Definition: xy-monotonic arrangement

The following formally describes a valid arrangement of xy-monotonic curves. It can be compared with the ideal case in Section 6.2.

(1) As before, the edges contained by a given line form a chain from one boundary of the bounding box to the other. The vertices in each chain must form an xy-monotonic sequence of points.

(2) Each edge maps to an xy-monotonic curve which has the same endpoints as the edge.

(3) The curve belonging to an edge does not deviate by more than ε from any of the lines in the bundle of that edge, and it does not intersect any other curve except at its endpoints.

Since each line in $\mathscr{L}$ contains an unbroken chain of edges, the curves of these edges can be joined to form one long monotonic curve with the same endpoints on the boundary as the line and which stays within $\varepsilon_{\mathrm{XYM}}$ of the line. The value $\varepsilon_{\mathrm{XYM}}$ (XYM stands for "xy-monotonic") can be set to twelve times the maximum error in the calculated distance from a point to a line.

6.4.2. Some terminology

For the purposes of describing xy-monotonic sequences, let us define the following order relations among points in the plane. Define the relations n, **s**, ***e***, and **w**, pronounced "is north of", "is south of", "is east of", and "is west of", respectively.

$$
\begin{aligned}
\langle x_1, y_1\rangle \mathbf{n} \langle x_2, y_2\rangle \quad &\text{iff} \quad y_1 \geq y_2\,,\\
\langle x_1, y_1\rangle \mathbf{s} \langle x_2, y_2\rangle \quad &\text{iff} \quad y_1 \leq y_2\,,\\
\langle x_1, y_1\rangle \mathbf{e} \langle x_2, y_2\rangle \quad &\text{iff} \quad x_1 \geq x_2\,,\\
\langle x_1, y_1\rangle \mathbf{w} \langle x_2, y_2\rangle \quad &\text{iff} \quad x_1 \leq x_2\,,
\end{aligned}
$$

The relations nw, ne, sw, and se, pronounced "north-west", "north-east", "south-west", and "south-east", respectively, are formed by combinations of the operations above.

6.4.3. *Line resolution*

The algorithm for generating models of xy-monotonic curves is called *line resolution.* The following pseudocode works only for lines with positive slope—those which leave the bounding box north-east of the point at which they enter the box. Furthermore, the lines must be oriented so that points north or west of the line lie to the left of the line and points south or east of the line lie to the right of the line.

In the general case, lines can be re-oriented to conform to the orientation condition above. The code for lines with negative slopes is analogous. Not included here is the method for discovering and calculating the intersection of negative slope lines with positive slope lines. This case is very simple to model because a positive slope (x nondecreasing, y nondecreasing) xy-monotonic curve and a negative slope (x nondecreasing, y nonincreasing) xy-monotonic curve can intersect in at most a single vertical or horizontal line segment.

The state of the algorithm is stored in a data structure keyed by a vertex and a line and expressed as $\text{SIDE}(V, L)$. For each vertex V and line L, $\text{SIDE}(V, L)$ returns a record containing a two-bit field with possible values: *unknown, left, right,* or *on (left* and *right).* The function $\text{LEFT}(V, L)$ gives access to the first bit and has value *true* if $\text{SIDE}(V, L)$ is *left* or *on*. The function $\text{RIGHT}(V, L)$ gives access to the second bit and has value *true* if $\text{SIDE}(V, L)$ is *right* or *on*. Finally, the function $\text{ON}(V, L)$ accesses the logical *and* of both bits, and it has value *true* if $\text{SIDE}(V, L)$ is *on*.

The function $\text{EVAL}(V, L)$ evaluates the signed distance between the vertex V and the line L using finite precision arithmetic. In the ideal case $\text{LEFT}(V, L)$ would be synonymous with a nonnegative value for $\text{EVAL}(V, L)$. However, in the presence of round-off error, we can only manage the following approximation conditions.

- $\text{LEFT}(V, L)$ implies that $\text{EVAL}(V, L) > -\varepsilon_{\text{XYM}}$,
- $\text{RIGHT}(V, L)$ implies that $\text{EVAL}(V, L) < \varepsilon_{\text{XYM}}$,
- $\text{ON}(V, L)$ implies that $|\text{EVAL}(V, L)| < \varepsilon_{\text{XYM}}$.

To these we add the following *logical conditions.*

- P **nw** Q and $\text{LEFT}(Q, L)$ implies that $\text{SIDE}(P, L) = \textit{left}$.
- P' **se** Q' and $\text{RIGHT}(Q', L)$ implies that $\text{SIDE}(P', L) = \textit{right}$.

Figure **14** illustrates the rationale for these conditions.

The following three procedures make up the line arrangement algorithm.

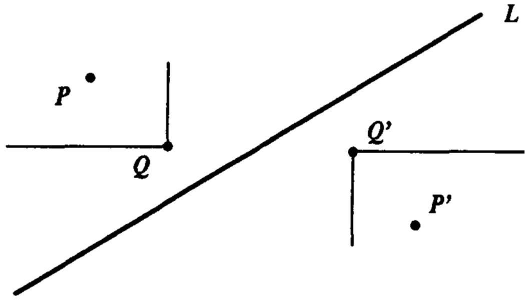

Fig. 14. Logical conditions on SIDE(P, L).

INSERT-VERTEX(vertex V)
  For each line L
    If there exists vertex V_{LEFT} such that
      (V **nw** V_{LEFT} and LEFT(V_{LEFT}, L))
        LEFT(V, L) $\leftarrow$ *true*
    If there exists vertex V_{RIGHT} such that
      (V **se** V_{RIGHT} and RIGHT(V_{RIGHT}, L))
        RIGHT(V, L) $\leftarrow$ *true*
    If SIDE(V, L) = *unknown*
      if EVAL(V, L) $\geqslant 0$
        LEFT(V, L) $\leftarrow$ *true*
      else
        RIGHT(V, L) $\leftarrow$ *true*

boolean FIND-INTERSECTION()
  {See Figure 15.}
  If there exists line L_1, L, and vertex V_1, V_2 such that
    LEFT(V_1, L,) and RIGHT(V_2, L,) and
    RIGHT(V_1, L_2) and LEFT(V_2, L_2)
      {Check if intersection exists already.}
      If there does not exist vertex V_I such that
        V_I **ne** V_1 and V_I **sw** V_2 and
        ON(V_I, L_1) and ON(V_I, L_2)
          vertex V_I = CALCULATE-INTERSECTION(L_1, L,, V_1, V_2)
          ON(V_I, L_1) $\leftarrow$ *true*
          ON(V_I, L_2) $\leftarrow$ *true*
          INSERT-VERTEX(V_I)
          return *true*
  return *false*

RESOLVE-LINES(set-of line LINES)
  {Bounding box has four lines and four vertices.}
  CREATE-BOUNDING-BOX()
  While FIND-INTERSECTION()

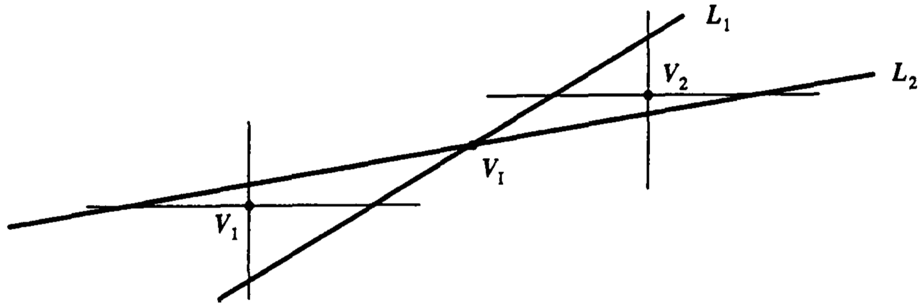

Fig. 15. Intersection condition.

6.4.4. *Discussion and proof*

The intersection condition in FIND-INTERSECTION is illustrated in Fig. 15. Lines L_1 and $L,$ must intersect within the rectangle with diagonal points V_1 and V_2. What is more important is that even if $L,$ and $L,$ were replaced by *xy*-monotonic curves, the curves would still have to intersect within the rectangle. Incidentally, $V,$ or V_2 can lie on one of the lines, but if either vector lies on both lines then it could act as V_I.

We have not seen yet how to implement CALCULATE-INTERSECTION using finite precision arithmetic. In most cases, the straightforward method of solving simultaneous line equations results in valid intersection. In the rare'case that this calculated intersection point *IC* lies outside the rectangle, a small vertical (or horizontal) shift will take it to a point *IR* on the boundary of the rectangle, as depicted in Fig. 16. The actual details of this algorithm are somewhat tedious, but in all cases a point *IR* can be found that lies within the rectangle and satisfies,

$$|\mathrm{EVAL}(IR, L_1)|, |\mathrm{EVAL}(IR, L_2)| < \varepsilon_{\mathrm{XYM}} .$$

Figure **17** illustrates another type of problem we may encounter. If RIGHT$(V, L,)$ and RIGHT(V, L_2) and *IR* se $V,$ then we cannot set SIDE$(IR, L,)$ and SIDE$(IR, L,)$ equal to *on*. In this case we simply throw away *IR* and use V as the vertex of intersection.

As in the case of edge cracking, there is the question of whether the

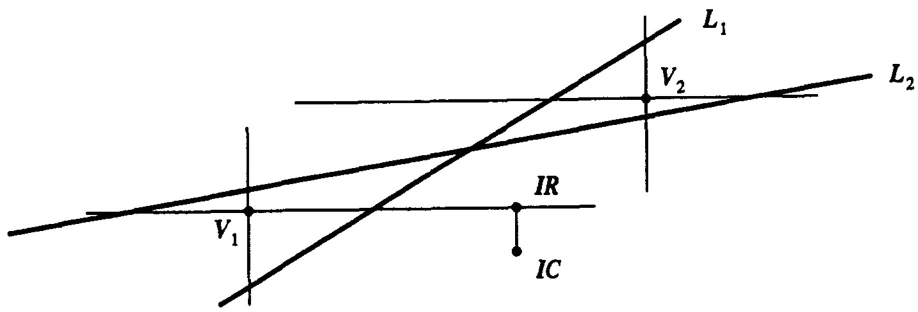

Fig. 16. Calculated intersection lies outside of rectangle.

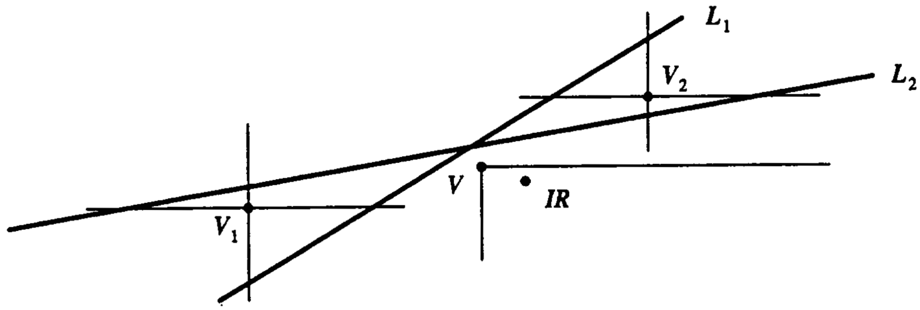

Fig. **17.** Vertex V invalidates calculated intersection ***ZR.***

algorithm terminates. Each intersection adds a new vertex, but there are only a finite number of finite precision vertex locations inside the bounding box; thus the process of finding intersections must stop. Of course, one can find much better bounds.

The output of the line resolution algorithm is a set of values for SIDE(V, L) which defines a topology. To convert to the form in Section 6.4.1, we need to introduce edges and bundles. After line resolution, the set of vertices ON a given line can be ordered into a monotonic sequence. By definition, the edges in the topology join consecutive vertices in such sequences. For any given edge AB in the topology, the bundle for that edge is the set of lines L which satisfy ON(A, L) and ON(B, L).

Theorem 6.3. After the termination of line resolution on a set of lines, the values of SIDE(V, L) define a topology. ***For*** this topology, there exist xy-monotonic curves which satisfy the conditions of Section 6.4.1.

The proof of this theorem involves constructing the xy-monotonic curves using infinite precision calculations. The essential step of the proof shows that if two curves intersect, then the condition of function FIND-INTERSECTION can be satisfied. Since the line resolution has terminated, this condition cannot hold.

6.5. Higher level: Approximate monotonic curves

The previous section solves for the arrangement of xy-monotonic curves. This result is unsatisfactory for general purposes because the form of such curves varies with orientation, Horizontal or vertical xy-monotonic curves are much "flatter" than those with orientation angle closer to 45 degrees. The conditions of the previous section allow "lines" (chains of xy-monotonic curves) to intersect more than once or even have sections in common. In these ways, xy-monotonic curves differ considerably from the lines they are supposed to model. **A** second application of the hidden variable method is necessary to arrive at a model which more closely matches the ideal case in Section 6.2.

The result of this higher-level design is the same as the ideal case except that the hidden curves are approximate monotonic curves instead of lines. Thus in the definition in Section 6.2 the set $\mathscr{L}'$ of hidden lines is replaced by a set $\mathscr{A}$ of hidden approximate monotonic curves. The only additional condition is that the elements of $\mathscr{A}$ must not deviate by more than ε_{AM} from the lines they represent.

6.5.1. Tiled highways

Using the xy-monotonic curve modeling system, this design solves a transformed problem. It replaces each line by a pair of parallel lines, each ε_{AM} distant from the original. The value ε_{AM} is at least four times the error value ε_{XYM} of the lower-level system. The higher-level system then solves the $2n$-line problem using the xy-monotonic system with its smaller error value ε_{XYM}. The resulting arrangement is a set of tiled highways, a term which is based on the observation that each line has become a strip cut up by intersecting strips. In Fig. 18 the shaded area is the tiled highway for line L of Fig. 11, and the polygons within the shaded area are the tiles of that highway.

Each tile represents a region within ε_{AM} of a certain subset of $\mathscr{L}$, and thus the existence of a common tile to the highways of lines $L_1, L_2, \ldots, L_k$ symbolically answers the numeric question: is there a vertex within ε_{AM} of these lines? The difficult numeric work has been done by the first stage.

A partial ordering can be defined among tiles on the same highway.

Definition 6.4. Let A and B be tiles on the highway of some line L. Let v be the unit vector parallel to L. If each vertex of B has a larger v-component than any vector in A, then $B > A$. Define $B \geq A$ as the opposite of $A > B$.

The evaluation of the condition $B > A$ involves the finite precision vector inner product. Even so, this partial order is transitive. The relation $A \leq B$ is not transitive, however, but it is introduced as a notational convenience.

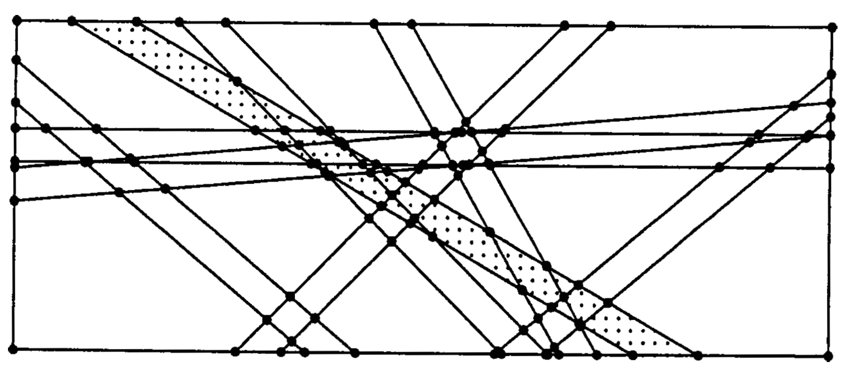

Fig. 18. Tiled highways.

6.5.2. *Paths on the tiled highways*

For each highway, we can create a path that "walks" from one end to the other. This path is purely symbolic (its vertices have no defined numerical location). We can use the following operations to create such paths:

- Split an internal edge of the highway at a new vertex.
- Add an edge joining two such vertices which are on the boundary of a common tile.

Vertices created by the crossing of paths (from different highways) can be reused. At most four edges can meet at any other type of vertex.

The paths on the highways must satisfy the following rules:

(1) A path does not leave its highway, and it crosses tile A before tile B only if $A \leqslant B$.

(2) The path does not cross any edge more than once.

(3) The path does not cross itself.

The second condition assures that there are only a finite number of such paths. Actually, a proof is required that the second condition does not eliminate any meaningful paths; in the case of line modeling it does not because pairs of lines do not cross more'than once. The proof is not included here. Figure 19 shows a close-up of four paths and the intersections between them.

6.5.3. *Converting to standard form*

As one would expect, the paths correspond to approximate monotonic curves, and the points at which paths cross are the vertices in the topology. All other vertices of the tiled highway "scaffolding" (to mix a metaphor) are ignored.

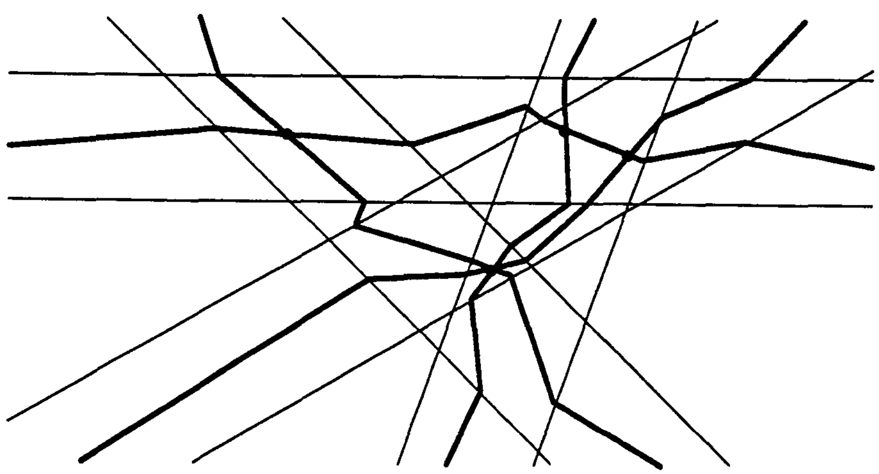

Fig. 19. Constructing paths on the tiled highways.

The finite precision location of each topological vertex is not crucial, but one can always find a point inside the appropriate tile by calculating the mean of the tile's vertex locations. The edges of the topology join consecutive vertices on the paths. One question remains: how can more than one line in $\mathscr{L}$ map to the same path? It turns out that if several lines are close, then it is possible for one path to lie within the highways of all of them. If such a path exists, it can represent all these lines.

Theorem 6.5. *For each set of paths, there exist an set of ideal vertex locations and approximate monotonic curves that satisfy the topology of the paths. For all lines represented by a path, the curve corresponding to that path does not deviate by more than* $\varepsilon_{\mathrm{AM}}$ *from any of these lines.*

This result is pretty obvious because we have constrained each path to stay within $\varepsilon_{\mathrm{AM}}$ of its lines and to pass through tiles in a way that does not violate the partial ordering on tiles.

6.5.4. *Satisfying other properties*

We can enumerate all sets of paths that lead to approximate monotonic curve models. All of these have planar topology, but among these we can choose those which satisfy the condition, two paths cross at once, which is analogous to the axiom of geometry that two lines intersect in at most one point. We can choose to satisfy other results such as Desargue's or Pappus' theorems if we wish.

Even after satisfying various geometric properties of lines, there will still be a number of possible sets of paths to choose from. Among these we can choose the one which minimizes the size of $\mathscr{A}$ first and minimizes the number of vertices second. If more than one set of paths is minimal, we can estimate the "strain", maximum or total deviation from the lines in $\mathscr{L}$, and choose the set of paths that minimizes this deviation measure.

The combination of the two stages of the hidden variable method can thus generate a topological arrangement which satisfies a set of useful geometric properties and minimizes the topological complexity in doing so.

7. Conclusion

The methods of data normalization and the hidden variable method both allow correct finite precision implementations which satisfy a set of useful properties. In Section 5 we saw how a polygonal region could be maintained in a normalized state, with edges and vertices separated by a lower bound distance ε, so that numerical tests would yield correct topological results. The modeling system described in that section generates only valid planar polygonal regions as the result of operations such as union and rotation or translation in the plane. Unfortunately, the normalization step may displace an edge many times the distance ε, and thus normalization can introduce a large deviation from the true infinite precision result.

The hidden variable method described in Section 6 models approximations to geometric lines. These approximations deviate at most ε from the lines they represent. It is a small step, involving only symbolic calculations, from an algorithm for calculating arrangements of lines to a polygonal region modeling system. Such a modeling system would only introduce a small error dependent on the number of lines, not the number of operations. In addition, it would maintain useful properties such as the collinearity of widely separated vertices.

Normalization has the advantage that it is simpler, but it has the potential to introduce more error and it does not retain as many properties of the ideal implementation. The hidden variable method allows more properties to be retained, and it is theoretically interesting in the manner in which it reasons about unspecifiable values. The system in Section 5 is suitable for modeling the growing and shrinking of regions which represent components in the fabrication of VLSI devices. For line-oriented problems, such as the design of buildings and bridges, the methods of Section 6 are more suitable. The normalization method is also very difficult to generalize to more complex domains such as objects with curves or curved surfaces. In the future, I will apply the hidden variable method to the problem of modeling these more complex objects, in particular, planar objects bounded by conic sections and solid objects bounded by quadric surfaces.

ACKNOWLEDGMENT

The research described here was done at AT&T Bell Laboratories, IBM Thomas J. Watson Research Center, and Carnegie-Mellon University. The portion done at CMU was funded by an IBM Manufacturing Research Fellowship. The views and conclusions in this document are those of the author.

I would like to thank Takeo Kanade and Daniel Sleator of Carnegie-Mellon, Michael Wesley of IBM Watson Research Center, and Doug McIlroy of AT&T Bell Laboratories, for their guidance on my research in the area of finite precision geometric reasoning. I am grateful to Takeo Kanade and Mark Stehlik of Carnegie-Mellon for proofreading this paper, and I thank them for their advice on style and presentation.

REFERENCES

1. Edelsbrunner, H. and Guibas, L.J., Topologically sweeping an arrangement, Stanford University Tech. Rept., Stanford, CA 1985).
2. Edelsbrunner, H., O'Rourke, J. and Seidel, R., Constructing arrangements of lines and hyperplanes with applications, in: *Proceedings* 24th Symposium on Foundations of Computer Science (1983) 83–91.
3. Guibas, L.J. and Stolfi, J., Primitives for the manipulation of general subdivisions and the computations of Voronoi diagrams, *ACM Trans. Graph.* **4** (1985) 74–123.
4. Wesley, M. and Koppelman, G., A robot planning system applied to VLSI manufacturing processes, Research Rept. 10510, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (1984).

Received October 1986; revised version received December 1987